partitura

Release 1.0.0

Oct 03, 2022

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
```

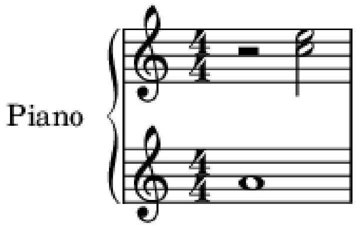
For convenience a MusicXML file with the above contents is included in the package. The path to the file is stored as `partitura.EXAMPLE_MUSICXML`, so that we load the above score as follows:

```
>>> 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()`:

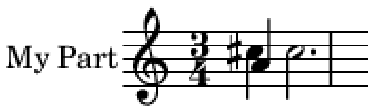
```
>>> score.add_measures(part)
```

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)
```

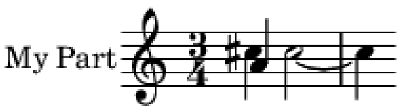


Although the notes are there, the music is not typeset correctly, since the first measure should have a duration of three quarter notes, but instead is has a duration of four quarter notes. The problem is that the note *n2* crosses a measure boundary, and thus should be tied.

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

CHAPTER 5

partitura.score

CHAPTER 6

partitura.performance

CHAPTER 7

partitura.musicanalysis

CHAPTER 8

partitura.utils