partitura

Release 0.4.0

Jun 15, 2021

Contents

CHAPTER 1

Introduction

The principal aim of the *partitura* package is to handle richly structured musical information as conveyed by modern staff music notation. It provides a much wider range of possibilities to deal with music than the more reductive (but very common) pianoroll-oriented approach inspired by the MIDI standard.

Specifically, the package allows for representing a variety of information in musical scores beyond the onset, duration and MIDI pitch numbers of notes, such as:

- pitch spellings,
- symbolic duration categories,
- and voicing information.

Moreover, it supports musical notions that are not note-related, like:

- measures,
- tempo indications,
- performance directions,
- repeat structures,
- and time/key signatures.

In addition to handling score information, the package can load MIDI recordings of performed scores, and alignments between scores and performances.

1.1 Supported file types

Musical data can be loaded from and saved to *MusicXML* and *MIDI* files. Furthermore, *partitura* uses MuseScore as a backend to load files in other formats, like *MuseScore*, *MuseData*, and *GuitarPro*. This requires a working installation of MuseScore on your computer. *MEI* format is currently not supported, but support is planned for a future release.

Score-performance alignments can be read from different file types by *partitura*. Firstly it supports reading from the *Matchfile* format used by the publicly available Vienna4x22 piano corpus research dataset. Secondly there is read support for *Match* and *Corresp* files produced by Nakamura's music alignment software.

1.2 Conceptual Overview

This section offers some conceptual and design considerations that may be helpful when working with the package.

1.2.1 Representing score information

The package defines a musical ontology to describe musical scores that roughly follows the elements defined by the MusicXML specification. More specifically, the elements of a musical score are represented as a collection of instances of classes like *Note*, *Measure*, *Slur*, and *Rest*. These instances are attached to an instance of class *Part*, which corresponds to the role of an instrument in a musical score. A part may contain one or more staffs, depending on the instrument.

In contrast to MusicXML documents, where musical time is largely implicit, time plays a crucial role in the representation of scores in *partitura*. Musical elements are associated to a *Part* instance by specifying their *start* (and possibly *end*) times. The *Part* instance thus acts as a timeline consisting of a number of discrete timepoints, each of which holds references to the musical elements starting and ending at that time. The musical elements themselves contain references to their respective starting and ending timepoints. Other than that, cross-references between musical elements are used sparingly, to keep the API simple.

Musical elements in a *Part* can be filtered by class and iterated over, either from a particular timepoint onward or backward, or within a specified range. For example to find the measure to which a note belongs, you would iterate backwards over elements of class Measure that start at or before the start time of the note and select the first element of that iteration.

1.2.2 Score vs. performance

Although the MIDI format can be used to represent both score-related (key/time signatures, tempo) and performance-related information (expressive timing, dynamics), partitura regards a MIDI file as a representation of either a a score or a performance. Therefore is has separate functions to load and save scores (`load_score_midi()`, `save_score_midi()`) and performances (`load_performance_midi()`, `save_performance_midi()`). `load_score_midi()` offers simple quantization for unquantized MIDIs but in general you should not expect a MIDI representation of a performance to be loaded correctly as a *Part* instance.

1.3 Relation to music21

The *music21* package has been around since 2008, and is one of the few python packages available for working with symbolic musical data. It is both more mature and more elaborate than *partitura*. The aims of *partitura* are different from and more modest than those of *music21*, which aims to provide a toolkit for computer-aided musicology. Instead, *partitura* intends to provide a convenient way to work with symbolic musical data in the context of problems such as musical expression modeling, or music generation. Although it is not the main aim of the package to provide music analysis tools, the package does offer functionality for pitch spelling, voice assignment and key estimation.

CHAPTER 2

Usage

In this Section we demonstrate basic usage of the package.

2.1 Quick start: Reading note information from a MIDI file

Before we present more in-depth usage of the package, we cover the common use case of reading note information from a MIDI file. The function `midi_to_notearray()` does exactly that: It loads the note information from the MIDI file MIDI into a structured numpy array with attributes onset (in seconds), duration (in seconds), pitch, velocity, and ID (automatically generated). For the purpose of this example we use a small MIDI file that comes with the *partitura* package. The path to the example MIDI file is stored as *partitura.EXAMPLE_MIDI*.

```
>>> import partitura
>>> path_to_midifile = partitura.EXAMPLE_MIDI
>>> note_array = partitura.midi_to_notearray(path_to_midifile)
>>> note_array # doctest: +NORMALIZE_WHITESPACE
array([(0., 2., 69, 64, 0, 1, 'n0'),
       (1., 1., 72, 64, 0, 2, 'n1'),
       (1., 1., 76, 64, 0, 2, 'n2')],
      dtype=[('onset_sec', '<f4'),
             ('duration_sec', '<f4'),
             ('pitch', '<i4'),
             ('velocity', '<i4'),
             ('track', '<i4'),
             ('channel', '<i4'),
             ('id', '<U256')])
```

The individual fields can be accessed using the field names as strings, e.g.:

```
>>> note_array["onset_sec"] # doctest: +NORMALIZE_WHITESPACE
array([0., 1., 1.], dtype=float32)
```

To access further information from MIDI files, such as time/key signatures, and control changes, see *Importing MIDI files*.

2.2 Importing MusicXML

As an example we take a MusicXML file with the following contents:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE score-partwise PUBLIC
  "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P1">
      <part-name>Piano</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <!--======================================================-->
    <measure number="1">
      <attributes>
        <divisions>12</divisions>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
      </attributes>
      <print new-page="yes" new-system="yes"/>
      <note id="n01">
        <pitch>
          <step>A</step>
          <octave>4</octave>
        </pitch>
        <duration>48</duration>
        <voice>1</voice>
        <type>whole</type>
        <staff>2</staff>
      </note>
      <backup>
        <duration>48</duration>
      </backup>
      <note id="r01">
        <rest/>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
      <note id="n02">
        <pitch>
          <step>C</step>
          <octave>5</octave>
        </pitch>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
      <note id="n03">
        <chord/>
```

(continues on next page)

(continued from previous page)

```
        <pitch>
          <step>E</step>
          <octave>5</octave>
        </pitch>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
    </measure>
  </part>
</score-partwise>
```

To load the score in python we first import the partitura package:

```
>>> import partitura
```

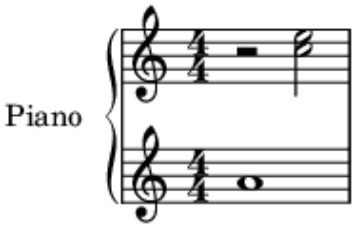
For convenience a MusicXML file with the above contents is included in the package. The path to the file is stored as *partitura.EXAMPLE_MUSICXML*, so that we load the above score as follows:

```
>>> path_to_musicxml = partitura.EXAMPLE_MUSICXML
>>> part = partitura.load_musicxml(path_to_musicxml)
```

2.3 Displaying the typeset part

The *partitura.render()* function displays the part as a typeset score:

```
>>> partitura.render(part)
```



This should open an image of the score in the default image viewing application of your desktop. The function requires that either MuseScore or lilypond is installed on your computer.

2.4 Exporting a score to MusicXML

The *partitura.save_musicxml()* function exports score information to MusicXML. The following line saves *part* to a file *mypart.musicxml*:

```
>>> partitura.save_musicxml(part, 'mypart.musicxml')
```

2.5 Viewing the contents of a score

The function *load_musicxml()* returns the score as a *Part* instance. When we print it, it displays its id and part-name:

```
>>> print(part)
Part id="P1" name="Piano"
```

To see all of the elements in the part at once, we can call its *pretty()* method:

```
>>> print(part.pretty())
Part id="P1" name="Piano"
  │
  ├─ TimePoint t=0 quarter=12
  │    │
  │    └─ starting objects
  │         │
  │         ├─ 0--48 Measure number=1
  │         ├─ 0--48 Note id=n01 voice=1 staff=2 type=whole pitch=A4
  │         ├─ 0--48 Page number=1
  │         ├─ 0--24 Rest id=r01 voice=2 staff=1 type=half
  │         ├─ 0--48 System number=1
  │         └─ 0-- TimeSignature 4/4
  │
  ├─ TimePoint t=24 quarter=12
  │    │
  │    ├─ ending objects
  │    │    │
  │    │    └─ 0--24 Rest id=r01 voice=2 staff=1 type=half
  │    │
  │    └─ starting objects
  │         │
  │         ├─ 24--48 Note id=n02 voice=2 staff=1 type=half pitch=C5
  │         └─ 24--48 Note id=n03 voice=2 staff=1 type=half pitch=E5
  │
  └─ TimePoint t=48 quarter=12
       │
       └─ ending objects
            │
            ├─ 0--48 Measure number=1
            ├─ 0--48 Note id=n01 voice=1 staff=2 type=whole pitch=A4
            ├─ 24--48 Note id=n02 voice=2 staff=1 type=half pitch=C5
            ├─ 24--48 Note id=n03 voice=2 staff=1 type=half pitch=E5
            ├─ 0--48 Page number=1
            └─ 0--48 System number=1
```

This reveals that the part has three time points at which one or more musical objects start or end. At *t=0* there are several starting objects, including a *TimeSignature*, *Measure*, *Page*, and *System*.

2.6 Extracting note information from a Part

The notes in this part can be accessed through the *notes* property:

```
>>> part.notes
[<partitura.score.Note object at 0x...>,
```

(continues on next page)

(continued from previous page)

```
 <partitura.score.Note object at 0x...>,
 <partitura.score.Note object at 0x...>]
>>> part.notes[0].duration  # duration in divs
48
```

Alternatively, basic note attributes can be accessed through the `note_array` property:

```
>>> arr = part.note_array
>>> arr.dtype
dtype([('onset_beat', '<f4'),
       ('duration_beat', '<f4'),
       ('onset_quarter', '<f4'),
               ('duration_quarter', '<f4'),
       ('onset_div', '<i4'),
               ('duration_div', '<i4'),
       ('pitch', '<i4'),
               ('voice', '<i4'),
               ('id', '<U256')])
```

The onsets and durations of the notes are specified in various units of time.

