madmom Documentation

Release 0.13.2

madmom development team

June 09, 2016

CHAPTER 1

Introduction

Madmom is an audio signal processing library written in Python with a strong focus on music information retrieval (MIR) tasks. The project is on GitHub.

It's main features / design goals are:

- ease of use,
- rapid prototyping of signal processing workflows,
- most things are modeled as numpy arrays (enhanced by additional methods and attributes),
- simple conversion of a workflow to a running program by the use of processors,
- no dependencies on other software packages (not even for machine learning stuff),
- inclusion of reference implementations for several state-of-the-art algorithms.

Madmom is a work in progress, thus input is always welcome.

The available documentation is limited for now, but *you can help to improve it*.

CHAPTER 2

Installation

Please do not try to install from the .zip files provided by GitHub. Rather install either install *from package* (if you just want to use it) or *from source* (if you plan to use it for development). Whichever variant you choose, please make sure that all *prerequisites* are installed.

2.1 Prerequisites

To install the `madmom` package, you must have either Python 2.7 or Python 3.3 or newer and the following packages installed:

- numpy
- scipy
- cython
- nose (to run the tests)

If you need support for audio files other than `.wav` with a sample rate of 44.1kHz and 16 bit depth, you need `ffmpeg` (`avconv` on Ubuntu Linux has some decoding bugs, so we advise not to use it!).

Please refer to the `requirements.txt` file for the minimum required versions and make sure that these modules are up to date, otherwise it can result in unexpected errors or false computations!

2.2 Install from package

The instructions given here should be used if you just want to install the package, e.g. to run the bundled programs or use some functionality for your own project. If you intend to change anything within the *madmom* package, please follow the steps in *the next section*.

The easiest way to install the package is via `pip` from the PyPI (Python Package Index):

```
pip install madmom
```

This includes the latest code and trained models and will install all dependencies automatically.

You might need higher privileges (use su or sudo) to install the package, model files and scripts globally. Alternatively you can install the package locally (i.e. only for you) by adding the `--user` argument:

```
pip install --user madmom
```

This will also install the executable programs to a common place (e.g. `/usr/local/bin`), which should be in your `$PATH` already. If you installed the package locally, the programs will be copied to a folder which might not be included in your `$PATH` (e.g. `~/Library/Python/2.7/bin` on Mac OS X or `~/.local/bin` on Ubuntu Linux, `pip` will tell you). Thus the programs need to be called explicitely or you can add their install path to your `$PATH` environment variable:

```
export PATH='path/to/scripts':$PATH
```

2.3 Install from source

If you plan to use the package as a developer, clone the Git repository:

```
git clone --recursive https://github.com/CPJKU/madmom.git
```

Since the pre-trained model/data files are not included in this repository but rather added as a Git submodule, you either have to clone the repo recursively. This is equivalent to these steps:

```
git clone  https://github.com/CPJKU/madmom.git
cd madmom
git submodule update --init --remote
```

You can then either include the package directory in your `$PYTHONPATH` and compile the Cython extensions with:

```
python setup.py build_ext --inplace
```

or you can simply install the package in development mode:

```
python setup.py develop --user
```

To run the included tests:

```
python setup.py test
```

2.4 Upgrade of existing installations

To upgrade the package, please use the same machanism (pip vs. source / global vs. local install) as you did for installation. If you want to change from package to source, please uninstall the package first.

2.4.1 Upgrade a package

Simply upgrade the package via pip:

```
pip install --upgrade madmom [--user]
```

If some of the provided programs or models changed (please refer to the CHANGELOG) you should first uninstall the package and then reinstall:

```
pip uninstall madmom
pip install madmom [--user]
```

2.4.2 Upgrade from source

Simply pull the latest sources:

```
git pull
```

To update the models contained in the submodule:

```
git submodule update
```

If any of the `.pyx` or `.pxd` files changes, you have to recompile the modules with Cython:

```
python setup.py build_ext --inplace
```

CHAPTER 3

Usage

3.1 Executable programs

The package includes executable programs in the `/bin` folder. These are standalone reference implementations of the algorithms contained in the package. If you just want to try/use these programs, please follow the *instruction to install from a package*.

All scripts can be run in different modes: in `single` file mode to process a single audio file and write the output to STDOUT or the given output file:

```
SuperFlux single [-o OUTFILE] INFILE
```

If multiple audio files should be processed, the scripts can also be run in `batch` mode to write the outputs to files with the given suffix:

```
SuperFlux batch [-o OUTPUT_DIR] [-s OUTPUT_SUFFIX] LIST OF INPUT FILES
```

If no output directory is given, the program writes the output files to same location as the audio files.

The `pickle` mode can be used to store the used parameters to be able to exactly reproduce experiments.

3.2 Library usage

To use the library, *installing it from source* is the preferred way. Installation from package works as well, but you're limited to the functionality provided and can't extend the library.

The basic usage is:

```
import madmom
import numpy as np
```

To learn more about how to use the library please follow the tutorials.

CHAPTER 4

Tutorials

This page gives instructions on how to use the package. They are bundled as a loose collection of jupyter (IPython) notebooks.

You can view them online:

https://github.com/CPJKU/madmom_tutorials

CHAPTER 5

Development

As an open-source project by researchers for researchers, we highly welcome any contribution!

5.1 What to contribute

5.1.1 Give feedback

To send us general feedback, questions or ideas for improvement, please post on our mailing list.

5.1.2 Report bugs

Please report any bugs at the issue tracker on GitHub. If you are reporting a bug, please include:

- your version of madmom,
- steps to reproduce the bug, ideally reduced to as few commands as possible,
- the results you obtain, and the results you expected instead.

If you are unsure whether the experienced behaviour is intended or a bug, please just ask on our mailing list first.

5.1.3 Fix bugs

Look for anything tagged with "bug" on the issue tracker on GitHub and fix it.

5.1.4 Features

Please do not hesitate to propose any ideas at the issue tracker on GitHub. Think about posting them on our mailing list first, so we can discuss it and/or guide you through the implementation.

Alternatively, you can look for anything tagged with "feature request" or "enhancement" on the issue tracker on GitHub.

5.1.5 Write documentation

Whenever you find something not explained well, misleading or just wrong, please update it! The *Edit on GitHub* link on the top right of every documentation page and the *[source]* link for every documented entity in the API reference will help you to quickly locate the origin of any text.

5.2 How to contribute

5.2.1 Edit on GitHub

As a very easy way of just fixing issues in the documentation, use the *Edit on GitHub* link on the top right of a documentation page or the *[source]* link of an entity in the API reference to open the corresponding source file in GitHub, then click the *Edit this file* link to edit the file in your browser and send us a Pull Request.

For any more substantial changes, please follow the steps below.

5.2.2 Fork the project

First, fork the project on GitHub.

Then, follow the general installation instructions and, more specifically, the *installation from source*. Please note that you should clone from your fork instead.

5.2.3 Documentation

The documentation is generated with Sphinx. To build it locally, run the following commands:

```
cd docs
make html
```

Afterwards, open `docs/_build/html/index.html` to view the documentation as it would appear on readthedocs. If you changed a lot and seem to get misleading error messages or warnings, run `make clean html` to force Sphinx to recreate all files from scratch.

When writing docstrings, follow existing documentation as much as possible to ensure consistency throughout the library. For additional information on the syntax and conventions used, please refer to the following documents:

- reStructuredText Primer
- Sphinx reST markup constructs
- A Guide to NumPy/SciPy Documentation

CHAPTER 6

madmom.audio

This package includes audio handling functionality and low-level features. The definition of "low" may vary, but all "high"-level features (e.g. beats, onsets, etc. – basically everything you want to evaluate) should be in the *madmom.features* package.

6.1 Notes

Almost all functionality blocks are split into two classes:

1. A data class: instances are signal dependent, i.e. they operate directly on the signal and show different values for different signals.
2. A processor class: for every data class there should be a processor class with the exact same name and a "Processor" suffix. This class must inherit from madmom.Processor and define a process() method which returns a data class or inherit from madmom.SequentialProcessor or ParallelProcessor.

The data classes should be either sub-classed from numpy arrays or be indexable and iterable. This way they can be used identically to numpy arrays.

6.2 Submodules

6.2.1 madmom.audio.signal

This module contains basic signal processing functionality.

madmom.audio.signal.**smooth**(*signal*, *kernel*)

Smooth the signal along its first axis.

Parameters **signal** : numpy array

Signal to be smoothed.

kernel : numpy array or int

Smoothing kernel (size).

Returns numpy array

Smoothed signal.

Notes

If *kernel* is an integer, a Hamming window of that length will be used as a smoothing kernel.

madmom.audio.signal.**adjust_gain**(*signal*, *gain*)
" Adjust the gain of the signal.

Parameters **signal** : numpy array

Signal to be adjusted.

gain : float

Gain adjustment level [dB].

Returns numpy array

Signal with adjusted gain.

Notes

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

gain values > 0 amplify the signal and are only supported for signals with float dtype to prevent clipping and integer overflows.

madmom.audio.signal.**attenuate**(*signal*, *attenuation*)
Attenuate the signal.

Parameters **signal** : numpy array

Signal to be attenuated.

attenuation : float

Attenuation level [dB].

Returns numpy array

Attenuated signal (same dtype as *signal*).

Notes

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

madmom.audio.signal.**normalize**(*signal*)
Normalize the signal to have maximum amplitude.

Parameters **signal** : numpy array

Signal to be normalized.

Returns numpy array

Normalized signal.

Notes

Signals with float dtypes cover the range [-1, +1], signals with integer dtypes will cover the maximally possible range, e.g. [-32768, 32767] for np.int16.

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

madmom.audio.signal.**remix**(*signal*, *num_channels*)
Remix the signal to have the desired number of channels.

Parameters **signal** : numpy array

Signal to be remixed.

num_channels : int

Number of channels.

Returns numpy array

Remixed signal (same dtype as *signal*).

Notes

This function does not support arbitrary channel number conversions. Only down-mixing to and up-mixing from mono signals is supported.

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

If the signal should be down-mixed to mono and has an integer dtype, it will be converted to float internally and then back to the original dtype to prevent clipping of the signal. To avoid this double conversion, convert the dtype first.

madmom.audio.signal.**rescale**(*signal*, *dtype=<type 'numpy.float32'>*)
Rescale the signal to range [-1, 1] and return as float dtype.

Parameters **signal** : numpy array

Signal to be remixed.

dtype : numpy dtype

Data type of the signal.

Returns numpy array

Signal rescaled to range [-1, 1].

madmom.audio.signal.**trim**(*signal*, *where='fb'*)
Trim leading and trailing zeros of the signal.

Parameters **signal** : numpy array

Signal to be trimmed.

where : str, optional

A string with 'f' representing trim from front and 'b' to trim from back. Default is 'fb', trim zeros from both ends of the signal.

Returns numpy array

Trimmed signal.

madmom.audio.signal.**root_mean_square**(*signal*)

Computes the root mean square of the signal. This can be used as a measurement of power.

Parameters **signal** : numpy array

Signal.

Returns **rms** : float

Root mean square of the signal.

madmom.audio.signal.**sound_pressure_level**(*signal*, *p_ref=None*)

Computes the sound pressure level of a signal.

Parameters **signal** : numpy array

Signal.

p_ref : float, optional

Reference sound pressure level; if 'None', take the max amplitude value for the data-type, if the data-type is float, assume amplitudes are between -1 and +1.

Returns **spl** : float

Sound pressure level of the signal [dB].

Notes

From http://en.wikipedia.org/wiki/Sound_pressure: Sound pressure level (SPL) or sound level is a logarithmic measure of the effective sound pressure of a sound relative to a reference value. It is measured in decibels (dB) above a standard reference level.

madmom.audio.signal.**load_wave_file**(*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*)

Load the audio data from the given file and return it as a numpy array.

Only supports wave files, does not support re-sampling or arbitrary channel number conversions. Reads the data as a memory-mapped file with copy-on-write semantics to defer I/O costs until needed.

Parameters **filename** : string

Name of the file.

sample_rate : int, optional

Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels : int, optional

Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

dtype : numpy data type, optional

The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Returns **signal** : numpy array

Audio signal.

sample_rate : int

Sample rate of the signal [Hz].

Notes

The *start* and *stop* positions are rounded to the closest sample; the sample corresponding to the *stop* value is not returned, thus consecutive segment starting with the previous *stop* can be concatenated to obtain the original signal without gaps or overlaps.

exception madmom.audio.signal.**LoadAudioFileError**(*value=None*)
Exception to be raised whenever an audio file could not be loaded.

madmom.audio.signal.**load_audio_file**(*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*)
Load the audio data from the given file and return it as a numpy array. This tries load_wave_file() load_ffmpeg_file() (for ffmpeg and avconv).

Parameters **filename** : str or file handle

Name of the file or file handle.

sample_rate : int, optional

Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels: int, optional

Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

dtype : numpy data type, optional

The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Returns **signal** : numpy array

Audio signal.

sample_rate : int

Sample rate of the signal [Hz].

Notes

For wave files, the *start* and *stop* positions are rounded to the closest sample; the sample corresponding to the *stop* value is not returned, thus consecutive segment starting with the previous *stop* can be concatenated to obtain the original signal without gaps or overlaps. For all other audio files, this can not be guaranteed.

class madmom.audio.signal.**Signal**(*data*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *norm=False*, *gain=0*, *dtype=None*)

The *Signal* class represents a signal as a (memory-mapped) numpy array and enhances it with a number of attributes.

Parameters **data** : numpy array, str or file handle

Signal data or file name or file handle.

sample_rate : int, optional

Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels : int, optional

Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

norm : bool, optional

Normalize the signal to the range [-1, +1].

gain : float, optional

Adjust the gain of the signal [dB].

dtype : numpy data type, optional

The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Notes

sample_rate or *num_channels* can be used to set the desired sample rate and number of channels if the audio is read from file. If set to 'None' the audio signal is used as is, i.e. the sample rate and number of channels are determined directly from the audio file.