```
>>> for pitch, onset, duration in arr[["pitch", "onset_beat", "duration_beat"]]:
...     print(pitch, onset, duration)
69 0.0 4.0
72 2.0 2.0
76 2.0 2.0
```

2.7 Iterating over arbitrary musical objects

In the previous Section we used *part.notes* to obtain the notes in the part as a list. This property is a shortcut for the following statement:

```
>>> list(part.iter_all(partitura.score.Note))
[<partitura.score.Note object at 0x...>,
 <partitura.score.Note object at 0x...>,
 <partitura.score.Note object at 0x...>]
```

That is, we iterate over all objects of class *partitura.score.Note*, and store them in a list. The *iter_all()* method can be used to iterate over objects of arbitrary classes in the part:

```
>>> for m in part.iter_all(partitura.score.Measure):
...     print(m)
0--48 Measure number=1
```

The *iter_all()* method has a keyword *include_subclasses* that indicates that we are also interested in any subclasses of the specified class. For example, the following statement iterates over all objects in the part:

```
>>> for m in part.iter_all(object, include_subclasses=True):
...     print(m)
0--48 Note id=n01 voice=1 staff=2 type=whole pitch=A4
0--24 Rest id=r01 voice=2 staff=1 type=half
0--48 Page number=1
0--48 System number=1
0--48 Measure number=1
```

(continues on next page)

(continued from previous page)

```
0-- TimeSignature 4/4
24--48 Note id=n02 voice=2 staff=1 type=half pitch=C5
24--48 Note id=n03 voice=2 staff=1 type=half pitch=E5
```

This approach is useful for example when we want to retrieve rests in addition to notes. Since rests and notes are both subclassess of *GenericNote*, the following works:

```
>>> for m in part.iter_all(partitura.score.GenericNote, include_subclasses=True):
...     print(m)
0--48 Note id=n01 voice=1 staff=2 type=whole pitch=A4
0--24 Rest id=r01 voice=2 staff=1 type=half
24--48 Note id=n02 voice=2 staff=1 type=half pitch=C5
24--48 Note id=n03 voice=2 staff=1 type=half pitch=E5
```

By default, *include_subclasses* is False.

2.8 Creating a musical score by hand

You can build a musical score from scratch, by creating a *partitura.score.Part* object. We start by renaming the *partitura.score* module to *score*, for convenience:

```
>>> import partitura.score as score
```

Then we create an empty part with id 'P0' and name 'My Part' (the name is optional, the id is mandatory), and a quarter note duration of 10 units.

```
>>> part = score.Part('P0', 'My Part', quarter_duration=10)
```

Adding elements to the part is done by the *add()* method, which takes a musical element, a start and an end time. Either of the *start* and *end* arguments can be omitted, but if both are omitted the method will do nothing.

We now add a 3/4 time signature at t=0, and three notes. The notes are instantiated by specifying an (optional) id, pitch information, and an (optional) voice:

```
>>> part.add(score.TimeSignature(3, 4), start=0)
>>> part.add(score.Note(id='n0', step='A', octave=4, voice=1), start=0, end=10)
>>> part.add(score.Note(id='n1', step='C', octave=5, alter=1, voice=2), start=0,␣
↪end=10)
>>> part.add(score.Note(id='n2', step='C', octave=5, alter=1, voice=2), start=10,␣
↪end=40)
```

Note that the duration of notes is not hard-coded in the Note instances, but defined implicitly by their start and end times in the part.

Here's what the part looks like:

```
>>> print(part.pretty())
Part id="P0" name="My Part"
  │
  ├─ TimePoint t=0 quarter=10
  │    │
  │    └─ starting objects
  │         │
  │         ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
```

(continues on next page)

(continued from previous page)

```
  │       ├─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │       └─ 0-- TimeSignature 3/4
  │
  ├─ TimePoint t=10 quarter=10
  │   │
  │   ├─ ending objects
  │   │   │
  │   │   ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
  │   │   └─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │   │
  │   └─ starting objects
  │       │
  │       └─ 10--40 Note id=n2 voice=2 staff=None type=half. pitch=C#5
  │
  └─ TimePoint t=40 quarter=10
      │
      └─ ending objects
          │
          └─ 10--40 Note id=n2 voice=2 staff=None type=half. pitch=C#5
```

We see that the notes n0, n1, and n2 have been correctly recognized as quarter, quarter, and dotted half, respectively.

Let's save the part to MusicXML:

```
>>> partitura.save_musicxml(part, 'mypart.musicxml')
```

When we look at the contents of *mypart.musicxml*, surprisingly, the *<part></part>* element is empty:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE score-partwise PUBLIC
  "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P0">
      <part-name>My Part</part-name>
    </score-part>
  </part-list>
  <part id="P0"/>
</score-partwise>
```

The problem with our newly created part is that it contains no measures. Since the MusicXML format requires musical elements to be contained in measures, saving the part to MusicXML omits the objects we added.

2.9 Adding measures

One option to add measures is to add them by hand like we've added the notes and time signature. A more convenient alternative is to use the function *add_measures()*:

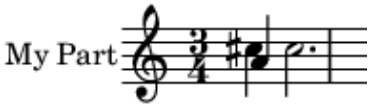
```
>>> score.add_measures(part)
```

This function uses the time signature information in the part to add measures accordingly:

```
>>> print(part.pretty())
Part id="P0" name="My Part"
  ├─ TimePoint t=0 quarter=10
  │    └─ starting objects
  │         ├─ 0--30 Measure number=1
  │         ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
  │         ├─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │         └─ 0-- TimeSignature 3/4
  ├─ TimePoint t=10 quarter=10
  │    ├─ ending objects
  │    │    ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
  │    │    └─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │    └─ starting objects
  │         └─ 10--40 Note id=n2 voice=2 staff=None type=half. pitch=C#5
  ├─ TimePoint t=30 quarter=10
  │    ├─ ending objects
  │    │    └─ 0--30 Measure number=1
  │    └─ starting objects
  │         └─ 30--40 Measure number=2
  └─ TimePoint t=40 quarter=10
       └─ ending objects
            ├─ 30--40 Measure number=2
            └─ 10--40 Note id=n2 voice=2 staff=None type=half. pitch=C#5
```

Let's see what our part with measures looks like in typeset form:

```
>>> partitura.render(part)
```

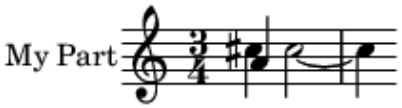


Although the notes are there, the music is not typeset correctly, since the first measure should have a duration of three quarter notes, but instead is has a duration of four quarter notes. The problem is that the note *n2* crosses a measure boundary, and thus should be tied.

2.10 Splitting up notes using ties

In musical notation notes that span measure boundaries are split up, and then tied together. This can be done automatically using the function *tie_notes()*:

```
>>> score.tie_notes(part)
>>> partitura.render(part)
```



Now the score looks correct. Displaying the contents reveals that the part now has an extra quarter note *n2a* that starts at the measure boundary, whereas the note *n2* is now a half note, ending at the measure boundary.

```
>>> print(part.pretty())
Part id="P0" name="My Part"
  ├─ TimePoint t=0 quarter=10
  │    └─ starting objects
  │         ├─ 0--30 Measure number=1
  │         ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
  │         ├─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │         └─ 0-- TimeSignature 3/4
  ├─ TimePoint t=10 quarter=10
  │    ├─ ending objects
  │    │    ├─ 0--10 Note id=n0 voice=1 staff=None type=quarter pitch=A4
  │    │    └─ 0--10 Note id=n1 voice=2 staff=None type=quarter pitch=C#5
  │    └─ starting objects
  │         └─ 10--30 Note id=n2 voice=2 staff=None type=half tie_group=n2+n2a pitch=C#5
  ├─ TimePoint t=30 quarter=10
  │    ├─ ending objects
  │    │    ├─ 0--30 Measure number=1
  │    │    └─ 10--30 Note id=n2 voice=2 staff=None type=half tie_group=n2+n2a pitch=C#5
  │    └─ starting objects
  │         ├─ 30--40 Measure number=2
  │         └─ 30--40 Note id=n2a voice=2 staff=None type=quarter tie_group=n2+n2a␣
↪pitch=C#5
  └─ TimePoint t=40 quarter=10
       └─ ending objects
            ├─ 30--40 Measure number=2
            └─ 30--40 Note id=n2a voice=2 staff=None type=quarter tie_group=n2+n2a␣
↪pitch=C#5
```

2.11 Removing elements

Just like we can add elements to a part, we can also remove them, using the *remove()* method. The following lines remove the measure instances that were added using the *add_measures()* function:

```
>>> for measure in list(part.iter_all(score.Measure)):
...     part.remove(measure)
```

Note that we create a list of all measures in *part* before we remove them. This is necessary to avoid changing the contents of *part* while we iterate over it.

2.12 Importing MIDI files

For quick access to note information from a MIDI file, use the function `midi_to_notearray()`, as described in *Quick start: Reading note information from a MIDI file*. In addition to this function, which returns a structured numpy array, partitura provides two further functions to load information from MIDI files, depending on whether the information should be treated as a performance or as a score (see introduction.html#score-vs-performance):

- *load_performance_midi()*
- *load_score_midi()*

The *load_performance_midi()* returns a *PerformedPart* instance. The *PerformedPart* instance stores notes, program change and control change messages. The notes in *notes* are dictionaries with the usual MIDI attributes "midi_pitch", "note_on", "note_off", etc. Additionally, there is a key called "sound_off" which returns note_off times adjusted by the sustain pedal. Set the on/off threshold value for the sustain_pedal MIDI cc message like so:

```
>>> path_to_midifile = partitura.EXAMPLE_MIDI
>>> performedpart = partitura.load_performance_midi(path_to_midifile)
>>> performedpart.sustain_pedal_threshold=64
```

Setting the sustain pedal threshold to 128 will prevent the change of "sound_off" values by sustain pedal. When the MIDI file does not contain any pedal information, the "sound_off" is equal to "note_off", and setting *sustain_pedal_threshold* has no effect. Calling *note_array* will return a structured array like `midi_to_notearray()`. The values in *note_array["duration_sec"]* are the actual duration of the note based on the *sound_off* time.