If the *data* is a numpy array, the *sample_rate* is set to the given value and *num_channels* is set to the number of columns of the array.

The *gain* can be used to adjust the level of the signal.

If both *norm* and *gain* are set, the signal is first normalized and then the gain is applied afterwards.

If *norm* or *gain* is set, the selected part of the signal is loaded into memory completely, i.e. .wav files are not memory-mapped any more.

num_samples
Number of samples.

num_channels
Number of channels.

length
Length of signal in seconds.

class madmom.audio.signal.**SignalProcessor**(*sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *norm=False*, *att=None*, *gain=0.0*, ***kwargs*)

The *SignalProcessor* class is a basic signal processor.

Parameters **sample_rate** : int, optional

Sample rate of the signal [Hz]; if set the signal will be re-sampled to that sample rate; if 'None' the sample rate of the audio file will be used.

num_channels : int, optional

Number of channels of the signal; if set, the signal will be reduced to that number of channels; if 'None' as many channels as present in the audio file are returned.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

norm : bool, optional

Normalize the signal to the range [-1, +1].

att : float, optional

Deprecated in version 0.13, use *gain* instead.

gain : float, optional

Adjust the gain of the signal [dB].

dtype : numpy data type, optional

The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

att

Attenuation of the signal [dB].

process(*data*, *start=None*, *stop=None*, ***kwargs*)

Processes the given audio file.

Parameters **data** : numpy array, str or file handle

Data to be processed.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

Returns **signal** : *Signal* instance

Signal instance.

static add_arguments(*parser*, *sample_rate=None*, *mono=None*, *start=None*, *stop=None*, *norm=None*, *gain=None*)

Add signal processing related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

sample_rate : int, optional

Re-sample the signal to this sample rate [Hz].

mono : bool, optional

Down-mix the signal to mono.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

norm : bool, optional

Normalize the signal to the range [-1, +1].

gain : float, optional

Adjust the gain of the signal [dB].

Returns argparse argument group

Signal processing argument parser group.

Notes

Parameters are included in the group only if they are not 'None'. To include *start* and *stop* arguments with a default value of 'None', i.e. do not set any start or stop time, they can be set to 'True'.

`madmom.audio.signal.`**`signal_frame`**(*signal*, *index*, *frame_size*, *hop_size*, *origin=0*)

This function returns frame at *index* of the *signal*.

Parameters **signal** : numpy array

Signal.

index : int

Index of the frame to return.

frame_size : int

Size of each frame in samples.

hop_size : float

Hop size in samples between adjacent frames.

origin : int

Location of the window center relative to the signal position.

Returns **frame** : numpy array

Requested frame of the signal.

Notes

The reference sample of the first frame (index == 0) refers to the first sample of the *signal*, and each following frame is placed *hop_size* samples after the previous one.

The window is always centered around this reference sample. Its location relative to the reference sample can be set with the *origin* parameter. Arbitrary integer values can be given:

- zero centers the window on its reference sample
- negative values shift the window to the right
- positive values shift the window to the left

An *origin* of half the size of the *frame_size* results in windows located to the left of the reference sample, i.e. the first frame starts at the first sample of the signal.

The part of the frame which is not covered by the signal is padded with zeros.

This function is totally independent of the length of the signal. Thus, contrary to common indexing, the index '-1' refers NOT to the last frame of the signal, but instead the frame left of the first frame is returned.

class madmom.audio.signal.**FramedSignal**(*signal*, *frame_size=2048*, *hop_size=441.0*, *fps=None*, *origin=0*, *end='normal'*, *num_frames=None*, ***kwargs*)

The *FramedSignal* splits a *Signal* into frames and makes it iterable and indexable.

Parameters **signal** : *Signal* instance

Signal to be split into frames.

frame_size : int, optional

Size of one frame [samples].

hop_size : float, optional

Progress *hop_size* samples between adjacent frames.

fps : float, optional

Use given frames per second; if set, this computes and overwrites the given *hop_size* value.

origin : int, optional

Location of the window relative to the reference sample of a frame.

end : int or str, optional

End of signal handling (see notes below).

num_frames : int, optional

Number of frames to return.

kwargs : dict, optional

If no *Signal* instance was given, one is instantiated with these additional keyword arguments.

Notes

The `FramedSignal` class is implemented as an iterator. It splits the given *signal* automatically into frames of *frame_size* length with *hop_size* samples (can be float, normal rounding applies) between the frames. The reference sample of the first frame refers to the first sample of the *signal*.

The location of the window relative to the reference sample of a frame can be set with the *origin* parameter (with the same behaviour as used by `scipy.ndimage` filters). Arbitrary integer values can be given:

- zero centers the window on its reference sample,
- negative values shift the window to the right,
- positive values shift the window to the left.

Additionally, it can have the following literal values:

- 'center', 'offline': the window is centered on its reference sample,
- 'left', 'past', 'online': the window is located to the left of its reference sample (including the reference sample),
- 'right', 'future': the window is located to the right of its reference sample.

The *end* parameter is used to handle the end of signal behaviour and can have these values:

- 'normal': stop as soon as the whole signal got covered by at least one frame (i.e. pad maximally one frame),
- 'extend': frames are returned as long as part of the frame overlaps with the signal to cover the whole signal.

Alternatively, *num_frames* can be used to retrieve a fixed number of frames.

In order to be able to stack multiple frames obtained with different frame sizes, the number of frames to be returned must be independent from the set *frame_size*. It is not guaranteed that every sample of the signal is returned in a frame unless the *origin* is either 'right' or 'future'.

frame_rate
Frame rate (same as fps).

fps
Frames per second.

overlap_factor
Overlapping factor of two adjacent frames.

shape
Shape of the FramedSignal (frames x samples).

ndim
Dimensionality of the FramedSignal.

class madmom.audio.signal.**FramedSignalProcessor**(*frame_size=2048*, *hop_size=441.0*, *fps=None*, *online=False*, *end='normal'*, ***kwargs*)

Slice a Signal into frames.

Parameters **frame_size** : int, optional

Size of one frame [samples].

hop_size : float, optional

Progress *hop_size* samples between adjacent frames.

fps : float, optional

Use given frames per second; if set, this computes and overwrites the given *hop_size* value.

online : bool, optional

Operate in online mode (see notes below).

end : int or str, optional

End of signal handling (see *FramedSignal*).

num_frames : int, optional

Number of frames to return.

kwargs : dict, optional

If no *Signal* instance was given, one is instantiated with these additional keyword arguments.

Notes

The location of the window relative to its reference sample can be set with the *online* parameter:

- 'False': the window is centered on its reference sample,
- 'True': the window is located to the left of its reference sample (including the reference sample), i.e. only past information is used.

process (*data*, ***kwargs*)
Slice the signal into (overlapping) frames.

Parameters **data** : *Signal* instance

Signal to be sliced into frames.

kwargs : dict

Keyword arguments passed to *FramedSignal* to instantiate the returned object.

Returns **frames** : *FramedSignal* instance

FramedSignal instance

static add_arguments (*parser*, *frame_size=2048*, *fps=100.0*, *online=None*)
Add signal framing related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

frame_size : int, optional

Size of one frame in samples.

fps : float, optional

Frames per second.

online : bool, optional

Online mode (use only past signal information, i.e. align the window to the left of the reference sample).

Returns argparse argument group

Signal framing argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

6.2.2 madmom.audio.filters

This module contains filter and filterbank related functionality.

madmom.audio.filters.**hz2mel**(*f*)
Convert Hz frequencies to Mel.

Parameters **f** : numpy array

Input frequencies [Hz].

Returns **m** : numpy array

Frequencies in Mel [Mel].

madmom.audio.filters.**mel2hz**(*m*)
Convert Mel frequencies to Hz.

Parameters **m** : numpy array

Input frequencies [Mel].

Returns f: numpy array

Frequencies in Hz [Hz].

madmom.audio.filters.**mel_frequencies**(*num_bands*, *fmin*, *fmax*)
Returns frequencies aligned on the Mel scale.

Parameters **num_bands** : int

Number of bands.

fmin : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

Returns mel_frequencies: numpy array

Frequencies with Mel spacing [Hz].

madmom.audio.filters.**bark_frequencies**(*fmin=20.0*, *fmax=15500.0*)
Returns frequencies aligned on the Bark scale.

Parameters **fmin** : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

Returns **bark_frequencies** : numpy array

Frequencies with Bark spacing [Hz].

madmom.audio.filters.**bark_double_frequencies**(*fmin=20.0*, *fmax=15500.0*)
Returns frequencies aligned on the Bark-scale.

The list also includes center frequencies between the corner frequencies.

Parameters **fmin** : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

Returns **bark_frequencies** : numpy array

Frequencies with Bark spacing [Hz].

madmom.audio.filters.**log_frequencies**(*bands_per_octave*, *fmin*, *fmax*, *fref=440.0*)
Returns frequencies aligned on a logarithmic frequency scale.

Parameters **bands_per_octave** : int

Number of filter bands per octave.

fmin : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

fref : float, optional

Tuning frequency [Hz].

Returns **log_frequencies** : numpy array

Logarithmically spaced frequencies [Hz].

Notes

If *bands_per_octave* = 12 and *fref* = 440 are used, the frequencies are equivalent to MIDI notes.

madmom.audio.filters.**semitone_frequencies**(*fmin*, *fmax*, *fref=440.0*)
Returns frequencies separated by semitones.

Parameters **fmin** : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

fref : float, optional

Tuning frequency of A4 [Hz].

Returns **semitone_frequencies** : numpy array

Semitone frequencies [Hz].

madmom.audio.filters.**hz2midi**(*f*, *fref=440.0*)
Convert frequencies to the corresponding MIDI notes.

Parameters **f** : numpy array

Input frequencies [Hz].

fref : float, optional

Tuning frequency of A4 [Hz].

Returns **m** : numpy array

MIDI notes

Notes

For details see: at http://www.phys.unsw.edu.au/jw/notes.html This function does not necessarily return a valid MIDI Note, you may need to round it to the nearest integer.

madmom.audio.filters.**midi2hz**(*m*, *fref=440.0*)

Convert MIDI notes to corresponding frequencies.

Parameters **m** : numpy array

Input MIDI notes.

fref : float, optional

Tuning frequency of A4 [Hz].

Returns **f** : numpy array

Corresponding frequencies [Hz].

madmom.audio.filters.**midi_frequencies**(*fmin*, *fmax*, *fref=440.0*)

Returns frequencies separated by semitones.

Parameters **fmin** : float

Minimum frequency [Hz].

fmax : float

Maximum frequency [Hz].

fref : float, optional

Tuning frequency of A4 [Hz].

Returns **semitone_frequencies** : numpy array

Semitone frequencies [Hz].

madmom.audio.filters.**hz2erb**(*f*)

Convert Hz to ERB.

Parameters **f** : numpy array

Input frequencies [Hz].

Returns **e** : numpy array

Frequencies in ERB [ERB].

Notes

Information about the ERB scale can be found at: https://ccrma.stanford.edu/~jos/bbt/Equivalent_Rectangular_Bandwidth.html

madmom.audio.filters.**erb2hz**(*e*)
Convert ERB scaled frequencies to Hz.

Parameters **e** : numpy array

Input frequencies [ERB].

Returns **f** : numpy array

Frequencies in Hz [Hz].

Notes

Information about the ERB scale can be found at: https://ccrma.stanford.edu/~jos/bbt/Equivalent_Rectangular_Bandwidth.html

madmom.audio.filters.**frequencies2bins**(*frequencies*, *bin_frequencies*, *unique_bins=False*)
Map frequencies to the closest corresponding bins.

Parameters **frequencies** : numpy array

Input frequencies [Hz].

bin_frequencies : numpy array

Frequencies of the (FFT) bins [Hz].

unique_bins : bool, optional

Return only unique bins, i.e. remove all duplicate bins resulting from insufficient resolution at low frequencies.

Returns **bins** : numpy array

Corresponding (unique) bins.

Notes

It can be important to return only unique bins, otherwise the lower frequency bins can be given too much weight if all bins are simply summed up (as in the spectral flux onset detection).

madmom.audio.filters.**bins2frequencies**(*bins*, *bin_frequencies*)
Convert bins to the corresponding frequencies.

Parameters **bins** : numpy array

Bins (e.g. FFT bins).

bin_frequencies : numpy array

Frequencies of the (FFT) bins [Hz].

Returns **f** : numpy array

Corresponding frequencies [Hz].

class madmom.audio.filters.**Filter**(*data*, *start=0*, *norm=False*)
Generic Filter class.

Parameters **data** : 1D numpy array

Filter data.

start : int, optional

Start position (see notes).

norm : bool, optional

Normalize the filter area to 1.

Notes

The start position is mandatory if a Filter should be used for the creation of a Filterbank.

classmethod band_bins(*bins*, ***kwargs*)
Must yield the center/crossover bins needed for filter creation.

Parameters **bins** : numpy array

Center/crossover bins used for the creation of filters.

kwargs : dict, optional

Additional parameters for for the creation of filters (e.g. if the filters should overlap or not).

classmethod filters(*bins*, *norm*, ***kwargs*)
Create a list with filters for the the given bins.

Parameters **bins** : list or numpy array

Center/crossover bins of the filters.

norm : bool

Normalize the area of the filter(s) to 1.

kwargs : dict, optional

Additional parameters passed to *band_bins()* (e.g. if the filters should overlap or not).

Returns **filters** : list

Filter(s) for the given bins.

class madmom.audio.filters.**TriangularFilter**(*start*, *center*, *stop*, *norm=False*)
Triangular filter class.

Create a triangular shaped filter with length *stop*, height 1 (unless normalized) with indices <= *start* set to 0.

Parameters **start** : int

Start bin of the filter.

center : int

Center bin of the filter.

stop : int

Stop bin of the filter.

norm : bool, optional

Normalize the area of the filter to 1.

classmethod band_bins(*bins*, *overlap=True*)
Yields start, center and stop bins for creation of triangular filters.