The function *load_score_midi()* returns a *Part* instance. The function estimates the score structure based on the "parts per quarter" value and the note_on/note_off times in a MIDI file. This function *only* works with deadpan "score" MIDI files that can be generated by Digital Audio Workstations, Scorewriters, and other sequencers. It is not suitable to estimate the score from a performed MIDI file, such as a recording of a pianist playing on a MIDI keyboard.

```
>>> midipart = partitura.load_score_midi(path_to_midifile)
>>> midipart.note_array  # doctest: +NORMALIZE_WHITESPACE
    array([(0., 4., 0., 4.,  0, 48, 69, 1, 'n0'),
           (2., 2., 2., 2., 24, 24, 72, 2, 'n1'),
           (2., 2., 2., 2., 24, 24, 76, 2, 'n2')],
          dtype=[('onset_beat', '<f4'),
                 ('duration_beat', '<f4'),
                 ('onset_quarter', '<f4'),
                 ('duration_quarter', '<f4'),
                 ('onset_div', '<i4'),
                 ('duration_div', '<i4'),
                 ('pitch', '<i4'),
```

(continues on next page)

(continued from previous page)

```
                ('voice', '<i4'),
                ('id', '<U256')])
```

The note_array of a part is a structured array similar to the one of the *PerformedPart* instance, but the first 6 fields refer to onset and duration in score time. The score MIDI function correctly identifies the note lengths of a whole note and two half notes. However, the position of the first measure bar (as well as other score properties) is only an estimate as a "score" MIDI file of a score that begins with a tied quarter note in an anacrusis measure would look exactly the same in the MIDI encoding.

2.13 Music Analysis

The package offers tools for various types music analysis, including key estimation, tonal tension estimation, voice separation, and pitch spelling. The functions take the note information of in the form of an instance of *Part*, *PartGroup*, or *PerformedPart*, a list of *Part* objects or a structured numpy array, as returned by the `note_array` attribute.

2.13.1 Key Estimation

Key estimation is performed by the function *estimate_key()*. The function returns a string representation of the root and mode of the key:

```
>>> key_name = partitura.musicanalysis.estimate_key(part.note_array)
>>> print(key_name)
C#m
```

The number of sharps/flats and the mode can be inferred from the key name using the convenience function *key_name_to_fifths_mode()*:

```
>>> partitura.utils.key_name_to_fifths_mode(key_name)
(4, 'minor')
```

2.13.2 Pitch Spelling

Pitch spelling estimation is performed by the function *estimate_spelling()*. The function returns a structured array with pitch spelling information (i.e., with fields *step*, *alter* and *octave*) for each note in the input *note_array*. If the input to this method is an instance of *Part*, *PartGroup*, or *PerformedPart*, a list of *Part*, each row of the output corresponds to order of the notes in the *note_array* that would be generated by using the helper method `ensure_notearray()`.

```
>>> pitch_spelling = partitura.musicanalysis.estimate_spelling(part.note_array)
>>> print(pitch_spelling)
[('A', 0, 4) ('C', 1, 5) ('C', 1, 5)]
```

2.13.3 Voice Estimation

Voice estimation is performed by the function *estimate_voices()*. The function returns a numpy array with voice information for each note in the input *note_array*. If the input to this method is an instance of *Part*, *PartGroup*, or *PerformedPart*, a list of *Part*, each row of the output corresponds to order of the notes in the *note_array* that would be generated by using the helper method `ensure_notearray()`.

```
>>> voices = partitura.musicanalysis.estimate_voices(part.note_array)
>>> print(voices)
[1 1 1]
```

2.13.4 Tonal Tension

Three tonal tension features proposed by Herremans and Chew (2016) are estimated by the function *estimate_tonaltension()*. The function returns a strured array with fields *cloud_diameter*, *cloud_momentum*, *tensile_strain* and *onset*. In contrast to the other methods in *partitura.musicanalysis*, the tonal tension features are not computed for each note, but for specific time points, which are specified by argument *ss*, which can be a float specifying the step size, a 1D numpy array with time values, or *'onset'*, which computes the tension features at each unique onset time.

```
>>> import numpy as np
>>> tonal_tension = partitura.musicanalysis.estimate_tonaltension(part, ss='onset')
>>> print(np.unique(part.note_array['onset_beat']))
[0. 1.]
>>> print(tonal_tension.dtype.names)
('onset_beat', 'cloud_diameter', 'cloud_momentum', 'tensile_strain')
>>> print(tonal_tension['cloud_momentum'])
[0.         0.16666667]
```

```
>>> partitura.musicanalysis.estimate_spelling(part.note_array)  # doctest: +NORMALIZE_
↪WHITESPACE
array([('A', 0, 4), ('C', 1, 5), ('C', 1, 5)],
      dtype=[('step', '<U1'), ('alter', '<i8'), ('octave', '<i8')])
```

CHAPTER 3

Index

CHAPTER 4

partitura

The top level of the package contains functions to load and save data, display rendered scores, and functions to estimate pitch spelling, voice assignment, and key signature.

partitura.**EXAMPLE_MUSICXML = '/home/docs/.cache/Python-Eggs/partitura-0.4.0-py3.7.egg-tmp/p**
An example MusicXML file for didactic purposes

partitura.**EXAMPLE_MIDI = '/home/docs/.cache/Python-Eggs/partitura-0.4.0-py3.7.egg-tmp/parti**
An example MIDI file for didactic purposes

partitura.**load_musicxml**(*xml*, *ensure_list=False*, *validate=False*, *force_note_ids=None*)
Parse a MusicXML file and build a composite score ontology structure from it (see also scoreontology.py).

Parameters

- **xml** (*str or file-like object*) – Path to the MusicXML file to be parsed, or a file-like object
- **ensure_list** (*bool, optional*) – When True return a list independent of how many part or partgroup elements were created from the MIDI file. By default, when the return value of *load_musicxml* produces a
- **single** (class:*partitura.score.Part* or) – *partitura.score.PartGroup* element, the element itself is returned instead of a list containing the element. Defaults to False.
- **validate** (*bool, optional*) – When True the validity of the MusicXML is checked against the MusicXML 3.1 specification before loading the file. An exception will be raised when the MusicXML is invalid. Defaults to False.
- **force_note_ids** (*(bool, 'keep') optional.*) – When True each Note in the returned Part(s) will have a newly assigned unique id attribute. Existing note id attributes in the MusicXML will be discarded. If 'keep', only notes without a note id will be assigned one.

Returns **partlist** – A list of either Part or PartGroup objects

Return type list

partitura.**save_musicxml**(*parts*, *out=None*)

Save a one or more Part or PartGroup instances in MusicXML format.

Parameters

- **parts** (*list, Part, or PartGroup*) – A *partitura.score.Part* object, *partitura.score.PartGroup* or a list of these
- **out** (*str, file-like object, or None, optional*) – Output file

Returns If no output file is specified using *out* the function returns the MusicXML data as a string. Otherwise the function returns None.

Return type None or str

partitura.**musicxml_to_notearray**(*fn*, *flatten_parts=True*, *include_pitch_spelling=False*, *include_key_signature=False*, *include_time_signature=False*)

Return pitch, onset, and duration information for notes from a MusicXML file as a structured array.

By default a single array is returned by combining the note information of all parts in the MusicXML file.

Parameters

- **fn** (*str*) – Path to a MusicXML file
- **flatten_parts** (*bool*) – If *True*, returns a single array containing all notes. Otherwise, returns a list of arrays for each part.
- **include_pitch_spelling** (*bool (optional)*) – If *True*, includes pitch spelling information for each note. Default is False
- **include_key_signature** (*bool (optional)*) – If *True*, includes key signature information, i.e., the key signature at the onset time of each note (all notes starting at the same time have the same key signature). Default is False
- **include_time_signature** (*bool (optional)*) – If *True*, includes time signature information, i.e., the time signature at the onset time of each note (all notes starting at the same time have the same time signature). Default is False

Returns **score** – Structured array or list of structured arrays containing score information.

Return type structured array or list of structured arrays

partitura.**load_score_midi**(*fn*, *part_voice_assign_mode=0*, *ensure_list=False*, *quantization_unit=None*, *estimate_voice_info=True*, *estimate_key=False*, *assign_note_ids=True*)

Load a musical score from a MIDI file and return it as a Part instance.

This function interprets MIDI information as describing a score. Pitch names are estimated using Meredith's PS13 algorithm[1]. Assignment of notes to voices can either be done using Chew and Wu's voice separation algorithm[2], or by choosing one of the part/voice assignment modes that assign voices based on track/channel information. Furthermore, the key signature can be estimated based on Krumhansl's 1990 key profiles[3].

This function expects times to be metrical/quantized. Optionally a quantization unit may be specified. If you wish to access the non- quantized time of MIDI events you may wish to used the *load_performance_midi* function instead.

Parameters

- **fn** (*str*) – Path to MIDI file

[1] Meredith, D. (2006). "The ps13 Pitch Spelling Algorithm". Journal of New Music Research, 35(2):121.

[2] Chew, E. and Wu, Xiaodan (2004) "Separating Voices in Polyphonic Music: A Contig Mapping Approach". In Uffe Kock, editor, Computer Music Modeling and Retrieval (CMMR), pp. 1–20, Springer Berlin Heidelberg.

[3] Krumhansl, Carol L. (1990) "Cognitive foundations of musical pitch", Oxford University Press, New York.

- **part_voice_assign_mode** (*{0, 1, 2, 3, 4, 5}, optional*) – This keyword controls how part and voice information is associated to track and channel information in the MIDI file. The semantics of the modes is as follows:

  0 Return one Part per track, with voices assigned by channel

  1 Return one PartGroup per track, with Parts assigned by channel (no voices)

  2 Return single Part with voices assigned by track (tracks are combined, channel info is ignored)

  3 Return one Part per track, without voices (channel info is ignored)

  4 Return single Part without voices (channel and track info is ignored)

  5 Return one Part per <track, channel> combination, without voices Defaults to 0.

- **ensure_list** (*bool, optional*) – When True, return a list independent of how many part or partgroup elements were created from the MIDI file. By default, when the return value of *load_score_midi* produces a single *partitura.score.Part* or *partitura.score.PartGroup* element, the element itself is returned instead of a list containing the element. Defaults to False.

- **quantization_unit** (*integer or None, optional*) – Quantize MIDI times to multiples of this unit. If None, the quantization unit is chosen automatically as the smallest division of the parts per quarter (MIDI "ticks") that can be represented as a symbolic duration. Defaults to None.

- **estimate_key** (*bool, optional*) – When True use Krumhansl's 1990 key profiles[3] to determine the most likely global key, discarding any key information in the MIDI file.

- **estimate_voice_info** (*bool, optional*) – When True use Chew and Wu's voice separation algorithm[2] to estimate voice information. This option is ignored for part/voice assignment modes that infer voice information from the track/channel info (i.e. *part_voice_assign_mode* equals 1, 3, 4, or 5). Defaults to True.

Returns One or more part or partgroup objects

Return type *partitura.score.Part*, *partitura.score.PartGroup*, or a list of these

References

partitura.**save_score_midi**(*parts*, *out*, *part_voice_assign_mode=0*, *velocity=64*, *anacrusis_behavior='shift'*)

Write data from Part objects to a MIDI file

Parameters

- **parts** (*Part, PartGroup or list of these*) – The musical score to be saved.

- **out** (*str or file-like object*) – Either a filename or a file-like object to write the MIDI data to.

- **part_voice_assign_mode** (*{0, 1, 2, 3, 4, 5}, optional*) – This keyword controls how part and voice information is associated to track and channel information in the MIDI file. The semantics of the modes is as follows:

  0 Write one track for each Part, with channels assigned by voices

  1 Write one track for each PartGroup, with channels assigned by Parts (voice info is lost) (There can be multiple levels of partgroups, I suggest using the highest level of part-

group/part) [note: this will e.g. lead to all strings into the same track] Each part not in a PartGroup will be assigned its own track

2 Write a single track with channels assigned by Part (voice info is lost)

3 Write one track per Part, and a single channel for all voices (voice info is lost)

4 Write a single track with a single channel (Part and voice info is lost)

5 Return one track per <Part, voice> combination, each track having a single channel.

The default mode is 0.

- **velocity** (*int, optional*) – Default velocity for all MIDI notes. Defaults to 64.
- **anacrusis_behavior** (*{"shift", "pad_bar"}, optional*) – Strategy to deal with anacrusis. If "shift", all time points are shifted by the anacrusis (i.e., the first note starts at 0). If "pad_bar", the "incomplete" bar of the anacrusis is padded with silence. Defaults to 'shift'.

partitura.**load_via_musescore**(*fn*, *ensure_list=False*, *validate=False*, *force_note_ids=True*)

Load a score through through the MuseScore program.

This function attempts to load the file in MuseScore, export it as MusicXML, and then load the MusicXML. This should enable loading of all file formats that for which MuseScore has import-support (e.g. MIDI, and ABC, but currently not MEI).

Parameters

- **fn** (*str*) – Filename of the score to load
- **ensure_list** (*bool, optional*) – When True return a list independent of how many part or partgroup elements were created from the MIDI file. By default, when the return value of *load_musicxml* produces a
- **single** (class:*partitura.score.Part* or) – *partitura.score.PartGroup* element, the element itself is returned instead of a list containing the element. Defaults to False.
- **validate** (*bool, optional*) – When True the validity of the MusicXML generated by MuseScore is checked against the MusicXML 3.1 specification before loading the file. An exception will be raised when the MusicXML is invalid. Defaults to False.
- **force_note_ids** (*bool, optional.*) – When True each Note in the returned Part(s) will have a newly assigned unique id attribute. Existing note id attributes in the MusicXML will be discarded.

Returns One or more part or partgroup objects

Return type *partitura.score.Part*, *partitura.score.PartGroup*, or a list of these

partitura.**load_performance_midi**(*fn*, *default_bpm=120*, *merge_tracks=False*)

Load a musical performance from a MIDI file.

This function should be used for MIDI files that encode performances, such as those obtained from a capture of a MIDI instrument. This function loads note on/off events as well as control events, but ignores other data such as time and key signatures. Furthermore, the PerformedPart instance that the function returns does not retain the ticks_per_beat or tempo events. The timing of all events is represented in seconds. If you wish to retain this information consider using the *load_score_midi* function.

Parameters

- **fn** (*str*) – Path to MIDI file
- **default_bpm** (*number, optional*) – Tempo to use wherever the MIDI does not specify a tempo. Defaults to 120.

Returns A PerformedPart instance.

Return type *partitura.performance.PerformedPart*

partitura.**save_performance_midi**(*performed_part*, *out*, *mpq=500000*, *ppq=480*, *default_velocity=64*)

Save a *PerformedPart* instance as a MIDI file.

Parameters

- **performed_part** (*PerformedPart*) – The performed part to save
- **out** (*str or file-like object*) – Either a filename or a file-like object to write the MIDI data to.
- **mpq** (*int, optional*) – Microseconds per quarter note. This is known in MIDI parlance as the "tempo" value. Defaults to 500000 (i.e. 120 BPM).
- **ppq** (*int, optional*) – Parts per quarter, also known as ticks per beat. Defaults to 480.
- **default_velocity** (*int, optional*) – A default velocity value (between 0 and 127) to be used for notes without a specified velocity. Defaults to 64.

partitura.**load_match**(*fn*, *create_part=False*, *pedal_threshold=64*, *first_note_at_zero=False*, *offset_duration_whole=True*)

Load a matchfile.

Parameters

- **fn** (*str*) – The matchfile
- **create_part** (*bool, optional*) – When True create a Part object from the snote information in the match file. Defaults to False.
- **pedal_threshold** (*int, optional*) – Threshold for adjusting sound off of the performed notes using pedal information. Defaults to 64.
- **first_note_at_zero** (*bool, optional*) – When True the note_on and note_off times in the performance are shifted to make the first note_on time equal zero.

Returns

- **ppart** (*list*) – The performed part, a list of dictionaries
- **alignment** (*list*) – The score–performance alignment, a list of dictionaries
- **spart** (*Part*) – The score part. This item is only returned when *create_part* = True.

partitura.**save_match**(*alignment*, *ppart*, *spart*, *out*, *mpq=500000*, *ppq=480*, *performer=None*, *composer=None*, *piece=None*)

Save an Alignment of a PerformedPart to a Part in a match file.

Parameters

- **alignment** (*list*) – A list of dictionaries containing alignment information. See *partitura.io.importmatch.alignment_from_matchfile*.
- **ppart** (partitura.performance.PerformedPart) – An instance of *PerformedPart* containing performance information.
- **spart** (partitura.score.Part) – An instance of *Part* containing score information.
- **out** (*str*) – Out to export the matchfile.
- **mpq** (*int*) – Milliseconds per quarter note.
- **ppq** (*int*) – Parts per quarter note.

- **performer** (*str or None*) – Name(s) of the performer(s) of the *PerformedPart*.
- **composer** (*str or None*) – Name(s) of the composer(s) of the piece represented by *Part*.
- **piece** (*str or None:*) – Name of the piece represented by *Part*.

partitura.**load_nakamuramatch**(*fn*)

Load a match file as returned by Nakamura et al.'s MIDI to musicxml alignment

Fields of the file format as specified in[8]: ID (onset time) (offset time) (spelled pitch) (onset velocity)(offset velocity) channel (match status) (score time) (note ID)(error index) (skip index)

Parameters **fn** (*str*) – The nakamura match.txt-file

Returns

- **align** (*structured array*) – structured array of performed notes
- **ref** (*structured array*) – structured array of score notes
- **alignment** (*list*) – The score–performance alignment, a list of dictionaries

References

partitura.**load_nakamuracorresp**(*fn*)

Load a corresp file as returned by Nakamura et al.'s MIDI to MIDI alignment.

Fields of the file format as specified in[8]: (ID) (onset time) (spelled pitch) (integer pitch) (onset velocity)

Parameters **fn** (*str*) – The nakamura match.txt-file

Returns

- **align** (*structured array*) – structured array of performed notes
- **ref** (*structured array*) – structured array of score notes
- **alignment** (*list*) – The score–performance alignment, a list of dictionaries

partitura.**render**(*part*, *fmt='png'*, *dpi=90*, *out_fn=None*)

Create a rendering of one or more parts or partgroups.

The function can save the rendered image to a file (when *out_fn* is specified), or shown in the default image viewer application.

Rendering is first attempted through musecore, and if that fails through lilypond. If that also fails the function returns without raising an exception.

Parameters

- **part** (*partitura.score.Part* or *partitura.score.PartGroup*) – or a list of these The score content to be displayed
- **fmt** (*{'png', 'pdf'}, optional*) – The image format of the rendered material
- **out_fn** (*str or None, optional*) – The path of the image output file. If None, the rendering will be displayed in a viewer.

[8] https://midialignment.github.io/MANUAL.pdf

CHAPTER 5

partitura.score

This module defines an ontology of musical elements to represent musical scores, such as measures, notes, slurs, words, tempo and loudness directions. A score is defined at the highest level by a *Part* object (or a hierarchy of *Part* objects, in a *PartGroup* object). This object serves as a timeline at which musical elements are registered in terms of their start and end times.

class partitura.score.**Part**(*id*, *part_name=None*, *part_abbreviation=None*, *quarter_duration=1*)

Bases: object

Represents a score part, e.g. all notes of one single instrument (or multiple instruments written in the same staff). Note that there may be more than one staff per score part.