Parameters **bins** : list or numpy array

Center bins of filters.

overlap : bool, optional

Filters should overlap (see notes).

Yields **start** : int

Start bin of the filter.

center : int

Center bin of the filter.

stop : int

Stop bin of the filter.

Notes

If *overlap* is 'False', the *start* and *stop* bins of the filters are interpolated between the centre bins, normal rounding applies.

class madmom.audio.filters.**RectangularFilter**(*start*, *stop*, *norm=False*)

Rectangular filter class.

Create a rectangular shaped filter with length *stop*, height 1 (unless normalized) with indices < *start* set to 0.

Parameters **start** : int

Start bin of the filter.

stop : int

Stop bin of the filter.

norm : bool, optional

Normalize the area of the filter to 1.

classmethod **band_bins**(*bins*, *overlap=False*)

Yields start and stop bins and normalization info for creation of rectangular filters.

Parameters **bins** : list or numpy array

Crossover bins of filters.

overlap : bool, optional

Filters should overlap.

Yields **start** : int

Start bin of the filter.

stop : int

Stop bin of the filter.

class madmom.audio.filters.**Filterbank**(*data*, *bin_frequencies*)

Generic filterbank class.

A Filterbank is a simple numpy array enhanced with several additional attributes, e.g. number of bands.

A Filterbank has a shape of (num_bins, num_bands) and can be used to filter a spectrogram of shape (num_frames, num_bins) to (num_frames, num_bands).

Parameters **data** : numpy array, shape (num_bins, num_bands)

Data of the filterbank .

bin_frequencies : numpy array, shape (num_bins,)

Frequencies of the bins [Hz].

Notes

The length of *bin_frequencies* must be equal to the first dimension of the given *data* array.

classmethod **from_filters** (*filters*, *bin_frequencies*)
Create a filterbank with possibly multiple filters per band.

Parameters **filters** : list (of lists) of Filters

List of Filters (per band); if multiple filters per band are desired, they should be also contained in a list, resulting in a list of lists of Filters.

bin_frequencies : numpy array

Frequencies of the bins (needed to determine the expected size of the filterbank).

Returns **filterbank** : *Filterbank* instance

Filterbank with respective filter elements.

num_bins
Number of bins.

num_bands
Number of bands.

corner_frequencies
Corner frequencies of the filter bands.

center_frequencies
Center frequencies of the filter bands.

fmin
Minimum frequency of the filterbank.

fmax
Maximum frequency of the filterbank.

class madmom.audio.filters.**FilterbankProcessor** (*data*, *bin_frequencies*)
Generic filterbank processor class.

A FilterbankProcessor is a simple wrapper for Filterbank which adds a process() method.

See also:

Filterbank

process (*data*)
Filter the given data with the Filterbank.

Parameters **data** : 2D numpy array

Data to be filtered.

Returns

——-

filt_data : numpy array

Filtered data.

Notes

This method makes the *Filterbank* act as a Processor.

static add_arguments(*parser*, *filterbank=None*, *num_bands=None*, *crossover_frequencies=None*, *fmin=None*, *fmax=None*, *norm_filters=None*, *unique_filters=None*)

Add filterbank related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

filterbank : *audio.filters.Filterbank*, optional

Use a filterbank of that type.

num_bands : int, optional

Number of bands (per octave).

crossover_frequencies : list or numpy array, optional

List of crossover frequencies at which the *spectrogram* is split into bands.

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filters of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

Returns argparse argument group

Filterbank argument parser group.

Notes

Parameters are included in the group only if they are not 'None'. Depending on the type of the *filterbank*, either *num_bands* or *crossover_frequencies* should be used.

class madmom.audio.filters.**MelFilterbank**(*bin_frequencies*, *num_bands=40*, *fmin=20.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

Mel filterbank class.

Parameters **bin_frequencies** : numpy array

Frequencies of the bins [Hz].

num_bands : int, optional

Number of filter bands.

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filters to area 1.

unique_filters : bool, optional

Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

Notes

Because of rounding and mapping of frequencies to bins and back to frequencies, the actual minimum, maximum and center frequencies do not necessarily match the parameters given.

class madmom.audio.filters.**BarkFilterbank**(*bin_frequencies*, *num_bands='normal'*, *fmin=20.0*, *fmax=15500.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

Bark filterbank class.

Parameters **bin_frequencies** : numpy array

Frequencies of the bins [Hz].

num_bands : {'normal', 'double'}, optional

Number of filter bands.

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filters to area 1.

unique_filters : bool, optional

Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

class madmom.audio.filters.**LogarithmicFilterbank**(*bin_frequencies*, *num_bands=12*, *fmin=30.0*, *fmax=17000.0*, *fref=440.0*, *norm_filters=True*, *unique_filters=True*, *bands_per_octave=True*)

Logarithmic filterbank class.

Parameters **bin_frequencies** : numpy array

Frequencies of the bins [Hz].

num_bands : int, optional

Number of filter bands (per octave).

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

fref : float, optional

Tuning frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filters to area 1.

unique_filters : bool, optional

Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

bands_per_octave : bool, optional

Indicates whether *num_bands* is given as number of bands per octave ('True', default) or as an absolute number of bands ('False').

Notes

num_bands sets either the number of bands per octave or the total number of bands, depending on the setting of *bands_per_octave*. *num_bands* is used to set also the number of bands per octave to keep the argument for all classes the same. If 12 bands per octave are used, a filterbank with semitone spacing is created.

madmom.audio.filters.**LogFilterbank**
alias of *LogarithmicFilterbank*

class madmom.audio.filters.**RectangularFilterbank**(*bin_frequencies*, *crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*)

Rectangular filterbank class.

Parameters **bin_frequencies** : numpy array

Frequencies of the bins [Hz].

crossover_frequencies : list or numpy array

Crossover frequencies of the bands [Hz].

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filters to area 1.

unique_filters : bool, optional

Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

6.2.3 madmom.audio.comb_filters

This module contains comb-filter and comb-filterbank functionality.

class madmom.audio.comb_filters.**CombFilterbankProcessor**
CombFilterbankProcessor class.

Parameters **filter_function** : filter function or str

Filter function to use {feed_forward_comb_filter, feed_backward_comb_filter} or a string literal {'forward', 'backward'}.

tau : list or numpy array, shape (N,)

Delay length(s) [frames].

alpha : list or numpy array, shape (N,)

Corresponding scaling factor(s).

Notes

tau and *alpha* must have the same length.

process(*self*, *data*)
Process the given data with the comb filter.

Parameters **data** : numpy array

Data to be filtered/processed.

Returns **comb_filtered_data** : numpy array

Comb filtered data with the different taus aligned along the (new) first dimension.

madmom.audio.comb_filters.**comb_filter**(*signal*, *filter_function*, *tau*, *alpha*)
Filter the signal with a bank of either feed forward or backward comb filters.

Parameters **signal** : numpy array

Signal.

filter_function : {feed_forward_comb_filter, feed_backward_comb_filter}

Filter function to use (feed forward or backward).

tau : list or numpy array, shape (N,)

Delay length(s) [frames].

alpha : list or numpy array, shape (N,)

Corresponding scaling factor(s).

Returns **comb_filtered_signal** : numpy array

Comb filtered signal with the different taus aligned along the (new) first dimension.

Notes

tau and *alpha* must be of same length.

madmom.audio.comb_filters.**feed_backward_comb_filter**(*signal*, *tau*, *alpha*)
Filter the signal with a feed backward comb filter.

Parameters **signal** : numpy array

Signal.

tau : int

Delay length.

alpha : float

Scaling factor.

Returns **comb_filtered_signal** : numpy array

Comb filtered signal.

Notes

y[n] = x[n] + α * y[n - τ] is used as a filter function.

madmom.audio.comb_filters.**feed_backward_comb_filter_1d**(*ndarray signal*, *unsigned int tau*, *float alpha*)

Filter the signal with a feed backward comb filter.

Parameters **signal** : 1D numpy array, float dtype

Signal.

tau : int

Delay length.

alpha : float

Scaling factor.

Returns **comb_filtered_signal** : numpy array

Comb filtered signal.

Notes

y[n] = x[n] + α * y[n - τ] is used as a filter function.

madmom.audio.comb_filters.**feed_backward_comb_filter_2d**(*ndarray signal*, *unsigned int tau*, *float alpha*)

Filter the signal with a feed backward comb filter.

Parameters **signal** : 2D numpy array, float dtype

Signal.

tau : int

Delay length.

alpha : float

Scaling factor.

Returns **comb_filtered_signal** : numpy array

Comb filtered signal.

Notes

y[n] = x[n] + α * y[n - τ] is used as a filter function.

madmom.audio.comb_filters.**feed_forward_comb_filter**(*signal*, *tau*, *alpha*)

Filter the signal with a feed forward comb filter.

Parameters **signal** : numpy array

Signal.

tau : int

Delay length.

alpha : float

Scaling factor.

Returns **comb_filtered_signal** : numpy array

Comb filtered signal.

Notes

y[n] = x[n] + α * x[n - τ] is used as a filter function.

6.2.4 madmom.audio.ffmpeg

This module contains audio handling via ffmpeg functionality.

madmom.audio.ffmpeg.**decode_to_disk**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *outfile=None*, *tmp_dir=None*, *tmp_suffix=None*, *cmd='ffmpeg'*)

Decodes the given audio file, optionally down-mixes it to mono and writes it to another file as a sequence of samples. Returns the file name of the output file.

Parameters **infile** : str

Name of the audio sound file to decode.

fmt : {'f32le', 's16le'}, optional

Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate : int, optional

Sample rate to re-sample the signal to (if set) [Hz].

num_channels : int, optional

Number of channels to reduce the signal to.

skip : float, optional

Number of seconds to skip at beginning of file.

max_len : float, optional

Maximum length in seconds to decode.

outfile : str, optional

The file to decode the sound file to; if not given, a temporary file will be created.

tmp_dir : str, optional

The directory to create the temporary file in (if no *outfile* is given).

tmp_suffix : str, optional

The file suffix for the temporary file if no *outfile* is given; e.g. ".pcm" (including the dot).

cmd : {'ffmpeg', 'avconv'}, optional

Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns **outfile** : str

The output file name.

madmom.audio.ffmpeg.**decode_to_memory**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *cmd='ffmpeg'*)

Decodes the given audio file, down-mixes it to mono and returns it as a binary string of a sequence of samples.

Parameters **infile** : str

Name of the audio sound file to decode.

fmt : {'f32le', 's16le'}, optional

Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate : int, optional

Sample rate to re-sample the signal to (if set) [Hz].

num_channels : int, optional

Number of channels to reduce the signal to.

skip : float, optional

Number of seconds to skip at beginning of file.

max_len : float, optional

Maximum length in seconds to decode.

cmd : {'ffmpeg', 'avconv'}, optional

Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns **samples** : str

a binary string of samples

madmom.audio.ffmpeg.**decode_to_pipe**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *buf_size=-1*, *cmd='ffmpeg'*)

Decodes the given audio file, down-mixes it to mono and returns a file-like object for reading the samples, as well as a process object. To stop decoding the file, call close() on the returned file-like object, then call wait() on the returned process object.

Parameters **infile** : str

Name of the audio sound file to decode.

fmt : {'f32le', 's16le'}, optional

Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate : int, optional

Sample rate to re-sample the signal to (if set) [Hz].

num_channels : int, optional

Number of channels to reduce the signal to.

skip : float, optional

Number of seconds to skip at beginning of file.

max_len : float, optional

Maximum length in seconds to decode.

buf_size : int, optional

Size of buffer for the file-like object: - '-1' means OS default (default), - '0' means unbuffered, - '1' means line-buffered, any other value is the buffer size in bytes.

cmd : {'ffmpeg','avconv'}, optional

Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns **pipe** : file-like object

File-like object for reading the decoded samples.

proc : process object

Process object for the decoding process.

madmom.audio.ffmpeg.**get_file_info**(*infile*, *cmd='ffprobe'*)

Extract and return information about audio files.

Parameters **infile** : str

Name of the audio file.

cmd : {'ffprobe', 'avprobe'}, optional

Probing command (defaults to ffprobe, alternatively supports avprobe).

Returns dict

Audio file information.

madmom.audio.ffmpeg.**load_ffmpeg_file**(*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*, *cmd_decode='ffmpeg'*, *cmd_probe='ffprobe'*)

Load the audio data from the given file and return it as a numpy array.

This uses ffmpeg (or avconv) and thus supports a lot of different file formats, resampling and channel conversions. The file will be fully decoded into memory if no start and stop positions are given.

Parameters **filename** : str

Name of the audio sound file to load.

sample_rate : int, optional

Sample rate to re-sample the signal to [Hz]; 'None' returns the signal in its original rate.

num_channels : int, optional

Reduce or expand the signal to *num_channels* channels; 'None' returns the signal with its original channels.

start : float, optional

Start position [seconds].

stop : float, optional

Stop position [seconds].

dtype : numpy dtype, optional

Numpy dtype to return the signal in (supports signed and unsigned 8/16/32-bit integers, and single and double precision floats, each in little or big endian). If 'None', np.int16 is used.

cmd_decode : {'ffmpeg', 'avconv'}, optional

Decoding command (defaults to ffmpeg, alternatively supports avconv).

cmd_probe : {'ffprobe', 'avprobe'}, optional

Probing command (defaults to ffprobe, alternatively supports avprobe).

Returns **signal** : numpy array

Audio samples.

sample_rate : int

Sample rate of the audio samples.

6.2.5 madmom.audio.stft

This module contains Short-Time Fourier Transform (STFT) related functionality.

`madmom.audio.stft.`**`fft_frequencies`**(*num_fft_bins*, *sample_rate*)

Frequencies of the FFT bins.

Parameters **num_fft_bins** : int

Number of FFT bins (i.e. half the FFT length).

sample_rate : float

Sample rate of the signal.

Returns **fft_frequencies** : numpy array

Frequencies of the FFT bins [Hz].