Parameters

- **id** (*str*) – The identifier of the part. In order to be compatible with MusicXML the identifier should not start with a number.
- **part_name** (*str or None, optional*) – Name for the part. Defaults to None
- **part_abbreviation** (*str or None, optional*) – Abbreviated name for part
- **quarter_duration** (*int, optional*) – The default quarter duration. See *set_quarter_duration()* for details.

id

See parameters

Type str

part_name

See parameters

Type str

part_abbreviation

See parameters

Type str

pretty()
Return a pretty representation of this object.

Returns A pretty representation

Return type str

time_signature_map
A function mapping timeline times to the beats and beat_type of the time signature at that time. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

key_signature_map
A function mappting timeline times to the key and mode of the key signature at that time. The function can take scalar values or lists/arrays of values

Returns The mapping function

Return type function

beat_map
A function mapping timeline times to beat times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

inv_beat_map
A function mapping beat times to timeline times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

quarter_map
A function mapping timeline times to quarter times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

inv_quarter_map
A function mapping quarter times to timeline times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

notes
Return a list of all Note objects in the part. This list includes GraceNote objects but not Rest objects.

Returns list of Note objects

Return type list

notes_tied
Return a list of all Note objects in the part that are either not tied, or the first note of a group of tied notes. This list includes GraceNote objects but not Rest objects.

Returns List of Note objects

Return type list

quarter_durations(*start=None*, *end=None*)
Return an Nx2 array with quarter duration (second column) and their respective times (first column).

When a start and or end time is specified, the returned array will contain only the entries within those bounds.

Parameters

- **start** (*number, optional*) – Start of range
- **end** (*number, optional*) – End of range

Returns An array with quarter durations and times

Return type ndarray

quarter_duration_map
A function mapping timeline times to quarter durations in effect at those times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

set_quarter_duration(*t*, *quarter*)
Set the duration of a quarter note from timepoint *t* onwards.

Setting the quarter note duration defines how intervals between timepoints are related to musical durations. For example when two timepoints *t1* and *t2* have associated times 10 and 20 respecively, then the interval between *t1* and *t2* corresponds to a half note when the quarter duration equals 5 during that interval.

The quarter duration can vary throughout the part. When setting a quarter duration at time t, then that value takes effect until the time of the next quarter duration. If a different quarter duration was already set at time t, it wil be replaced.

Note setting the quarter duration does not change the timepoints, only the relation to musical time. For illustration: in the example above, when changing the current quarter duration from 5 to 10, a note that starts at *t1* and ends at *t2* will change from being a half note to being a quarter note.

Parameters

- **t** (*int*) – Time at which to set the quarter duration
- **quarter** (*int*) – The quarter duration

get_point(*t*)
Return the *TimePoint* object with time *t*, or None if there is no such object.

get_or_add_point(*t*)
Return the *TimePoint* object with time *t*; if there is no such object, create it, add it to the time line, and return it.

Parameters **t** (*int*) – Time value *t*

Returns a TimePoint object with time *t*

Return type *TimePoint*

add(*o*, *start=None*, *end=None*)
Add an object to the timeline.

An object can be added by start time, end time, or both, depending on which of the *start* and *end* keywords are provided. If neither is provided this method does nothing.

start and *end* should be non-negative integers.

Parameters

- **o** (*TimedObject*) – Object to be removed
- **start** (*int, optional*) – The start time of the object
- **end** (*int, optional*) – The end time of the object

remove(*o*, *which='both'*)
Remove an object from the timeline.

An object can be removed by start time, end time, or both.

Parameters

- **o** (*TimedObject*) – Object to be removed
- **which** (*{'start', 'end', 'both'}, optional*) – Whether to remove o as a starting object, an ending object, or both. Defaults to 'both'.

iter_all(*cls=None*, *start=None*, *end=None*, *include_subclasses=False*, *mode='starting'*)
Iterate (in direction of increasing time) over all instances of *cls* that either start or end (depending on *mode*) in the interval *start* to *end*. When *start* and *end* are omitted, the whole timeline is searched.

Parameters

- **cls** (*class, optional*) – The class of objects to iterate over. If omitted, iterate over all objects in the part.
- **start** (*TimePoint*, optional) – The start of the interval to search. If omitted or None, the search starts at the start of the timeline. Defaults to None.
- **end** (*TimePoint*, optional) – The end of the interval to search. If omitted or None, the search ends at the end of the timeline. Defaults to None.
- **include_subclasses** (*bool, optional*) – If True also return instances that are subclasses of *cls*. Defaults to False.
- **mode** (*{'starting', 'ending'}, optional*) – Flag indicating whether to search for starting or ending objects. Defaults to 'starting'.

Yields *object* – Instances of the specified type.

last_point
The last TimePoint on the timeline, or None if the timeline is empty.

Returns

Return type *TimePoint*

first_point
The first TimePoint on the timeline, or None if the timeline is empty.

Returns

Return type *TimePoint*

class partitura.score.**TimePoint**(*t*, *quarter=None*)
Bases: partitura.utils.generic.ComparableMixin

A TimePoint represents a temporal position within a *Part*.

TimePoints are used to keep track of the starting and ending of musical elements in the part. They are created automatically when adding musical elements to a part using its *add()* method, so there should be normally no reason to instantiate TimePoints manually.

Parameters

- **t** (*int*) – The time associated to this TimePoint. Should be a non- negative integer.
- **quarter** (*int*) – The duration of a quarter note at this TimePoint

t
See parameters

Type int

quarter
See parameters

Type int

starting_objects
A dictionary where the musical objects starting at this time are grouped by class.

Type dictionary

ending_objects
A dictionary where the musical objects ending at this time are grouped by class.

Type dictionary

prev
The preceding TimePoint (or None if there is none)

Type *TimePoint*

next
The succeding TimePoint (or None if there is none)

Type *TimePoint*

add_starting_object(*obj*)
Add object *obj* to the list of starting objects.

remove_starting_object(*obj*)
Remove object *obj* from the list of starting objects.

remove_ending_object(*obj*)
Remove object *obj* from the list of ending objects.

add_ending_object(*obj*)
Add object *obj* to the list of ending objects.

iter_starting(*cls*, *include_subclasses=False*)
Iterate over all objects of type *cls* that start at this time point.

Parameters

- **cls** (*class*) – The type of objects to iterate over
- **include_subclasses** (*bool, optional*) – When True, include all objects of all subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instance of type *cls*

iter_ending(*cls*, *include_subclasses=False*)
Iterate over all objects of type *cls* that end at this time point.

Parameters

- **cls** (*class*) – The type of objects to iterate over
- **include_subclasses** (*bool, optional*) – When True, include all objects of all subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instance of type *cls*

iter_prev(*cls*, *eq=False*, *include_subclasses=False*)
Iterate backwards in time from the current timepoint over starting object(s) of type *cls*.

Parameters

- **cls** (*class*) – Class of objects to iterate over
- **eq** (*bool, optional*) – If True start iterating at the current timepoint, rather than its predecessor. Defaults to False.
- **include_subclasses** (*bool, optional*) – If True include subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instances of *cls*

iter_next(*cls*, *eq=False*, *include_subclasses=False*)
Iterate forwards in time from the current timepoint over starting object(s) of type *cls*.

Parameters

- **cls** (*class*) – Class of objects to iterate over
- **eq** (*bool, optional*) – If True start iterating at the current timepoint, rather than its successor. Defaults to False.
- **include_subclasses** (*bool, optional*) – If True include subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instances of *cls*

class partitura.score.**TimedObject**
Bases: partitura.utils.generic.ReplaceRefMixin

This is the base class of all classes that have a start and end point. The start and end attributes initialized to None, and are set/unset when the object is added to/removed from a Part, using its *add()* and *remove()* methods, respectively.

start
Start time of the object

Type *TimePoint*

end
End time of the object

Type *TimePoint*

duration
The duration of the timed object in divisions. When either the start or the end property of the object are None, the duration is None.

Returns

Return type int or None

class partitura.score.**GenericNote**(*id=None*, *voice=None*, *staff=None*, *symbolic_duration=None*, *articulations=None*, *doc_order=None*)

Bases: *partitura.score.TimedObject*

Represents the common aspects of notes, rests, and unpitched notes.

Parameters

- **id** (*str, optional (default: None)*) – A string identifying the note. To be compatible with the MusicXML format, the id must be unique within a part and must not start with a number.
- **voice** (*int, optional*) – An integer representing the voice to which the note belongs. Defaults to None.
- **staff** (*str, optional*) – An integer representing the staff to which the note belongs. Defaults to None.
- **doc_order** (*int, optional*) – The document order index (zero-based), expressing the order of appearance of this note (with respect to other notes) in the document in case the Note belongs to a part that was imported from MusicXML. Defaults to None.

symbolic_duration

The symbolic duration of the note.

This property returns a dictionary specifying the symbolic duration of the note. The dictionary may have the following keys:

- type : the note type as a string, e.g. 'quarter', 'half'
- dots : an integer specifying the number of dots. When this key is missing it means there are no dots.
- actual_notes : Specifies the number of actual notes in a rhythmical tuplet. Used in conjunction with *normal_notes*.
- normal_notes : Specifies the normal number of notes in a rhythmical tuplet. For example a triplet of eights in the time of two eights would correspond to actual_notes=3, normal_notes=2.

The symbolic duration dictionary of a note can either be set manually (for example by specifying the *symbolic_duration* constructor keyword argument), or left unspecified (i.e. None). In the latter case the symbolic duration is estimated dynamically based on the note start and end times. Note that this latter case is generally preferrable because it ensures that the symbolic duration is consistent with the numeric duration.

If the symbolic duration cannot be estimated from the numeric duration None is returned.

Returns A dictionary specifying the symbolic duration of the note, or None if the symbolic duration could not be estimated from the numeric duration.

Return type dict or None

end_tied

The *Timepoint* corresponding to the end of the note, or— when this note belongs to a group of tied notes—the end of the last note in the group.

Returns End of note

Return type *TimePoint*

duration_tied

Time difference of the start of the note to the end of the note, or—when this note belongs to a group of tied notes— the end of the last note in the group.