`madmom.audio.stft.`**`stft`**(*frames*, *window*, *fft_size=None*, *circular_shift=False*)

Calculates the complex Short-Time Fourier Transform (STFT) of the given framed signal.

Parameters **frames** : numpy array or iterable, shape (num_frames, frame_size)

Framed signal (e.g. `FramedSignal` instance)

window : numpy array, shape (frame_size,)

Window (function).

fft_size : int, optional

FFT size (should be a power of 2); if 'None', the 'frame_size' given by the *frames* is used; if the given *fft_size* is greater than the 'frame_size', the frames are zero-padded accordingly.

circular_shift : bool, optional

Circular shift the individual frames before performing the FFT; needed for correct phase.

Returns **stft** : numpy array, shape (num_frames, frame_size)

The complex STFT of the framed signal.

madmom.audio.stft.**phase**(*stft*)

Returns the phase of the complex STFT of a signal.

Parameters **stft** : numpy array, shape (num_frames, frame_size)

The complex STFT of a signal.

Returns **phase** : numpy array

Phase of the STFT.

madmom.audio.stft.**local_group_delay**(*phase*)

Returns the local group delay of the phase of a signal.

Parameters **phase** : numpy array, shape (num_frames, frame_size)

Phase of the STFT of a signal.

Returns **lgd** : numpy array

Local group delay of the phase.

madmom.audio.stft.**lgd**(*phase*)

Returns the local group delay of the phase of a signal.

Parameters **phase** : numpy array, shape (num_frames, frame_size)

Phase of the STFT of a signal.

Returns **lgd** : numpy array

Local group delay of the phase.

class madmom.audio.stft.**PropertyMixin**

Mixin which provides *num_frames*, *num_bins* properties to classes.

num_frames

Number of frames.

num_bins

Number of bins.

class madmom.audio.stft.**ShortTimeFourierTransform**(*frames*, *window=<function hanning>*, *fft_size=None*, *circular_shift=False*, ***kwargs*)

ShortTimeFourierTransform class.

Parameters **frames** : *audio.signal.FramedSignal* instance

FramedSignal instance.

window : numpy ufunc or numpy array, optional

Window (function); if a function (e.g. np.hanning) is given, a window of the given shape of size of the *frames* is used.

fft_size : int, optional

FFT size (should be a power of 2); if 'None', the *frame_size* given by the *frames* is used, if the given *fft_size* is greater than the *frame_size*, the frames are zero-padded accordingly.

circular_shift : bool, optional

Circular shift the individual frames before performing the FFT; needed for correct phase.

kwargs : dict, optional

If no *audio.signal.FramedSignal* instance was given, one is instantiated with these additional keyword arguments.

Notes

If the `Signal` (wrapped in the `FramedSignal`) has an integer dtype, it is automatically scaled as if it has a float dtype with the values being in the range [-1, 1]. This results in same valued STFTs independently of the dtype of the signal. On the other hand, this prevents extra memory consumption since the data-type of the signal does not need to be converted (and if no decoding is needed, the audio signal can be memory mapped).

spec(***kwargs*)

Returns the magnitude spectrogram of the STFT.

Parameters **kwargs** : dict, optional

Keyword arguments passed to *audio.spectrogram.Spectrogram*.

Returns **spec** : *audio.spectrogram.Spectrogram*

audio.spectrogram.Spectrogram instance.

phase(***kwargs*)

Returns the phase of the STFT.

Parameters **kwargs** : dict, optional

keyword arguments passed to *Phase*.

Returns **phase** : *Phase*

Phase instance.

madmom.audio.stft.**STFT**

alias of *ShortTimeFourierTransform*

class madmom.audio.stft.**ShortTimeFourierTransformProcessor**(*window=<function hanning>*, *fft_size=None*, *circular_shift=False*, ***kwargs*)

ShortTimeFourierTransformProcessor class.

Parameters **window** : numpy ufunc, optional

Window function.

fft_size : int, optional

FFT size (should be a power of 2); if 'None', it is determined by the size of the frames; if is greater than the frame size, the frames are zero-padded accordingly.

circular_shift : bool, optional

Circular shift the individual frames before performing the FFT; needed for correct phase.

process (*data*, ***kwargs*)
Perform FFT on a framed signal and return the STFT.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict, optional

Keyword arguments passed to *ShortTimeFourierTransform*.

Returns **stft** : *ShortTimeFourierTransform*

ShortTimeFourierTransform instance.

static add_arguments (*parser*, *window=None*, *fft_size=None*)
Add STFT related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser.

window : numpy ufunc, optional

Window function.

fft_size : int, optional

Use this size for FFT (should be a power of 2).

Returns argpase argument group

STFT argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

madmom.audio.stft.**STFTProcessor**
alias of *ShortTimeFourierTransformProcessor*

class madmom.audio.stft.**Phase** (*stft*, ***kwargs*)
Phase class.

Parameters **stft** : *ShortTimeFourierTransform* instance

ShortTimeFourierTransform instance.

kwargs : dict, optional

If no *ShortTimeFourierTransform* instance was given, one is instantiated with these additional keyword arguments.

local_group_delay (***kwargs*)
Returns the local group delay of the phase.

Parameters **kwargs** : dict, optional

Keyword arguments passed to *LocalGroupDelay*.

Returns **lgd** : *LocalGroupDelay* instance

LocalGroupDelay instance.

lgd(***kwargs*)
Returns the local group delay of the phase.

Parameters **kwargs** : dict, optional

Keyword arguments passed to *LocalGroupDelay*.

Returns **lgd** : *LocalGroupDelay* instance

LocalGroupDelay instance.

class madmom.audio.stft.**LocalGroupDelay**(*phase*, ***kwargs*)
Local Group Delay class.

Parameters **stft** : *Phase* instance

Phase instance.

kwargs : dict, optional

If no *Phase* instance was given, one is instantiated with these additional keyword arguments.

madmom.audio.stft.**LGD**
alias of *LocalGroupDelay*

6.2.6 madmom.audio.spectrogram

This module contains spectrogram related functionality.

madmom.audio.spectrogram.**spec**(*stft*)
Computes the magnitudes of the complex Short Time Fourier Transform of a signal.

Parameters **stft** : numpy array

Complex STFT of a signal.

Returns **spec** : numpy array

Magnitude spectrogram.

class madmom.audio.spectrogram.**Spectrogram**(*stft*, ***kwargs*)
A *Spectrogram* represents the magnitude spectrogram of a *audio.stft.ShortTimeFourierTransform*.

Parameters **stft** : *audio.stft.ShortTimeFourierTransform* instance

Short Time Fourier Transform.

kwargs : dict, optional

If no *audio.stft.ShortTimeFourierTransform* instance was given, one is instantiated with these additional keyword arguments.

Attributes

stft	(*audio.stft.ShortTimeFourierTransform* instance) Underlying ShortTimeFourierTransform instance.
frames	(*audio.signal.FramedSignal* instance) Underlying FramedSignal instance.

diff(***kwargs*)
Return the difference of the magnitude spectrogram.

Parameters **kwargs** : dict

Keyword arguments passed to *SpectrogramDifference*.

Returns **diff** : *SpectrogramDifference* instance

The differences of the magnitude spectrogram.

filter(***kwargs*)
Return a filtered version of the magnitude spectrogram.

Parameters **kwargs** : dict

Keyword arguments passed to *FilteredSpectrogram*.

Returns **filt_spec** : *FilteredSpectrogram* instance

Filtered version of the magnitude spectrogram.

log(***kwargs*)
Return a logarithmically scaled version of the magnitude spectrogram.

Parameters **kwargs** : dict

Keyword arguments passed to *LogarithmicSpectrogram*.

Returns **log_spec** : *LogarithmicSpectrogram* instance

Logarithmically scaled version of the magnitude spectrogram.

class madmom.audio.spectrogram.**SpectrogramProcessor**(***kwargs*)
SpectrogramProcessor class.

process(*data*, ***kwargs*)
Create a Spectrogram from the given data.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *Spectrogram*.

Returns **spec** : *Spectrogram* instance

Spectrogram.

class madmom.audio.spectrogram.**FilteredSpectrogram**(*spectrogram*, *filterbank=<class 'madmom.audio.filters.LogarithmicFilterbank'>*, *num_bands=12*, *fmin=30.0*, *fmax=17000.0*, *fref=440.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)
FilteredSpectrogram class.

Parameters **spectrogram** : *Spectrogram* instance

Spectrogram.

filterbank : *audio.filters.Filterbank*, optional

Filterbank class or instance; if a class is given (rather than an instance), one will be created with the given type and parameters.

num_bands : int, optional

Number of filter bands (per octave, depending on the type of the *filterbank*).

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

fref : float, optional

Tuning frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filter bands of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

kwargs : dict, optional

If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

class madmom.audio.spectrogram.**FilteredSpectrogramProcessor**(*filterbank=<class 'madmom.audio.filters.LogarithmicFilterbank'>*, *num_bands=12*, *fmin=30.0*, *fmax=17000.0*, *fref=440.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

FilteredSpectrogramProcessor class.

Parameters **filterbank** : *audio.filters.Filterbank*

Filterbank used to filter a spectrogram.

num_bands : int

Number of bands (per octave).

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

fref : float, optional

Tuning frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filter of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

process(*data*, ***kwargs*)

Create a FilteredSpectrogram from the given data.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *FilteredSpectrogram*.

Returns **filt_spec** : *FilteredSpectrogram* instance

Filtered spectrogram.

class madmom.audio.spectrogram.**LogarithmicSpectrogram**(*spectrogram*, *mul=1.0*, *add=1.0*, ***kwargs*)

LogarithmicSpectrogram class.

Parameters **spectrogram** : *Spectrogram* instance

Spectrogram.

mul : float, optional

Multiply the magnitude spectrogram with this factor before taking the logarithm.

add : float, optional

Add this value before taking the logarithm of the magnitudes.

kwargs : dict, optional

If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

class madmom.audio.spectrogram.**LogarithmicSpectrogramProcessor**(*mul=1.0*, *add=1.0*, ***kwargs*)

Logarithmic Spectrogram Processor class.

Parameters **mul** : float, optional

Multiply the magnitude spectrogram with this factor before taking the logarithm.

add : float, optional

Add this value before taking the logarithm of the magnitudes.

process(*data*, ***kwargs*)

Perform logarithmic scaling of a spectrogram.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *LogarithmicSpectrogram*.

Returns **log_spec** : *LogarithmicSpectrogram* instance

Logarithmically scaled spectrogram.

static add_arguments(*parser*, *log=None*, *mul=None*, *add=None*)

Add spectrogram scaling related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

log : bool, optional

Take the logarithm of the spectrogram.

mul : float, optional

Multiply the magnitude spectrogram with this factor before taking the logarithm.

add : float, optional

Add this value before taking the logarithm of the magnitudes.

Returns argparse argument group

Spectrogram scaling argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

class madmom.audio.spectrogram.**LogarithmicFilteredSpectrogram**(*spectrogram*, ***kwargs*)

LogarithmicFilteredSpectrogram class.

Parameters **spectrogram** : *FilteredSpectrogram* instance

Filtered spectrogram.

kwargs : dict, optional

If no *FilteredSpectrogram* instance was given, one is instantiated with these additional keyword arguments and logarithmically scaled afterwards, i.e. passed to *LogarithmicSpectrogram*.

See also:

FilteredSpectrogram, *LogarithmicSpectrogram*

Notes

For the filtering and scaling parameters, please refer to *FilteredSpectrogram* and *LogarithmicSpectrogram*.

class madmom.audio.spectrogram.**LogarithmicFilteredSpectrogramProcessor**(*filterbank=<class 'madmom.audio.filters.LogarithmicFi*, *num_bands=12*, *fmin=30.0*, *fmax=17000.0*, *fref=440.0*, *norm_filters=True*, *unique_filters=True*, *mul=1.0*, *add=1.0*, ***kwargs*)

Logarithmic Filtered Spectrogram Processor class.

Parameters **filterbank** : *audio.filters.Filterbank*

Filterbank used to filter a spectrogram.

num_bands : int

Number of bands (per octave).

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

fref : float, optional

Tuning frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filter of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

mul : float, optional

Multiply the magnitude spectrogram with this factor before taking the logarithm.

add : float, optional

Add this value before taking the logarithm of the magnitudes.

process(*data*, ***kwargs*)

Perform filtering and logarithmic scaling of a spectrogram.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *LogarithmicFilteredSpectrogram*.

Returns **log_filt_spec** : *LogarithmicFilteredSpectrogram* instance

Logarithmically scaled filtered spectrogram.

class madmom.audio.spectrogram.**SpectrogramDifference**(*spectrogram*, *diff_ratio=0.5*, *diff_frames=None*, *diff_max_bins=None*, *positive_diffs=False*, ***kwargs*)

SpectrogramDifference class.

Parameters **spectrogram** : *Spectrogram* instance

Spectrogram.

diff_ratio : float, optional

Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames : int, optional

Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins : int, optional

Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs : bool, optional

Keep only the positive differences, i.e. set all diff values < 0 to 0.

kwargs : dict, optional

If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

Notes

The SuperFlux algorithm *[R1]* uses a maximum filtered spectrogram with 3 *diff_max_bins* together with a 24 band logarithmic filterbank to calculate the difference spectrogram with a *diff_ratio* of 0.5.

The effect of this maximum filter applied to the spectrogram is that the magnitudes are "widened" in frequency direction, i.e. the following difference calculation is less sensitive against frequency fluctuations. This effect is exploitet to suppress false positive energy fragments for onsets detection originating from vibrato.

References

[R1]

positive_diff()
Positive diff.

class madmom.audio.spectrogram.**SpectrogramDifferenceProcessor**(*diff_ratio=0.5*, *diff_frames=None*, *diff_max_bins=None*, *positive_diffs=False*, *stack_diffs=None*, ***kwargs*)

Difference Spectrogram Processor class.

Parameters **diff_ratio** : float, optional

Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames : int, optional

Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins : int, optional

Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs : bool, optional

Keep only the positive differences, i.e. set all diff values < 0 to 0.

stack_diffs : numpy stacking function, optional

If 'None', only the differences are returned. If set, the diffs are stacked with the underlying spectrogram data according to the *stack* function:

- `np.vstack` the differences and spectrogram are stacked vertically, i.e. in time direction,
- `np.hstack` the differences and spectrogram are stacked horizontally, i.e. in frequency direction,

- `np.dstack` the differences and spectrogram are stacked in depth, i.e. return them as a 3D representation with depth as the third dimension.

process(*data*, ***kwargs*)

Perform a temporal difference calculation on the given data.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *SpectrogramDifference*.

Returns **diff** : *SpectrogramDifference* instance

Spectrogram difference.

static add_arguments(*parser*, *diff=None*, *diff_ratio=None*, *diff_frames=None*, *diff_max_bins=None*, *positive_diffs=None*)

Add spectrogram difference related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

diff : bool, optional

Take the difference of the spectrogram.

diff_ratio : float, optional

Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames : int, optional

Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins : int, optional

Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs : bool, optional

Keep only the positive differences, i.e. set all diff values < 0 to 0.

Returns argparse argument group

Spectrogram difference argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

Only the *diff_frames* parameter behaves differently, it is included if either the *diff_ratio* is set or a value != 'None' is given.

class madmom.audio.spectrogram.**SuperFluxProcessor**(***kwargs*)

Spectrogram processor which sets the default values suitable for the SuperFlux algorithm.

class madmom.audio.spectrogram.**MultiBandSpectrogram**(*spectrogram*, *crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

MultiBandSpectrogram class.

Parameters **spectrogram** : *Spectrogram* instance

Spectrogram.

crossover_frequencies : list or numpy array

List of crossover frequencies at which the *spectrogram* is split into multiple bands.

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filter bands of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

kwargs : dict, optional

If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

Notes

The MultiBandSpectrogram is implemented as a *Spectrogram* which uses a *audio.filters.RectangularFilterbank* to combine multiple frequency bins.

class madmom.audio.spectrogram.**MultiBandSpectrogramProcessor**(*crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

Spectrogram processor which combines the spectrogram magnitudes into multiple bands.

Parameters **crossover_frequencies** : list or numpy array

List of crossover frequencies at which a spectrogram is split into the individual bands.

fmin : float, optional

Minimum frequency of the filterbank [Hz].

fmax : float, optional

Maximum frequency of the filterbank [Hz].

norm_filters : bool, optional

Normalize the filter bands of the filterbank to area 1.

unique_filters : bool, optional

Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

process(*data*, ***kwargs*)

Return the a multi-band representation of the given data.

Parameters **data** : numpy array

Data to be processed.

kwargs : dict

Keyword arguments passed to *MultiBandSpectrogram*.

Returns **multi_band_spec** : *MultiBandSpectrogram* instance

Spectrogram split into multiple bands.

class madmom.audio.spectrogram.**StackedSpectrogramProcessor**

Deprecated in v0.13, will be removed in v0.14.

Functionality added to *SpectrogramDifferenceProcessor* as *stack_diffs* argument.

CHAPTER 7

madmom.features

This package includes high-level features. Your definition of "high" may vary, but we define high-level features as the ones you want to evaluate (e.g. onsets, beats, etc.). All lower-level features can be found the *madmom.audio* package.

7.1 Notes

All features should be implemented as classes which inherit from Processor (or provide a XYZProcessor(Processor) variant). This way, multiple Processor objects can be chained/combined to achieve the wanted functionality.

class madmom.features.**Activations**(*data*, *fps=None*, *sep=None*, *dtype=<type 'numpy.float32'>*)

The Activations class extends a numpy ndarray with a frame rate (fps) attribute.

Parameters **data** : str, file handle or numpy array

Either file name/handle to read the data from or array.

fps : float, optional

Frames per second (must be set if *data* is given as an array).

sep : str, optional

Separator between activation values (if read from file).

dtype : numpy dtype

Data-type the activations are stored/saved/kept.

Notes

If a filename or file handle is given, an undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

Attributes

fps	(float) Frames per second.

classmethod **load**(*infile*, *fps=None*, *sep=None*)

Load the activations from a file.

Parameters **infile** : str or file handle

Input file name or file handle.

fps : float, optional

Frames per second; if set, it overwrites the saved frame rate.

sep : str, optional

Separator between activation values.

Returns *Activations* instance

Activations instance.

Notes

An undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

save(*outfile*, *sep=None*, *fmt='%.5f'*)

Save the activations to a file.

Parameters **outfile** : str or file handle

Output file name or file handle.

sep : str, optional

Separator between activation values if saved as text file.

fmt : str, optional

Format of the values if saved as text file.

Notes

An undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

If the activations are a 1D array, its values are interpreted as features of a single time step, i.e. all values are printed in a single line. If you want each value to appear in an individual line, use 'n' as a separator.

If the activations are a 2D array, the first axis corresponds to the time dimension, i.e. the features are separated by *sep* and the time steps are printed in separate lines. If you like to swap the dimensions, please use the *T* attribute.

class madmom.features.**ActivationsProcessor**(*mode*, *fps=None*, *sep=None*, ***kwargs*)

ActivationsProcessor processes a file and returns an Activations instance.

Parameters **mode** : {'r', 'w', 'in', 'out', 'load', 'save'}

Mode of the Processor: read/write.

fps : float, optional

Frame rate of the activations (if set, it overwrites the saved frame rate).

sep : str, optional

Separator between activation values if saved as text file.

Notes

An undefined or empty ("") separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations.

process(*data*, *output=None*)

Depending on the mode, either loads the data stored in the given file and returns it as an Activations instance or save the data to the given output.

Parameters **data** : str, file handle or numpy array

Data or file to be loaded (if *mode* is 'r') or data to be saved to file (if *mode* is 'w').

output : str or file handle, optional

output file (only in write-mode)

Returns *Activations* instance

Activations instance (only in read-mode)

static add_arguments(*parser*)

Add options to save/load activations to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser.

Returns **parser_group** : argparse argument group

Input/output argument parser group.

7.2 Submodules

7.2.1 madmom.features.beats

This module contains beat tracking related functionality.

class madmom.features.beats.**MultiModelSelectionProcessor**(*num_ref_predictions*, ***kwargs*)

Processor for selecting the most suitable model (i.e. the predictions thereof) from a multiple models/predictions.

Parameters **num_ref_predictions** : int

Number of reference predictions (see below).

Notes

This processor selects the most suitable prediction from multiple models by comparing them to the predictions of a reference model. The one with the smallest mean squared error is chosen.

If *num_ref_predictions* is 0 or None, an averaged prediction is computed from the given predictions and used as reference.

References

[R13]

process(*predictions*)

Selects the most appropriate predictions form the list of predictions.

Parameters **predictions** : list

Predictions (beat activation functions) of multiple models.

Returns numpy array

Most suitable prediction.

Notes

The reference beat activation function must be the first one in the list of given predictions.

madmom.features.beats.**detect_beats**(*activations*, *interval*, *look_aside=0.2*)

Detects the beats in the given activation function as in *[R14]*.

Parameters **activations** : numpy array

Beat activations.

interval : int

Look for the next beat each *interval* frames.

look_aside : float

Look this fraction of the *interval* to each side to detect the beats.

Returns numpy array

Beat positions [frames].

Notes

A Hamming window of 2 * *look_aside* * *interval* is applied around the position where the beat is expected to prefer beats closer to the centre.

References

[R14]

class madmom.features.beats.**BeatTrackingProcessor**(*look_aside=0.2*, *look_ahead=10*, *fps=None*, ***kwargs*)

Track the beats according to previously determined (local) tempo by iteratively aligning them around the estimated position *[R15]*.

Parameters **look_aside** : float, optional

Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

look_ahead : float, optional

Look *look_ahead* seconds in both directions to determine the local tempo and align the beats accordingly.

fps : float, optional

Frames per second.

Notes

If *look_ahead* is not set, a constant tempo throughout the whole piece is assumed. If *look_ahead* is set, the local tempo (in a range +/- *look_ahead* seconds around the actual position) is estimated and then the next beat is tracked accordingly. This procedure is repeated from the new position to the end of the piece.

Instead of the auto-correlation based method for tempo estimation proposed in *[R15]*, it uses a comb filter based method *[R16]* per default. The behaviour can be controlled with the *tempo_method* parameter.

References

[R15], *[R16]*

process(*activations*)

Detect the beats in the given activation function.

Parameters **activations** : numpy array

Beat activation function.

Returns

——-

beats : numpy array

Detected beat positions [seconds].

static add_arguments(*parser*, *look_aside=0.2*, *look_ahead=10*)

Add beat tracking related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

look_aside : float, optional

Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

look_ahead : float, optional

Look *look_ahead* seconds in both directions to determine the local tempo and align the beats accordingly.

Returns **parser_group** : argparse argument group

Beat tracking argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

class madmom.features.beats.**BeatDetectionProcessor**(*look_aside=0.2*, *fps=None*, ***kwargs*)

Class for detecting beats according to the previously determined global tempo by iteratively aligning them around the estimated position *[R18]*.

Parameters **look_aside** : float

Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

fps : float, optional

Frames per second.

See also:

BeatTrackingProcessor

Notes

A constant tempo throughout the whole piece is assumed.

Instead of the auto-correlation based method for tempo estimation proposed in *[R18]*, it uses a comb filter based method *[R19]* per default. The behaviour can be controlled with the *tempo_method* parameter.

References

[R18], *[R19]*

class madmom.features.beats.**CRFBeatDetectionProcessor**(*interval_sigma=0.18*, *use_factors=False*, *num_intervals=5*, *factors=array([0.5, 0.67, 1., 1.5, 2.])*, ***kwargs*)

Conditional Random Field Beat Detection.

Tracks the beats according to the previously determined global tempo using a conditional random field (CRF) model.

Parameters **interval_sigma** : float, optional

Allowed deviation from the dominant beat interval per beat.

use_factors : bool, optional

Use dominant interval multiplied by factors instead of intervals estimated by tempo estimator.

num_intervals : int, optional

Maximum number of estimated intervals to try.

factors : list or numpy array, optional

Factors of the dominant interval to try.

References

[R21]

process(*activations*)

Detect the beats in the given activation function.

Parameters **activations** : numpy array

Beat activation function.

Returns numpy array

Detected beat positions [seconds].

static **add_arguments**(*parser*, *interval_sigma=0.18*, *use_factors=False*, *num_intervals=5*, *factors=array([0.5, 0.67, 1., 1.5, 2.])*)

Add CRFBeatDetection related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

interval_sigma : float, optional

allowed deviation from the dominant beat interval per beat

use_factors : bool, optional

use dominant interval multiplied by factors instead of intervals estimated by tempo estimator

num_intervals : int, optional

max number of estimated intervals to try

factors : list or numpy array, optional

factors of the dominant interval to try

Returns **parser_group** : argparse argument group

CRF beat tracking argument parser group.

class madmom.features.beats.**DBNBeatTrackingProcessor**(*min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=None*, *transition_lambda=100*, *observation_lambda=16*, *correct=True*, *fps=None*, ***kwargs*)

Beat tracking with RNNs and a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters **min_bpm** : float, optional

Minimum tempo used for beat tracking [bpm].

max_bpm : float, optional

Maximum tempo used for beat tracking [bpm].

num_tempi : int, optional

Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing.

transition_lambda : float, optional

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one).

observation_lambda : int, optional

Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

correct : bool, optional

Correct the beats (i.e. align them to the nearest peak of the beat activation function).

fps : float, optional

Frames per second.

Notes

Instead of the originally proposed state space and transition model for the DBN *[R22]*, the more efficient version proposed in *[R23]* is used.

References

[R22], *[R23]*

process(*activations*)
Detect the beats in the given activation function.

Parameters **activations** : numpy array

Beat activation function.

Returns **beats** : numpy array

Detected beat positions [seconds].

static add_arguments(*parser*, *min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=None*, *transition_lambda=100*, *observation_lambda=16*, *correct=True*)
Add DBN related arguments to an existing parser object.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

min_bpm : float, optional

Minimum tempo used for beat tracking [bpm].

max_bpm : float, optional

Maximum tempo used for beat tracking [bpm].

num_tempi : int, optional

Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing.

transition_lambda : float, optional

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo over a tempo change from one beat to the next one).

observation_lambda : int, optional

Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

correct : bool, optional

Correct the beats (i.e. align them to the nearest peak of the beat activation function).

Returns **parser_group** : argparse argument group

DBN beat tracking argument parser group

class madmom.features.beats.**DownbeatTrackingProcessor**
Renamed to *PatternTrackingProcessor* in v0.13. Will be removed in v0.14.

class madmom.features.beats.**PatternTrackingProcessor**(*pattern_files, min_bpm=[55, 60], max_bpm=[205, 225], num_tempi=[None, None], transition_lambda=[100, 100], downbeats=False, fps=None, **kwargs*)

Pattern tracking with a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters **pattern_files** : list

List of files with the patterns (including the fitted GMMs and information about the number of beats).

min_bpm : list, optional

Minimum tempi used for pattern tracking [bpm].

max_bpm : list, optional

Maximum tempi used for pattern tracking [bpm].

num_tempi : int or list, optional

Number of tempi to model; if set, limit the number of tempi and use a log spacings, otherwise a linear spacings.

transition_lambda : float or list, optional

Lambdas for the exponential tempo change distributions (higher values prefer constant tempi from one beat to the next .one)

downbeats : bool, optional

Report only the downbeats instead of the beats and the respective position inside the bar.

fps : float, optional

Frames per second.

Notes

min_bpm, *max_bpm*, *num_tempo_states*, and *transition_lambda* must contain as many items as rhythmic patterns are modeled (i.e. length of *pattern_files*). If a single value is given for *num_tempo_states* and *transition_lambda*, this value is used for all rhythmic patterns.

Instead of the originally proposed state space and transition model for the DBN *[R24]*, the more efficient version proposed in *[R25]* is used.