Returns Duration of note

Return type int

duration_from_symbolic
Return the numeric duration given the symbolic duration of the note and the quarter_duration in effect.

Returns

Return type int or None

tie_prev_notes
TODO

Returns Description of return value

Return type type

tie_next_notes
TODO

Returns Description of return value

Return type type

iter_chord(*same_duration=True*, *same_voice=True*)
Iterate over notes with coinciding start times.

Parameters

- **same_duration** (*bool, optional*) – When True limit the iteration to notes that have the same duration as the current note. Defaults to True.
- **same_voice** (*bool, optional*) – When True limit the iteration to notes that have the same voice as the current note. Defaults to True.

Yields *GenericNote*

class partitura.score.**Note**(*step*, *octave*, *alter=None*, *beam=None*, ***kwargs*)
Bases: *partitura.score.GenericNote*

Subclass of GenericNote representing pitched notes.

Parameters

- **step** (*{'C', 'D', 'E', 'F', 'G', 'A', 'B'}*) – The note name of the pitch (in upper case). If a lower case note name is given, it will be converted to upper case.
- **octave** (*int*) – An integer representing the octave of the pitch
- **alter** (*int, optional*) – An integer (or None) representing the alteration of the pitch as follows:

  -2 double flat

  -1 flat

  0 or None unaltered

  1 sharp

  2 double sharp

  Defaults to None.

midi_pitch
The midi pitch value of the note (MIDI note number). C4 (middle C, in german: c') is note number 60.

Returns The note's pitch as MIDI note number.

Return type integer

alter_sign
The alteration of the note

Returns

Return type str

class partitura.score.**Rest**(**args*, ***kwargs*)
Bases: *partitura.score.GenericNote*

A subclass of GenericNote representing a rest.

class partitura.score.**Beam**(*id=None*)
Bases: *partitura.score.TimedObject*

Represent beams (for MEI)

class partitura.score.**GraceNote**(*grace_type*, **args*, *steal_proportion=None*, ***kwargs*)
Bases: *partitura.score.Note*

A subclass of Note representing a grace note.

Parameters

- **grace_type** (*{'grace', 'acciaccatura', 'appoggiatura'}*) – The type of grace note. Use 'grace' for a unspecified grace note type.
- **steal_proportion** (*float, optional*) – The proportion of the previous (acciaccatura) or next (appoggiatura) note duration that is occupied by the grace note. Defaults to None.

main_note
The (non-grace) note to which this grace note belongs.

Type *Note*

grace_seq_len
The length of the sequence of grace notes to which this grace note belongs.

Type list

iter_grace_seq(*backwards=False*)
Iterate over this and all subsequent/preceding grace notes, excluding the main note.

Parameters **backwards** (*bool, optional*) – When True, iterate over preceding grace notes. Otherwise iterate over subsequent grace notes. Defaults to False.

Yields *GraceNote*

class partitura.score.**Page**(*number=0*)
Bases: *partitura.score.TimedObject*

A page in a musical score. Its start and end times describe the range of musical time that is spanned by the page.

Parameters **number** (*int, optional*) – The number of the system. Defaults to 0.

number
See parameters

Type int

class partitura.score.**System**(*number=0*)
Bases: *partitura.score.TimedObject*

A system in a musical score. Its start and end times describe the range of musical time that is spanned by the system.

Parameters **number** (*int, optional*) – The number of the system. Defaults to 0.

number
See parameters

Type int

class partitura.score.**Clef**(*number*, *sign*, *line*, *octave_change*)
Bases: *partitura.score.TimedObject*

Clefs associate the lines of a staff to musical pitches.

Parameters

- **number** (*int, optional*) – The number of the staff to which this clef belongs.
- **sign** (*{'G', 'F', 'C', 'percussion', 'TAB', 'jianpu', 'none'}*) – The sign of the clef
- **line** (*int*) – The staff line at which the sign is positioned
- **octave_change** (*int*) – The number of octaves to shift the pitches up (postive) or down (negative)

nr
See parameters

Type int

sign
See parameters

Type {'G', 'F', 'C', 'percussion', 'TAB', 'jianpu', 'none'}

line
See parameters

Type int

octave_change
See parameters

Type int

class partitura.score.**Slur**(*start_note=None*, *end_note=None*)
Bases: *partitura.score.TimedObject*

Slurs indicate musical grouping across notes.

Parameters

- **start_note** (*Note*, optional) – The note at which this slur starts. Defaults to None.
- **end_note** (*Note*, optional) – The note at which this slur ends. Defaults to None.

start_note
See parameters

Type *Note* or None

end_note
See parameters

Type *Note* or None

class partitura.score.**Tuplet**(*start_note=None*, *end_note=None*)
Bases: *partitura.score.TimedObject*

Tuplets indicate musical grouping across notes.

Parameters

- **start_note** (*Note*, optional) – The note at which this tuplet starts. Defaults to None.
- **end_note** (*Note*, optional) – The note at which this tuplet ends. Defaults to None.

start_note
See parameters

Type *Note* or None

end_note
See parameters

Type *Note* or None

class partitura.score.**Repeat**
Bases: *partitura.score.TimedObject*

Repeats represent a repeated section in the score, designated by its start and end times.

class partitura.score.**DaCapo**
Bases: *partitura.score.TimedObject*

A Da Capo sign.

class partitura.score.**Fine**
Bases: *partitura.score.TimedObject*

A Fine sign.

class partitura.score.**Fermata**(*ref=None*)
Bases: *partitura.score.TimedObject*

A Fermata sign.

Parameters **ref** (*TimedObject* or None, optional) – An object to which this fermata applies. In practice this is a Note or a Barline. Defaults to None.

ref
See parameters

Type *TimedObject* or None

class partitura.score.**Ending**(*number*)
Bases: *partitura.score.TimedObject*

Class that represents one part of a 1—2— type ending of a musical passage (a.k.a Volta brackets).

Parameters **number** (*int*) – The number associated to this ending

number
See parameters

Type int

class partitura.score.**Barline**(*style*)
Bases: *partitura.score.TimedObject*

Class that represents the style of a barline

class partitura.score.**Measure**(*number=None*)
Bases: *partitura.score.TimedObject*

A measure

Parameters **number** (*int or None, optional*) – The number of the measure. Defaults to None

number
See parameters

Type int

page
The page number on which this measure appears, or None if there is no associated page.

Returns

Return type int or None

system
The system number in which this measure appears, or None if there is no associated system.

Returns

Return type int or None

class partitura.score.**TimeSignature**(*beats*, *beat_type*)
Bases: *partitura.score.TimedObject*

A time signature.

Parameters

- **beats** (*int*) – The number of beats in a measure
- **beat_type** (*int*) – The note type that defines the beat unit. (4 for quarter notes, 2 for half notes, etc.)

beats
See parameters

Type int

beat_type
See parameters

Type int

class partitura.score.**Tempo**(*bpm*, *unit=None*)
Bases: *partitura.score.TimedObject*

A tempo indication.

Parameters

- **bpm** (*number*) – The tempo indicated in rate per minute
- **unit** (*str or None, optional*) – The unit to which the specified rate correspnds. This is a string that expreses a duration category, such as "q" for quarter "h." for dotted half, and so on. When None, the unit is assumed to be quarters. Defaults to None.

bpm
See parameters

Type number

unit
See parameters

Type str or None

microseconds_per_quarter
The number of microseconds per quarter under this tempo.

This is useful for MIDI representations.

Returns

Return type int

class partitura.score.**KeySignature**(*fifths*, *mode*)
Bases: *partitura.score.TimedObject*

Key signature.

Parameters

- **fifths** (*number*) – Number of sharps (positive) or flats (negative)
- **mode** (*str*) – Mode of the key, either 'major' or 'minor'

fifths
See parameters

Type number

mode
See parameters

Type str

name
The key signature name, where the root is uppercase, and an trailing 'm' indicates minor modes (e.g. 'Am', 'G#').

Returns The key signature name

Return type str

class partitura.score.**Transposition**(*diatonic*, *chromatic*)
Bases: *partitura.score.TimedObject*

Represents a <transpose> tag that tells how to change all (following) pitches of that part to put it to concert pitch (i.e. sounding pitch).

Parameters

- **diatonic** (*int*) – TODO
- **chromatic** (*int*) – The number of semi-tone steps to add or subtract to the pitch to get to the (sounding) concert pitch.

diatonic
See parameters

Type int

chromatic
See parameters

Type int

class partitura.score.**Words**(*text*, *staff=None*)
Bases: *partitura.score.TimedObject*

A textual element in the score.

Parameters

- **text** (*str*) – The text
- **staff** (*int or None, optional*) – The staff to which the text is associated. Defaults to None

text
See parameters

Type str

staff
See parameters

Type int or None, optional

class partitura.score.**Direction**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.TimedObject*

Base class for performance directions in the score.

class partitura.score.**LoudnessDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**TempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ArticulationDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**PedalDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ConstantDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**DynamicDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ImpulsiveDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ConstantLoudnessDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**ConstantTempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.TempoDirection*

class partitura.score.**ConstantArticulationDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.ArticulationDirection*

class partitura.score.**DynamicLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**DynamicTempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicDirection*, *partitura.score.TempoDirection*

class partitura.score.**IncreasingLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicLoudnessDirection*

class partitura.score.**DecreasingLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicLoudnessDirection*

class partitura.score.**IncreasingTempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicTempoDirection*

class partitura.score.**DecreasingTempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicTempoDirection*

class partitura.score.**ImpulsiveLoudnessDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ImpulsiveDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**SustainPedalDirection**(*line=False*, **args*, ***kwargs*)
Bases: *partitura.score.PedalDirection*

Represents a Sustain Pedal Direction

class partitura.score.**ResetTempoDirection**(*text=None*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantTempoDirection*

class partitura.score.**PartGroup**(*group_symbol=None*, *group_name=None*, *number=None*)
Bases: `object`

Represents a grouping of several instruments, usually named, and expressed in the score with a group symbol such as a brace or a bracket. In symphonic scores, bracketed part groups usually group families of instruments, such as woodwinds or brass, whereas braces are often used to group multiple instances of the same instrument. See the MusicXML documentation for further information.