References

[R24], *[R25]*

process(*activations*)

Detect the beats based on the given activations.

Parameters **activations** : numpy array

Activations (i.e. multi-band spectral features).

Returns **beats** : numpy array

Detected beat positions [seconds].

static add_arguments(*parser, pattern_files=None, min_bpm=[55, 60], max_bpm=[205, 225], num_tempi=[None, None], transition_lambda=[100, 100]*)

Add DBN related arguments for pattern tracking to an existing parser object.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

pattern_files : list

Load the patterns from these files.

min_bpm : list, optional

Minimum tempi used for beat tracking [bpm].

max_bpm : list, optional

Maximum tempi used for beat tracking [bpm].

num_tempi : int or list, optional

Number of tempi to model; if set, limit the number of states and use log spacings, otherwise a linear spacings.

transition_lambda : float or list, optional

Lambdas for the exponential tempo change distribution (higher values prefer constant tempi from one beat to the next one).

Returns **parser_group** : argparse argument group

Pattern tracking argument parser group

Notes

pattern_files, *min_bpm*, *max_bpm*, *num_tempi*, and *transition_lambda* must the same number of items.

7.2.2 madmom.features.beats_crf

This module contains the speed crucial Viterbi functionality for the CRFBeatDetector plus some functions computing the distributions and normalisation factors.

References

madmom.features.beats_crf.**best_sequence**(*activations*, *interval*, *interval_sigma*)

Extract the best beat sequence for a piece with the Viterbi algorithm.

Parameters **activations** : numpy array

Beat activation function of the piece.

interval : int

Beat interval of the piece.

interval_sigma : float

Allowed deviation from the interval per beat.

Returns **beat_pos** : numpy array

Extracted beat positions [frame indices].

log_prob : float

Log probability of the beat sequence.

madmom.features.beats_crf.**initial_distribution**(*num_states*, *interval*)

Compute the initial distribution.

Parameters **num_states** : int

Number of states in the model.

interval : int

Beat interval of the piece [frames].

Returns numpy array

Initial distribution of the model.

madmom.features.beats_crf.**normalisation_factors**(*activations*, *transition_distribution*)

Compute normalisation factors for model.

Parameters **activations** : numpy array

Beat activation function of the piece.

transition_distribution : numpy array

Transition distribution of the model.

Returns numpy array

Normalisation factors for model.

madmom.features.beats_crf.**transition_distribution**(*interval*, *interval_sigma*)

Compute the transition distribution between beats.

Parameters **interval** : int

Interval of the piece [frames].

interval_sigma : float

Allowed deviation from the interval per beat.

Returns numpy array

Transition distribution between beats.

madmom.features.beats_crf.**viterbi**(*__Pyx_memviewslice pi*, *__Pyx_memviewslice transition*, *__Pyx_memviewslice norm_factor*, *__Pyx_memviewslice activations*, *int tau*)

Viterbi algorithm to compute the most likely beat sequence from the given activations and the dominant interval.

Parameters **pi** : numpy array

Initial distribution.

transition : numpy array

Transition distribution.

norm_factor : numpy array

Normalisation factors.

activations : numpy array

Beat activations.

tau : int

Dominant interval [frames].

Returns **beat_pos** : numpy array

Extracted beat positions [frame indices].

log_prob : float

Log probability of the beat sequence.

7.2.3 madmom.features.beats_hmm

This module contains HMM state spaces, transition and observation models used for beat and downbeat tracking.

Notes

Please note that (almost) everything within this module is discretised to integer values because of performance reasons.

class madmom.features.beats_hmm.**BeatStateSpace**(*min_interval*, *max_interval*, *num_intervals=None*)

State space for beat tracking with a HMM.

Parameters **min_interval** : float

Minimum interval to model.

max_interval : float

Maximum interval to model.

num_intervals : int, optional

Number of intervals to model; if set, limit the number of intervals and use a log spacing instead of the default linear spacing.

References

[R27]

Attributes

num_states	(int) Number of states.
intervals	(numpy array) Modeled intervals.
num_intervals	(int) Number of intervals.
state_positions	(numpy array) Positions of the states.
state_intervals	(numpy array) Intervals of the states.
first_states	(numpy array) First states for each interval.
last_states	(numpy array) Last states for each interval.

class madmom.features.beats_hmm.**BarStateSpace**(*num_beats*, *min_interval*, *max_interval*, *num_intervals=None*)

State space for bar tracking with a HMM.

Parameters **num_beats** : int

Number of beats per bar.

min_interval : float

Minimum beat interval to model.

max_interval : float

Maximum beat interval to model.

num_intervals : int, optional

Number of beat intervals to model; if set, limit the number of intervals and use a log spacing instead of the default linear spacing.

References

[R28]

Attributes

num_beats	(int) Number of beats.
num_states	(int) Number of states.
num_intervals	(int) Number of intervals.
state_positions	(numpy array) Positions of the states.
state_intervals	(numpy array) Intervals of the states.
first_states	(list) First interval states for each beat.
last_states	(list) Last interval states for each beat.

class madmom.features.beats_hmm.**MultiPatternStateSpace**(*state_spaces*)

State space for rhythmic pattern tracking with a HMM.

Parameters **state_spaces** : list

List with state spaces to model.

References

[R29]

madmom.features.beats_hmm.**exponential_transition**(*from_intervals*, *to_intervals*, *transition_lambda*, *threshold=2.2204460492503131e-16*, *norm=True*)

Exponential tempo transition.

Parameters **from_intervals** : numpy array

Intervals where the transitions originate from.

to_intervals

Intervals where the transitions destinate to.

transition_lambda : float

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat/bar to the next one). If None, allow only transitions from/to the same interval.

threshold : float, optional

Set transition probabilities below this threshold to zero.

norm : bool, optional

Normalize the emission probabilities to sum 1.

Returns **probabilities** : numpy array, shape (num_from_intervals, num_to_intervals)

Probability of each transition from an interval to another.

References

[R30]

class madmom.features.beats_hmm.**BeatTransitionModel**(*state_space*, *transition_lambda*)
Transition model for beat tracking with a HMM.

Within the beat the tempo stays the same; at beat boundaries transitions from one tempo (i.e. interval) to another following an exponential distribution are allowed.

Parameters **state_space** : *BeatStateSpace* instance

BeatStateSpace instance.

transition_lambda : float

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one).

References

[R31]

class madmom.features.beats_hmm.**BarTransitionModel**(*state_space*, *transition_lambda*)
Transition model for bar tracking with a HMM.

Within the beats of the bar the tempo stays the same; at beat boundaries transitions from one tempo (i.e. interval) to another following an exponential distribution are allowed.

Parameters **state_space** : *BarStateSpace* instance

BarStateSpace instance.

transition_lambda : float or list

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one). None can be used to set the tempo change probability to 0. If a list is given, the individual values represent the lambdas for each transition into the beat at this index position.

Notes

Bars performing tempo changes only at bar boundaries (and not at the beat boundaries) must have set all but the first *transition_lambda* values to None, e.g. [100, None, None] for a bar with 3 beats.

References

[R32]

class madmom.features.beats_hmm.**MultiPatternTransitionModel**(*transition_models*, *transition_prob=None*, *transition_lambda=None*)

Transition model for pattern tracking with a HMM.

Parameters **transition_models** : list

List with `TransitionModel` instances.

transition_prob : numpy array, optional

Matrix with transition probabilities from one pattern to another.

transition_lambda : float, optional

Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one pattern to the next one).

Notes

Right now, no transitions from one pattern to another are allowed.

class madmom.features.beats_hmm.**RNNBeatTrackingObservationModel**(*state_space*, *observation_lambda*)

Observation model for beat tracking with a HMM.

Parameters **state_space** : *BeatStateSpace* instance

BeatStateSpace instance.

observation_lambda : int

Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

References

[R33]

log_densities(*observations*)

Computes the log densities of the observations.

Parameters **observations** : numpy array

Observations (i.e. activations of the RNN).

Returns numpy array

Log densities of the observations.

class madmom.features.beats_hmm.**GMMPatternTrackingObservationModel**(*pattern_files*, *state_space*)

Observation model for GMM based beat tracking with a HMM.

Parameters **pattern_files** : list

List with files representing the rhythmic patterns, one entry per pattern; each pattern being a list with fitted GMMs.

state_space : `MultiPatternStateSpeac` instance

Multi pattern state space.

References

[R34]

log_densities(*observations*)

Computes the log densities of the observations using (a) GMM(s).

Parameters **observations** : numpy array

Observations (i.e. multiband spectral flux features).

Returns numpy array

Log densities of the observations.

7.2.4 madmom.features.notes

This module contains note transcription related functionality.

madmom.features.notes.**load_notes**(**args*, ***kwargs*)

Load the notes from a file.

Parameters **filename** : str or file handle

Input file to load the notes from.

Returns numpy array

Notes.

Notes

The file format must be (duration and velocity being optional):

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

with one note per line and individual fields separated by whitespace.

madmom.features.notes.**expand_notes**(*notes*, *duration=0.6*, *velocity=100*)

Expand the notes to include all columns.

Parameters **notes** : numpy array, shape (num_notes, 2)

Notes, one per row (column definition see notes).

duration : float, optional

Note duration if not defined by *notes*.

velocity : int, optional

Note velocity if not defined by *notes*.

Returns numpy array

Notes (including note duration and velocity).

Notes

The note columns format must be (duration and velocity being optional):

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

madmom.features.notes.**write_notes**(*notes*, *filename*, *sep='\t'*, *fmt=None*, *header=''*)

Write the notes to a file (as many columns as given).

Parameters **notes** : numpy array, shape (num_notes, 2)

Notes, one per row (column definition see notes).

filename : str or file handle

Output filename or handle.

sep : str, optional

Separator for the fields.

fmt : list, optional

Format of the fields (i.e. columns, see notes)

header : str, optional

Header to be written (as a comment).

Returns numpy array

Notes.

Notes

The note columns format must be (duration and velocity being optional):

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

madmom.features.notes.**write_midi**(*notes*, *filename*, *duration=0.6*, *velocity=100*)

Write the notes to a MIDI file.

Parameters **notes** : numpy array, shape (num_notes, 2)

Notes, one per row (column definition see notes).

filename : str

Output MIDI file.

duration : float, optional

Note duration if not defined by *notes*.

velocity : int, optional

Note velocity if not defined by *notes*.

Returns numpy array

Notes (including note length and velocity).

Notes

The note columns format must be (duration and velocity being optional):

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

madmom.features.notes.**write_mirex_format**(*notes*, *filename*, *duration=0.6*)

Write the frequencies of the notes to file (in MIREX format).

Parameters **notes** : numpy array, shape (num_notes, 2)

Notes, one per row (column definition see notes).

filename : str or file handle

Output filename or handle.

duration : float, optional

Note duration if not defined by *notes*.

Returns numpy array

Notes in MIREX format.

Notes

The note columns format must be (duration and velocity being optional):

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

The output format required by MIREX is:

'onset_time' 'offset_time' 'note_frequency'

madmom.features.notes.**note_reshaper**(*notes*)

Reshapes the activations produced by a RNN to have the right shape.

Parameters **notes** : numpy array

Note activations.

Returns numpy array

Reshaped array to represent the 88 MIDI notes.

7.2.5 madmom.features.onsets

This module contains onset detection related functionality.

madmom.features.onsets.**wrap_to_pi**(*phase*)

Wrap the phase information to the range -π...π.

Parameters **phase** : numpy array

Phase of the STFT.

Returns **wrapped_phase** : numpy array

Wrapped phase.

madmom.features.onsets.**correlation_diff**(*spec*, *diff_frames=1*, *pos=False*, *diff_bins=1*)

Calculates the difference of the magnitude spectrogram relative to the N-th previous frame shifted in frequency to achieve the highest correlation between these two frames.

Parameters **spec** : numpy array

Magnitude spectrogram.

diff_frames : int, optional

Calculate the difference to the *diff_frames*-th previous frame.

pos : bool, optional

Keep only positive values.

diff_bins : int, optional

Maximum number of bins shifted for correlation calculation.

Returns **correlation_diff** : numpy array

(Positive) magnitude spectrogram differences.

Notes

This function is only because of completeness, it is not intended to be actually used, since it is extremely slow. Please consider the superflux() function, since if performs equally well but much faster.

madmom.features.onsets.**high_frequency_content**(*spectrogram*)

High Frequency Content.

Parameters **spectrogram** : `Spectrogram` instance

Spectrogram instance.

Returns **high_frequency_content** : numpy array

High frequency content onset detection function.

References

[R35]

madmom.features.onsets.**spectral_diff**(*spectrogram*, *diff_frames=None*)

Spectral Diff.

Parameters **spectrogram** : `Spectrogram` instance

Spectrogram instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

Returns **spectral_diff** : numpy array

Spectral diff onset detection function.

References

[R36]

madmom.features.onsets.**spectral_flux**(*spectrogram*, *diff_frames=None*)

Spectral Flux.

Parameters **spectrogram** : `Spectrogram` instance

Spectrogram instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

Returns **spectral_flux** : numpy array

Spectral flux onset detection function.

References

[R37]

madmom.features.onsets.**superflux**(*spectrogram*, *diff_frames=None*, *diff_max_bins=3*)

SuperFlux method with a maximum filter vibrato suppression stage.

Calculates the difference of bin k of the magnitude spectrogram relative to the N-th previous frame with the maximum filtered spectrogram.

Parameters **spectrogram** : `Spectrogram` instance

Spectrogram instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

diff_max_bins : int, optional

Number of bins used for maximum filter.

Returns **superflux** : numpy array

SuperFlux onset detection function.

Notes

This method works only properly, if the spectrogram is filtered with a filterbank of the right frequency spacing. Filter banks with 24 bands per octave (i.e. quarter-tone resolution) usually yield good results. With *max_bins* = 3, the maximum of the bins k-1, k, k+1 of the frame *diff_frames* to the left is used for the calculation of the difference.

References

[R38]

madmom.features.onsets.**complex_flux**(*spectrogram*, *diff_frames=None*, *diff_max_bins=3*, *temporal_filter=3*, *temporal_origin=0*)

ComplexFlux.

ComplexFlux is based on the SuperFlux, but adds an additional local group delay based tremolo suppression.

Parameters **spectrogram** : `Spectrogram` instance

`Spectrogram` instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

diff_max_bins : int, optional

Number of bins used for maximum filter.

temporal_filter : int, optional

Temporal maximum filtering of the local group delay [frames].

temporal_origin : int, optional

Origin of the temporal maximum filter.

Returns **complex_flux** : numpy array

ComplexFlux onset detection function.

References

[R39]

`madmom.features.onsets.`**`modified_kullback_leibler`**(*spectrogram*, *diff_frames=1*, *epsilon=1e-06*)

Modified Kullback-Leibler.

Parameters **spectrogram** : `Spectrogram` instance

`Spectrogram` instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

epsilon : float, optional

Add *epsilon* to the *spectrogram* avoid division by 0.

Returns **modified_kullback_leibler** : numpy array

MKL onset detection function.

Notes

The implementation presented in *[R40]* is used instead of the original work presented in *[R41]*.

References

[R40], *[R41]*

`madmom.features.onsets.`**`phase_deviation`**(*spectrogram*)

Phase Deviation.

Parameters **spectrogram** : `Spectrogram` instance

`Spectrogram` instance.

Returns **phase_deviation** : numpy array

Phase deviation onset detection function.

References

[R42]

madmom.features.onsets.**weighted_phase_deviation**(*spectrogram*)
Weighted Phase Deviation.

Parameters **spectrogram** : Spectrogram instance

Spectrogram instance.

Returns **weighted_phase_deviation** : numpy array

Weighted phase deviation onset detection function.

References

[R43]

madmom.features.onsets.**normalized_weighted_phase_deviation**(*spectrogram*, *epsilon=1e-06*)
Normalized Weighted Phase Deviation.

Parameters **spectrogram** : Spectrogram instance

Spectrogram instance.

epsilon : float, optional

Add *epsilon* to the *spectrogram* avoid division by 0.

Returns **normalized_weighted_phase_deviation** : numpy array

Normalized weighted phase deviation onset detection function.

References

[R44]

madmom.features.onsets.**complex_domain**(*spectrogram*)
Complex Domain.

Parameters **spectrogram** : Spectrogram instance

Spectrogram instance.

Returns **complex_domain** : numpy array

Complex domain onset detection function.

References

[R45]

madmom.features.onsets.**rectified_complex_domain**(*spectrogram*, *diff_frames=None*)
Rectified Complex Domain.

Parameters **spectrogram** : Spectrogram instance

Spectrogram instance.

diff_frames : int, optional

Number of frames to calculate the diff to.

Returns **rectified_complex_domain** : numpy array

Rectified complex domain onset detection function.

References

[R46]

class madmom.features.onsets.**SpectralOnsetProcessor**(*onset_method='superflux'*, ***kwargs*)

The SpectralOnsetProcessor class implements most of the common onset detection functions based on the magnitude or phase information of a spectrogram.

Parameters **onset_method** : str, optional

Onset detection function. See *METHODS* for possible values.

process(*spectrogram*)

Detect the onsets in the given activation function.

Parameters **spectrogram** : Spectrogram instance

Spectrogram instance.

Returns **odf** : numpy array

Onset detection function.

classmethod add_arguments(*parser*, *onset_method=None*)

Add spectral onset detection arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

onset_method : str, optional

Default onset detection method.

Returns **parser_group** : argparse argument group

Spectral onset detection argument parser group.

madmom.features.onsets.**peak_picking**(*activations*, *threshold*, *smooth=None*, *pre_avg=0*, *post_avg=0*, *pre_max=1*, *post_max=1*)

Perform thresholding and peak-picking on the given activation function.

Parameters **activations** : numpy array

Activation function.

threshold : float

Threshold for peak-picking

smooth : int or numpy array

Smooth the activation function with the kernel (size).

pre_avg : int, optional

Use *pre_avg* frames past information for moving average.

post_avg : int, optional

Use *post_avg* frames future information for moving average.

pre_max : int, optional

Use *pre_max* frames past information for moving maximum.

post_max : int, optional

Use *post_max* frames future information for moving maximum.

Returns **peak_idx** : numpy array

Indices of the detected peaks.

See also:

`smooth()`

Notes

If no moving average is needed (e.g. the activations are independent of the signal's level as for neural network activations), set *pre_avg* and *post_avg* to 0. For peak picking of local maxima, set *pre_max* and *post_max* to 1. For online peak picking, set all *post_* parameters to 0.

References

[R47]

class `madmom.features.onsets.`**`PeakPickingProcessor`**(*threshold=0.5*, *smooth=0.0*, *pre_avg=0.0*, *post_avg=0.0*, *pre_max=0.0*, *post_max=0.0*, *combine=0.03*, *delay=0.0*, *online=False*, *fps=100*, ***kwargs*)

This class implements the onset peak-picking functionality which can be used universally. It transparently converts the chosen values from seconds to frames.

Parameters **threshold** : float

Threshold for peak-picking.

smooth : float, optional

Smooth the activation function over *smooth* seconds.

pre_avg : float, optional

Use *pre_avg* seconds past information for moving average.

post_avg : float, optional

Use *post_avg* seconds future information for moving average.

pre_max : float, optional

Use *pre_max* seconds past information for moving maximum.

post_max : float, optional

Use *post_max* seconds future information for moving maximum.

combine : float, optional

Only report one onset within *combine* seconds.

delay : float, optional

Report the detected onsets *delay* seconds delayed.

online : bool, optional

Use online peak-picking, i.e. no future information.

fps : float, optional

Frames per second used for conversion of timings.

Returns **onsets** : numpy array

Detected onsets [seconds].

Notes

If no moving average is needed (e.g. the activations are independent of the signal's level as for neural network activations), *pre_avg* and *post_avg* should be set to 0. For peak picking of local maxima, set *pre_max* >= 1. / *fps* and *post_max* >= 1. / *fps*. For online peak picking, all *post_* parameters are set to 0.

References

[R48]

process(*activations*)

Detect the onsets in the given activation function.

Parameters **activations** : numpy array

Onset activation function.

Returns **onsets** : numpy array

Detected onsets [seconds].

static add_arguments(*parser*, *threshold=0.5*, *smooth=None*, *pre_avg=None*, *post_avg=None*, *pre_max=None*, *post_max=None*, *combine=0.03*, *delay=0.0*)

Add onset peak-picking related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

threshold : float

Threshold for peak-picking.

smooth : float, optional

Smooth the activation function over *smooth* seconds.

pre_avg : float, optional

Use *pre_avg* seconds past information for moving average.

post_avg : float, optional

Use *post_avg* seconds future information for moving average.

pre_max : float, optional

Use *pre_max* seconds past information for moving maximum.

post_max : float, optional

Use *post_max* seconds future information for moving maximum.

combine : float, optional

Only report one onset within *combine* seconds.

delay : float, optional

Report the detected onsets *delay* seconds delayed.

Returns **parser_group** : argparse argument group

Onset peak-picking argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

7.2.6 madmom.features.tempo

This module contains tempo related functionality.

madmom.features.tempo.**smooth_histogram**(*histogram*, *smooth*)

Smooth the given histogram.

Parameters **histogram** : tuple

Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

smooth : int or numpy array

Smoothing kernel (size).

Returns **histogram_bins** : numpy array

Bins of the smoothed histogram.

histogram_delays : numpy array

Corresponding delays.

Notes

If *smooth* is an integer, a Hamming window of that length will be used as a smoothing kernel.

madmom.features.tempo.**interval_histogram_acf**(*activations*, *min_tau=1*, *max_tau=None*)

Compute the interval histogram of the given (beat) activation function via auto-correlation as in *[R49]*.

Parameters **activations** : numpy array

Beat activation function.

min_tau : int, optional

Minimal delay for the auto-correlation function [frames].

max_tau : int, optional

Maximal delay for the auto-correlation function [frames].

Returns **histogram_bins** : numpy array

Bins of the tempo histogram.

histogram_delays : numpy array

Corresponding delays [frames].

References

[R49]

madmom.features.tempo.**interval_histogram_comb**(*activations*, *alpha*, *min_tau=1*, *max_tau=None*)

Compute the interval histogram of the given (beat) activation function via a bank of resonating comb filters as in *[R50]*.

Parameters **activations** : numpy array

Beat activation function.

alpha : float or numpy array

Scaling factor for the comb filter; if only a single value is given, the same scaling factor for all delays is assumed.

min_tau : int, optional

Minimal delay for the comb filter [frames].

max_tau : int, optional

Maximal delta for comb filter [frames].

Returns **histogram_bins** : numpy array

Bins of the tempo histogram.

histogram_delays : numpy array

Corresponding delays [frames].

References

[R50]

madmom.features.tempo.**dominant_interval**(*histogram*, *smooth=None*)

Extract the dominant interval of the given histogram.

Parameters **histogram** : tuple

Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

smooth : int or numpy array, optional

Smooth the histogram with the given kernel (size).

Returns **interval** : int

Dominant interval.

Notes

If *smooth* is an integer, a Hamming window of that length will be used as a smoothing kernel.

madmom.features.tempo.**detect_tempo**(*histogram*, *fps*)

Extract the tempo from the given histogram.

Parameters **histogram** : tuple

Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

fps : float

Frames per second.

Returns **tempi** : numpy array

Numpy array with the dominant tempi [bpm] (first column) and their relative strengths (second column).

class madmom.features.tempo.**TempoEstimationProcessor**(*method='comb'*, *min_bpm=40.0*, *max_bpm=250.0*, *act_smooth=0.14*, *hist_smooth=7*, *alpha=0.79*, *fps=None*, ***kwargs*)

Tempo Estimation Processor class.

Parameters **method** : {'comb', 'acf', 'dbn'}

Method used for tempo estimation.

min_bpm : float, optional

Minimum tempo to detect [bpm].

max_bpm : float, optional

Maximum tempo to detect [bpm].

act_smooth : float, optional (default: 0.14)

Smooth the activation function over *act_smooth* seconds.

hist_smooth : int, optional (default: 7)

Smooth the tempo histogram over *hist_smooth* bins.

alpha : float, optional

Scaling factor for the comb filter.

fps : float, optional

Frames per second.

min_interval

Minimum beat interval [frames].

max_interval

Maximum beat interval [frames].

process(*activations*)

Detect the tempi from the (beat) activations.

Parameters **activations** : numpy array

Beat activation function.

Returns **tempi** : numpy array

Array with the dominant tempi [bpm] (first column) and their relative strengths (second column).

interval_histogram(*activations*)

Compute the histogram of the beat intervals with the selected method.

Parameters **activations** : numpy array

Beat activation function.

Returns **histogram_bins** : numpy array

Bins of the beat interval histogram.

histogram_delays : numpy array

Corresponding delays [frames].

dominant_interval(*histogram*)

Extract the dominant interval of the given histogram.

Parameters **histogram** : tuple

Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

Returns **interval** : int

Dominant interval.

static add_arguments(*parser*, *method='comb'*, *min_bpm=40.0*, *max_bpm=250.0*, *act_smooth=0.14*, *hist_smooth=7*, *alpha=0.79*)

Add tempo estimation related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser.

method : {'comb', 'acf', 'dbn'}

Method used for tempo estimation.

min_bpm : float, optional

Minimum tempo to detect [bpm].

max_bpm : float, optional

Maximum tempo to detect [bpm].

act_smooth : float, optional

Smooth the activation function over *act_smooth* seconds.

hist_smooth : int, optional

Smooth the tempo histogram over *hist_smooth* bins.

alpha : float, optional

Scaling factor for the comb filter.

Returns **parser_group** : argparse argument group

Tempo argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

madmom.features.tempo.**write_tempo**(*tempi*, *filename*, *mirex=False*)

Write the most dominant tempi and the relative strength to a file.

Parameters **tempi** : numpy array

Array with the detected tempi (first column) and their strengths (second column).

filename : str or file handle

Output file.

mirex : bool, optional

Report the lower tempo first (as required by MIREX).

Returns **tempo_1** : float

The most dominant tempo.

tempo_2 : float

The second most dominant tempo.

strength : float

Their relative strength.

CHAPTER 8

madmom.evaluation

Evaluation package.

madmom.evaluation.**find_closest_matches**(*detections*, *annotations*)

Find the closest annotation for each detection.

Parameters **detections** : list or numpy array

Detected events.

annotations : list or numpy array

Annotated events.

Returns **indices** : numpy array

Indices of the closest matches.

Notes

The sequences must be ordered.

madmom.evaluation.**calc_errors**(*detections*, *annotations*, *matches=None*)

Errors of the detections to the closest annotations.

Parameters **detections** : list or numpy array

Detected events.

annotations : list or numpy array

Annotated events.

matches : list or numpy array

Indices of the closest events.

Returns **errors** : numpy array

Errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

madmom.evaluation.**calc_absolute_errors**(*detections*, *annotations*, *matches=None*)
Absolute errors of the detections to the closest annotations.

Parameters **detections** : list or numpy array

Detected events.

annotations : list or numpy array

Annotated events.

matches : list or numpy array

Indices of the closest events.

Returns **errors** : numpy array

Absolute errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

madmom.evaluation.**calc_relative_errors**(*detections*, *annotations*, *matches=None*)
Relative errors of the detections to the closest annotations.

Parameters **detections** : list or numpy array

Detected events.

annotations : list or numpy array

Annotated events.

matches : list or numpy array

Indices of the closest events.

Returns **errors** : numpy array

Relative errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

class madmom.evaluation.**EvaluationMixin**
Evaluation mixin class.

This class has a *name* attribute which is used for display purposes and defaults to 'None'.