Parameters **group_symbol** (*str or None, optional*) – The symbol used for grouping instruments.

group_symbol

Type str or None

name

Type str or None

number

Type int

parent

Type *PartGroup* or None

children

Type list of Part or PartGroup objects

pretty()
Return a pretty representation of this object.

Returns A pretty representation

Return type str

note_array
A structured array containing pitch, onset, duration, voice and id for each note in each part of the PartGroup. The note ids in this array include the number of the part to which they belong.

partitura.score.**iter_unfolded_parts**(*part*)
Iterate over unfolded clones of *part*.

For each repeat construct in *part* the iterator produces two clones, one with the repeat included and another without the repeat. That means the number of items returned is two to the power of the number of repeat constructs in the part.

The first item returned by the iterator is the version of the part without any repeated sections, the last item is the version of the part with all repeat constructs expanded.

Parameters **part** (*Part*) – Part to unfold

partitura.score.**unfold_part_maximal**(*part*, *update_ids=False*)
Return the "maximally" unfolded part, that is, a copy of the part where all segments marked with repeat signs are included twice.

Parameters

- **part** (*Part*) – The Part to unfold.
- **update_ids** (*bool (optional)*) – Update note ids to reflect the repetitions. Note IDs will have a '-<repetition number>', e.g., 'n132-1' and 'n132-2' represent the first and second repetition of 'n132' in the input *part*. Defaults to False.

Returns **unfolded_part** – The unfolded Part

Return type *Part*

partitura.score.**unfold_part_alignment**(*part*, *alignment*)
Return the unfolded part given an alignment, that is, a copy of the part where the segments are repeated according to the repetitions in a performance.

Parameters

- **part** (*Part*) – The Part to unfold.
- **alignment** (*list of dictionaries*) – List of dictionaries containing an alignment (like the ones obtained from a MatchFile (see *alignment_from_matchfile*).

Returns **unfolded_part** – The unfolded Part

Return type *Part*

partitura.score.**make_score_variants**(*part*)
Create a list of ScoreVariant objects, each representing a distinct way to unfold the score, based on the repeat structure.

Parameters **part** (*Part*) – A part for which to make the score variants

Returns List of ScoreVariant objects

Return type list

Notes

This function does not currently support nested repeats, such as in case 45d of the MusicXML Test Suite.

partitura.score.**add_measures**(*part*)
Add measures to a part.

This function adds Measure objects to the part according to any time signatures present in the part. Any existing measures will be untouched, and added measures will be delimited by the existing measures.

The Part object will be modified in place.

Parameters **part** (*Part*) – Part instance

partitura.score.**remove_grace_notes**(*part*)
Remove all grace notes from a timeline.

The specified timeline object will be modified in place.

Parameters **timeline** (*Timeline*) – The timeline from which to remove the grace notes

partitura.score.**expand_grace_notes**(*part*)
Expand grace note durations in a part.

The specified part object will be modified in place.

Parameters **part** (*Part*) – The part on which to expand the grace notes

partitura.score.**iter_parts**(*partlist*)
Iterate over all Part instances in partlist, which is a list of either Part or PartGroup instances. PartGroup instances contain one or more parts or further partgroups, and are traversed in a depth-first fashion.

This function is designed to take the result of *partitura.load_score_midi()* and *partitura.load_musicxml()* as input.

Parameters **partlist** (*list, Part, or PartGroup*) – A *partitura.score.Part* object, *partitura.score.PartGroup* or a list of these

Yields *Part* instances in *partlist*

partitura.score.**repeats_to_start_end**(*repeats*, *first*, *last*)
Return pairs of (start, end) TimePoints corresponding to the start and end times of each Repeat object. If any of the start or end attributes are None, replace it with the end/start of the preceding/succeeding Repeat, respectively, or *first* or *last*.

Parameters

- **repeats** (*list*) – list of Repeat instances, possibly with None-valued start/end attributes
- **first** (TimePoint) – The first TimePoint in the timeline
- **last** (TimePoint) – The last TimePoint in the timeline

Returns list of (start, end) TimePoints corresponding to each Repeat in *repeats*

Return type list

partitura.score.**tie_notes**(*part*)
Find notes that span measure boundaries and notes with composite durations, and split them adding ties.

Parameters **part** (*Part*) – Description of *part*

partitura.score.**set_end_times**(*parts*)
Set missing end times of musical elements in a part to equal the start times of the subsequent element of the same class. This is useful for some classes

This function modifies the parts in place.

Parameters **part** (*Part or PartGroup, or list of these*) – Parts to be processed

partitura.score.**find_tuplets**(*part*)
Identify tuplets in *part* and set their symbolic durations explicitly.

This function adds *actual_notes* and *normal_notes* keys to the symbolic duration of tuplet notes.

This function modifies the part in place.

Parameters **part** (*Part*) – Part instance

partitura.score.**sanitize_part**(*part*)

Find and remove incomplete structures in a part such as Tuplets and Slurs without start or end and grace notes without a main note.

This function modifies the part in place.

Parameters **part** (*Part*) – Part instance

exception partitura.score.**InvalidTimePointException**(*message=None*)

Bases: Exception

Raised when a time point is instantiated with an invalid number.

CHAPTER 6

partitura.performance

This module contains a lightweight ontology to represent a performance in a MIDI-like format. A performance is defined at the highest level by a *PerformedPart*. This object contains performed notes as well as continuous control parameters, such as sustain pedal.

class partitura.performance.**PerformedPart**(*notes*, *id=None*, *part_name=None*, *controls=None*, *programs=None*, *sustain_pedal_threshold=64*)

Bases: object

Represents a performed part, e.g. all notes and related controller/modifiers of one single instrument.

Performed notes are stored as a list of dictionaries, where each dictionary represents a performed note, should have at least the keys "note_on", "note_off", the onset and offset times of the note in seconds, respectively.

Continuous controls are also stored as a list of dictionaries, where each dictionary represents a control change. Each dictionary should have a key "type" (the name of the control, e.g. "sustain_pedal", "soft_pedal"), "time" (in seconds), and "value" (a number).

Parameters

- **notes** (*list*) – A list of dictionaries containing performed note information.
- **id** (*str*) – The identifier of the part
- **controls** (*list*) – A list of dictionaries containing continuous control information
- **part_name** (*str*) – Name for the part
- **sustain_pedal_threshold** (*int*) – The threshold above which sustain pedal values are considered to be equivalent to on. For values below the threshold the sustain pedal is treated as off. Defaults to 64.

notes

A list of dictionaries containing performed note information.

Type list

id

The identifier of the part

Type str

part_name
Name for the part

Type str

controls
A list of dictionaries containing continuous control information

Type list

programs
List of dictionaries containing program change information

Type list

classmethod from_note_array(*note_array*, *id=None*, *part_name=None*)
Create an instance of PerformedPart from a note_array. Note that this property does not include non-note information (i.e. controls such as sustain pedal).

note_array
Structured array containing performance information. The fields are 'id', 'pitch', 'onset_div', 'duration_div', 'onset_sec', 'duration_sec' and 'velocity'.

sustain_pedal_threshold
The threshold value (number) above which sustain pedal values are considered to be equivalent to on. For values below the threshold the sustain pedal is treated as off. Defaults to 64.

Based on the control items of type "sustain_pedal", in combination with the value of the "sustain_pedal_threshold" attribute, the note dictionaries will be extended with a key "sound_off". This key represents the time the note will stop sounding. When the sustain pedal is off, *sound_off* will coincide with *note_off*. When the sustain pedal is on, *sound_off* will equal the earliest time the sustain pedal is off after *note_off*. The *sound_off* values of notes will be automatically recomputed each time the *sustain_pedal_threshold* is set.

CHAPTER 7

partitura.musicanalysis

Tools for music analysis.

partitura.musicanalysis.**estimate_voices**(*note_info*, *monophonic_voices=False*)

Voice estimation using the voice separation algorithm proposed in[6].

Parameters

- **note_info** (structured array, *Part* or *PerformedPart*) – Note information as a *Part* or *PerformedPart* instances or as a structured array. If it is a structured array, it has to contain the fields generated by the *note_array* properties of *Part* or *PerformedPart* objects. If the array contains onset and duration information of both score and performance, (e.g., containing both *onset_beat* and *onset_sec*), the score information will be preferred.
- **monophonic_voices** (*bool*) – If True voices are guaranteed to be monophonic. Otherwise notes with the same onset and duration are treated as a chord and assigned to the same voice. Defaults to False.

Returns **voice** – Voice for each note in the notearray. (The voices start with 1, as is the MusicXML convention).

Return type numpy array

References

partitura.musicanalysis.**estimate_key**(*note_info*, *method='krumhansl'*, **args*, ***kwargs*)

Estimate key of a piece by comparing the pitch statistics of the note array to key profiles[2],[3].

Parameters

[6] Chew, E. and Wu, Xiaodan (2004) "Separating Voices in Polyphonic Music: A Contig Mapping Approach". In Uffe Kock, editor, "Computer Music Modeling and Retrieval". Springer Berlin Heidelberg.

[2] Krumhansl, Carol L. (1990) "Cognitive foundations of musical pitch", Oxford University Press, New York.

[3] Temperley, D. (1999) "What's key for key? The Krumhansl-Schmuckler key-finding algorithm reconsidered". Music Perception. 17(1), pp. 65–100.

- **note_info** (structured array, *Part* or *PerformedPart*) – Note information as a *Part* or *PerformedPart* instances or as a structured array. If it is a structured array, it has to contain the fields generated by the *note_array* properties of *Part* or *PerformedPart* objects. If the array contains onset and duration information of both score and performance, (e.g., containing both *onset_beat* and *onset_sec*), the score information will be preferred.
- **method** (*{'krumhansl'}*) – Method for estimating the key. For now 'krumhansl' is the only supported method.
- **kwargs** (*args,*) – Positional and Keyword arguments for the key estimation method

Returns String representing the key name (i.e., Root(alteration)(m if minor)). See *partitura.utils.key_name_to_fifths_mode* and *partitura.utils.fifths_mode_to_key_name*.

Return type str

References

partitura.musicanalysis.**estimate_spelling**(*note_info*, *method='ps13s1'*, ***kwargs*)

Estimate pitch spelling using the ps13 algorithm[4],[5].

Parameters

- **note_info** (structured array, *Part* or *PerformedPart*) – Note information as a *Part* or *PerformedPart* instances or as a structured array. If it is a structured array, it has to contain the fields generated by the *note_array* properties of *Part* or *PerformedPart* objects. If the array contains onset and duration information of both score and performance, (e.g., containing both *onset_beat* and *onset_sec*), the score information will be preferred.
- **method** (*{'ps13s1'}*) – Pitch spelling algorithm. More methods will be added.
- ****kwargs** – Keyword arguments for the algorithm specified in *method*.

Returns **spelling** – Array with pitch spellings. The fields are 'step', 'alter' and 'octave'

Return type structured array

References

partitura.musicanalysis.**estimate_tonaltension**(*note_info*, *ws=1.0*, *ss='onset'*, *scale_factor=0.09249316305671976*, *w=array([0.516, 0.315, 0.168])*, *alpha=0.75*, *beta=0.75*)

Compute tonal tension ribbons defined in[1]

Parameters

- **note_info** (structured array, *Part* or *PerformedPart*) – Note information as a *Part* or *PerformedPart* instances or as a structured array. If it is a structured array, it has to contain the fields generated by the *note_array* properties of *Part* or *PerformedPart* objects. If the array contains onset and duration information of both score and performance, (e.g., containing both *onset_beat* and *onset_sec*), the score information will be preferred. Furthermore, this method requires pitch spelling and key signature information. If a structured note array is provided as input, this information can be optionally provided in fields

[4] Meredith, D. (2006). "The ps13 Pitch Spelling Algorithm". Journal of New Music Research, 35(2):121.

[5] Meredith, D. (2019). "RecurSIA-RRT: Recursive translatable point-set pattern discovery with removal of redundant translators". 12th International Workshop on Machine Learning and Music. Würzburg, Germany.

[1] D. Herremans and E. Chew (2016) Tension ribbons: Quantifying and visualising tonal tension. Proceedings of the Second International Conference on Technologies for Music Notation and Representation (TENOR), Cambridge, UK.

step, *alter*, *ks_fifths* and *ks_mode*. If these fields are not found in the input structured array, they will be estimated using the key and pitch spelling estimation methods from *partitura.musicanalysis.estimate_key* and and *partitura.musicanalysis.estimate_spelling*, respectively.

- **ws** (*{int, float, np.array}, optional*) – Window size for computing the tonal tension. If a number, it determines the size of the window centered at each specified score position (see *ss* below). If a numpy array, a 2D array of shape (*len(ss)*, 2) specifying the left and right distance from each score position in *ss*. Default is 1 beat.
- **ss** (*{float, int, np.array, 'onset'}, optional.*) – Step size or score position for computing the tonal tension features. If a number, this parameter determines the size of the step (in beats) starting from the first score position. If an array, it specifies the score positions at which the tonal tension is estimated. If 'onset', it computes the tension at each unique score position (i.e., all notes in a chord have the same score position). Default is 'onset'.
- **scale_factor** (*float*) – A multiplicative scaling factor.
- **w** (*np.ndarray*) – Weights for the chords
- **alpha** (*float*) – Alpha.
- **beta** (*float*) – Beta.

Returns tonal_tension – Array containing the tonal tension features. It contains the fields *cloud_diameter*, *cloud_momentum*, *tensile_strain* and *onset*.

Return type structured array

References

CHAPTER 8

partitura.utils

partitura.utils.**key_name_to_fifths_mode**(*name*)

Return the number of sharps or flats and the mode of a key signature name. A negative number denotes the number of flats (i.e. -3 means three flats), and a positive number the number of sharps. The mode is specified as 'major' or 'minor'.

Parameters **name** ({*"A", "A#m", "Ab", "Abm", "Am", "B", "Bb", "Bbm", "Bm", "C","C#", "C#m", "Cb", "Cm", "D", "D#m", "Db", "Dm", "E", "Eb","Ebm", "Em", "F", "F#", "F#m", "Fm", "G", "G#m", "Gb", "Gm"*}) – Name of the key signature

Returns Tuple containing the number of fifths and the mode

Return type (int, str)

Examples

```
>>> key_name_to_fifths_mode('Am')
(0, 'minor')
>>> key_name_to_fifths_mode('C')
(0, 'major')
>>> key_name_to_fifths_mode('A')
(3, 'major')
```

partitura.utils.**fifths_mode_to_key_name**(*fifths*, *mode=None*)

Return the key signature name corresponding to a number of sharps or flats and a mode. A negative value for *fifths* denotes the number of flats (i.e. -3 means three flats), and a positive number the number of sharps. The mode is specified as 'major' or 'minor'. If *mode* is None, the key is assumed to be major.

Parameters

- **fifths** (*int*) – Number of fifths
- **mode** ({*'major', 'minor', None, -1, 1*}) – Mode of the key signature

Returns The name of the key signature, e.g. 'Am'

Return type str

Examples

```
>>> fifths_mode_to_key_name(0, 'minor')
'Am'
>>> fifths_mode_to_key_name(0, 'major')
'C'
>>> fifths_mode_to_key_name(3, 'major')
'A'
>>> fifths_mode_to_key_name(-1, 1)
'F'
```

partitura.utils.**key_mode_to_int**(*mode*)

Return the mode of a key as an integer (1 for major and -1 for minor).

Parameters **mode** (*{'major', 'minor', None, 1, -1}*) – Mode of the key

Returns Integer representation of the mode.

Return type int

partitura.utils.**compute_pianoroll**(*note_info*, *time_unit='auto'*, *time_div='auto'*, *onset_only=False*, *note_separation=False*, *pitch_margin=-1*, *time_margin=0*, *return_idxs=False*, *piano_range=False*, *remove_drums=True*)

Computes a piano roll from a structured note array (as generated by the *note_array* methods in *partitura.score.Part* and *partitura.performance.PerformedPart* instances).

Parameters

- **note_info** (structured array, *Part*, *PartGroup*, *PerformedPart*) – Note information
- **time_unit** (*('auto', 'beat', 'quarter', 'div', 'second')*) –
- **time_div** (*int, optional*) – How many sub-divisions for each time unit (beats for a score or seconds for a performance. See *is_performance* below).
- **onset_only** (*bool, optional*) – If True, code only the onsets of the notes, otherwise code onset and duration.
- **pitch_margin** (*int, optional*) – If *pitch_margin* > -1, the resulting array will have *pitch_margin* empty rows above and below the highest and lowest pitches, respectively; if *pitch_margin* == -1, the resulting pianoroll will have span the fixed pitch range between (and including) 1 and 127.
- **time_margin** (*int, optional*) – The resulting array will have *time_margin* * *time_div* empty columns before and after the piano roll
- **return_idxs** (*bool, optional*) – If True, return the indices (i.e., the coordinates) of each note in the piano roll.
- **piano_range** (*bool, optional*) – If True, the pitch axis of the piano roll is in piano keys instead of MIDI note numbers (and there are only 88 pitches). This is equivalent as slicing *piano_range_pianoroll = pianoroll[21:109, :]*.
- **remove_drums** (*bool, optional*) – If True, removes the drum track (i.e., channel 9) from the notes to be considered in the piano roll. This option is only relevant for piano rolls generated from a *PerformedPart*. Default is True.

Returns

- **pianoroll** (*scipy.sparse.csr_matrix*) – A sparse int matrix of size representing the pianoroll; The first dimension is pitch, the second is time; The sizes of the dimensions vary with the parameters *pitch_margin*, *time_margin*, and *time_div*
- **pr_idx** (*ndarray*) – Indices of the onsets and offsets of the notes in the piano roll (in the same order as the input note_array). This is only returned if *return_idxs* is *True*.

Examples

```
>>> import numpy as np
>>> from partitura.utils import compute_pianoroll
>>> note_array = np.array([(60, 0, 1)], dtype=[('pitch',
↪'i4'),                                 ('onset_beat', 'f4'),
↪              ('duration_beat', 'f4')])
>>> pr = compute_pianoroll(note_array, pitch_margin=2, time_div=2)
>>> pr.toarray()
array([[0, 0],
       [0, 0],
       [1, 1],
       [0, 0],
       [0, 0]])
```

Notes

The default values in this function assume that the input *note_array* represents a score.

partitura.utils.**pianoroll_to_notearray**(*pianoroll*, *time_div=8*, *time_unit='sec'*)

Extract a structured note array from a piano roll.

For now, the structured note array is considered a "performance".

Parameters

- **pianoroll** (*array-like*) – 2D array containing a piano roll. The first dimension is pitch, and the second is time. The value of each "pixel" in the piano roll is considered to be the MIDI velocity, and it is supposed to be between 0 and 127.
- **time_div** (*int*) – How many sub-divisions for each time unit (see *notearray_to_pianoroll*).
- **time_unit** (*{'beat', 'quarter', 'div', 'sec'}*) – time unit of the output note array.

Returns Structured array with pitch, onset, duration and velocity fields.

Return type np.ndarray

Notes

Please note that all non-zero pixels will contribute to a note. For the case of piano rolls with continuous values between 0 and 1 (as might be the case of those piano rolls produced using probabilistic/generative models), we recomend to either 1) hard- threshold the piano roll to have only 0s (note-off) or 1s (note- on) or, 2) soft-threshold the notes (values below a certain threshold are considered as not active and scale the active notes to lie between 1 and 127).

Python Module Index

p

Index

G

I

K

L

M

N

O

P

Q

R

S

T

U

W