METRIC_NAMES is a list of tuples, containing the attribute's name and the corresponding label, e.g.:

The attributes defined in *METRIC_NAMES* will be provided as an ordered dictionary as the *metrics* property unless the subclass overwrites the property.

FLOAT_FORMAT is used to format floats.

metrics
Metrics as a dictionary.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

Notes

This is a fallback method formatting the *metrics* dictionary in a human readable way. Classes inheriting from this mixin class should provide a method better suitable.

class madmom.evaluation.**SimpleEvaluation**(*num_tp=0*, *num_fp=0*, *num_tn=0*, *num_fn=0*, *name=None*, ***kwargs*)
Simple Precision, Recall, F-measure and Accuracy evaluation based on the numbers of true/false positive/negative detections.

Parameters **num_tp** : int

Number of true positive detections.

num_fp : int

Number of false positive detections.

num_tn : int

Number of true negative detections.

num_fn : int

Number of false negative detections.

name : str

Name to be displayed.

Notes

This class is only suitable for a 1-class evaluation problem.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

precision
Precision.

recall
Recall.

fmeasure
F-measure.

accuracy
Accuracy.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.**Evaluation**(*tp=None*, *fp=None*, *tn=None*, *fn=None*, ***kwargs*)
Evaluation class for measuring Precision, Recall and F-measure based on numpy arrays or lists with true/false positive/negative detections.

Parameters **tp** : list or numpy array

True positive detections.

fp : list or numpy array

False positive detections.

tn : list or numpy array

True negative detections.

fn : list or numpy array

False negative detections.

name : str

Name to be displayed.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

class madmom.evaluation.**MultiClassEvaluation**(*tp=None*, *fp=None*, *tn=None*, *fn=None*, ***kwargs*)
Evaluation class for measuring Precision, Recall and F-measure based on 2D numpy arrays with true/false positive/negative detections.

Parameters **tp** : list of tuples or numpy array, shape (num_tp, 2)

True positive detections.

fp : list of tuples or numpy array, shape (num_fp, 2)

False positive detections.

tn : list of tuples or numpy array, shape (num_tn, 2)

True negative detections.

fn : list of tuples or numpy array, shape (num_fn, 2)

False negative detections.

name : str

Name to be displayed.

Notes

The second item of the tuples or the second column of the arrays denote the class the detection belongs to.

tostring(*verbose=False*, ***kwargs*)
Format the evaluation metrics as a human readable string.

Parameters **verbose** : bool

Add evaluation for individual classes.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.**SumEvaluation**(*eval_objects*, *name=None*)
Simple class for summing evaluations.

Parameters **eval_objects** : list

Evaluation objects.

name : str

Name to be displayed.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

class madmom.evaluation.**MeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Simple class for averaging evaluation.

Parameters **eval_objects** : list

Evaluation objects.

name : str

Name to be displayed.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

precision
Precision.

recall
Recall.

fmeasure
F-measure.

accuracy
Accuracy.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.**tostring**(*eval_objects*, ***kwargs*)
Format the given evaluation objects as human readable strings.

Parameters **eval_objects** : list

Evaluation objects.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.**tocsv**(*eval_objects*, *metric_names=None*, *float_format='{:.3f}'*, ***kwargs*)
Format the given evaluation objects as a CSV table.

Parameters **eval_objects** : list

Evaluation objects.

metric_names : list of tuples, optional

List of tuples defining the name of the property corresponding to the metric, and the metric label e.g. ('fp', 'False Positives').

float_format : str, optional

How to format the metrics.

Returns str

CSV table representation of the evaluation objects.

Notes

If no *metric_names* are given, they will be extracted from the first evaluation object.

madmom.evaluation.**totex**(*eval_objects*, *metric_names=None*, *float_format='{:.3f}'*, ***kwargs*)
Format the given evaluation objects as a LaTeX table.

Parameters **eval_objects** : list

Evaluation objects.

metric_names : list of tuples, optional

List of tuples defining the name of the property corresponding to the metric, and the metric label e.g. ('fp', 'False Positives').

float_format : str, optional

How to format the metrics.

Returns str

LaTeX table representation of the evaluation objects.

Notes

If no *metric_names* are given, they will be extracted from the first evaluation object.

madmom.evaluation.**evaluation_io**(*parser*, *ann_suffix*, *det_suffix*, *ann_dir=None*, *det_dir=None*)

Add evaluation input/output and formatting related arguments to an existing parser object.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

ann_suffix : str

Suffix of the annotation files.

det_suffix : str

Suffix of the detection files.

ann_dir : str, optional

Use only annotations from this folder (and sub-folders).

det_dir : str, optional

Use only detections from this folder (and sub-folders).

Returns **io_group** : argparse argument group

Evaluation input / output argument group.

formatter_group : argparse argument group

Evaluation formatter argument group.

8.1 Submodules

8.1.1 madmom.evaluation.alignment

This module contains global alignment evaluation functionality.

exception madmom.evaluation.alignment.**AlignmentFormatError**(*value=None*)

Exception to be raised whenever an incorrect alignment format is given.

madmom.evaluation.alignment.**load_alignment**(*values*)

Load the alignment from given values or file.

Parameters **values** : str, file handle, list or numpy array

Alignment values.

Returns numpy array

Time and score position columns.

madmom.evaluation.alignment.**compute_event_alignment**(*alignment*, *ground_truth*)

This function finds the alignment outputs corresponding to each ground truth alignment. In general, the alignment algorithm will output more alignment positions than events in the score, e.g. if it is designed to output the current alignment at constant intervals.

Parameters **alignment** : 2D numpy array

The score follower's resulting alignment. 2D array, first value is the time in seconds, second value is the beat position.

ground_truth : 2D numpy array

Ground truth of the aligned performance. 2D array, first value is the time in seconds, second value is the beat position. It can contain the alignment positions for each individual note. In this case, the deviation for each note is taken into account.

Returns numpy array

Array of the same size as *ground_truth*, with each row representing the alignment of the corresponding ground truth element..

madmom.evaluation.alignment.**compute_metrics**(*event_alignment*, *ground_truth*, *window*, *err_hist_bins*)

This function computes the evaluation metrics based on the paper *[R2]* plus an cumulative histogram of absolute errors.

Parameters **event_alignment** : 2D numpy array

Sequence alignment as computed by the score follower. 2D array, where the first column is the alignment time in seconds and the second column the position in beats. Needs to be the same length as *ground_truth*, hence for each element in the ground truth the corresponding alignment has to be available. Use the *compute_event_alignment()* function to compute this.

ground_truth : 2D numpy array

Ground truth of the aligned performance. 2D array, first value is the time in seconds, second value is the beat position. It can contain the alignment positions for each individual note. In this case, the deviation for each note is taken into account.

window : float

Tolerance window in seconds. Alignments off less than this amount from the ground truth will be considered correct.

err_hist_bins : list

List of error bounds for which the cumulative histogram of absolute error will be computed (e.g. [0.1, 0.3] will give the percentage of events aligned with an error smaller than 0.1 and 0.3).

Returns **metrics** : dict

(Some) of the metrics described in *[R2]* and the error histogram.

References

[R2]

class madmom.evaluation.alignment.**AlignmentEvaluation**(*alignment*, *ground_truth*, *window=0.25*, *name=None*, ***kwargs*)

Alignment evaluation class for beat-level alignments. Beat-level aligners output beat positions for points in time, rather than computing a time step for each individual event in the score. The following metrics are available:

Parameters **alignment** : 2D numpy array or list of tuples

Computed alignment; first value is the time in seconds, second value is the beat position.

ground_truth : 2D numpy array or list of tuples

Ground truth of the aligned file; first value is the time in seconds, second value is the beat position. It can contain the alignment positions for each individual event. In this case, the deviation for each event is taken into account.

window : float

Tolerance window in seconds. Alignments off less than this amount from the ground truth will be considered correct.

name : str

Name to be displayed.

Attributes

miss_rate	(float) Percentage of missed events (events that exist in the reference score, but are not reported).
misalign_rate	(float) Percentage of misaligned events (events with an alignment that is off by more than a defined *window*).
avg_imprecision	(float) Average alignment error of non-misaligned events.
std-dev_imprecision	(float) Standard deviation of alignment error of non-misaligned events.
avg_error	(float) Average alignment error.
stddev_error	(float) Standard deviation of alignment error.
piece_completion	(float) Percentage of events that was followed until the aligner hangs, i.e from where on there are only misaligned or missed events.
be-low_{x}_{yy}	(float) Percentage of events that are aligned with an error smaller than x.yy seconds.

tostring(*histogram=False*, ***kwargs*)

Format the evaluation metrics as a human readable string.

Parameters **histogram** : bool

Also output the error histogram.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.alignment.**AlignmentSumEvaluation**(*eval_objects*, *name=None*)

Class for averaging alignment evaluation scores, considering the lengths of the aligned pieces. For a detailed description of the available metrics, refer to AlignmentEvaluation.

Parameters **eval_objects** : list

Evaluation objects.

name : str

Name to be displayed.

class madmom.evaluation.alignment.**AlignmentMeanEvaluation**(*eval_objects*, *name=None*)

Class for averaging alignment evaluation scores, averaging piecewise (i.e. ignoring the lengths of the pieces). For a detailed description of the available metrics, refer to AlignmentEvaluation.

Parameters **eval_objects** : list

Evaluation objects.

name : str

Name to be displayed.

madmom.evaluation.alignment.**add_parser**(*parser*)

Add an alignment evaluation sub-parser to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

Returns **sub_parser** : argparse sub-parser instance

Alignment evaluation sub-parser.

parser_group : argparse argument group

Alignment evaluation argument group.

8.1.2 madmom.evaluation.beats

This module contains beat evaluation functionality.

The measures are described in *[R3]*, a Matlab implementation exists here: http://code.soundsoftware.ac.uk/projects/beat-evaluation/repository

Notes

Please note that this is a complete re-implementation, which took some other design decisions. For example, the beat detections and annotations are not quantised before being evaluated with F-measure, P-score and other metrics. Hence these evaluation functions DO NOT report the exact same results/scores. This approach was chosen, because it is simpler and produces more accurate results.

References

exception madmom.evaluation.beats.**BeatIntervalError**(*value=None*)

Exception to be raised whenever an interval cannot be computed.

madmom.evaluation.beats.**load_beats**(**args*, ***kwargs*)

Load the beats from the given values or file.

To make this function more universal, it also accepts lists or arrays.

Parameters **values** : str, file handle, list or numpy array

Name / values to be loaded.

downbeats : bool, optional

Load downbeats instead of beats.

Returns numpy array

Beats.

Notes

Expected format:

'beat_time' [additional information will be ignored]

madmom.evaluation.beats.**variations**(*sequence*, *offbeat=False*, *double=False*, *half=False*, *triple=False*, *third=False*)

Create variations of the given beat sequence.

Parameters **sequence** : numpy array

Beat sequence.

offbeat : bool, optional

Create an offbeat sequence.

double : bool, optional

Create a double tempo sequence.

half : bool, optional

Create half tempo sequences (includes offbeat version).

triple : bool, optional

Create triple tempo sequence.

third : bool, optional

Create third tempo sequences (includes offbeat versions).

Returns list

Beat sequence variations.

madmom.evaluation.beats.**calc_intervals**(*events*, *fwd=False*)

Calculate the intervals of all events to the previous/next event.

Parameters **events** : numpy array

Beat sequence.

fwd : bool, optional

Calculate the intervals towards the next event (instead of previous).

Returns numpy array

Beat intervals.

Notes

The sequence must be ordered. The first (last) interval will be set to the same value as the second (second to last) interval (when used in *fwd* mode).

madmom.evaluation.beats.**find_closest_intervals**(*detections*, *annotations*, *matches=None*)

Find the closest annotated interval to each beat detection.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

matches : list or numpy array

Indices of the closest beats.

Returns numpy array

Closest annotated beat intervals.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

The function does NOT test if each detection has a surrounding interval, it always returns the closest interval.

madmom.evaluation.beats.**find_longest_continuous_segment**(*sequence_indices*)

ind the longest consecutive segment in the given sequence.

Parameters **sequence_indices** : numpy array

Indices of the beats

Returns **length** : int

Length of the longest consecutive segment.

start : int

Start position of the longest continuous segment.

madmom.evaluation.beats.**calc_relative_errors**(*detections*, *annotations*, *matches=None*)

Errors of the detections relative to the closest annotated interval.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

matches : list or numpy array

Indices of the closest beats.

Returns numpy array

Errors relative to the closest annotated beat interval.

Notes

The sequences must be ordered! To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

madmom.evaluation.beats.**pscore**(*detections*, *annotations*, *tolerance=0.2*)

Calculate the P-score accuracy for the given detections and annotations.

The P-score is determined by taking the sum of the cross-correlation between two impulse trains, representing the detections and annotations allowing for a tolerance of 20% of the median annotated interval *[R4]*.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

tolerance : float, optional

Evaluation tolerance (fraction of the median beat interval).

Returns **pscore** : float

P-Score.

Notes

Contrary to the original implementation which samples the two impulse trains with 100Hz, we do not quantise the annotations and detections but rather count all detections falling withing the defined tolerance window.

References

[R4]

madmom.evaluation.beats.**cemgil**(*detections*, *annotations*, *sigma=0.04*)

Calculate the Cemgil accuracy for the given detections and annotations.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

sigma : float, optional

Sigma for Gaussian error function.

Returns **cemgil** : float

Cemgil beat tracking accuracy.

References

[R5]

madmom.evaluation.beats.**goto**(*detections*, *annotations*, *threshold=0.175*, *sigma=0.1*, *mu=0.1*)

Calculate the Goto and Muraoka accuracy for the given detections and annotations.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

threshold : float, optional

Threshold.

sigma : float, optional

Allowed std. dev. of the errors in the longest segment.

mu : float, optional

Allowed mean. of the errors in the longest segment.

Returns **goto** : float

Goto beat tracking accuracy.

Notes

[R6] requires that the first correct beat detection must occur within the first 3/4 of the excerpt. In order to be able to deal with audio with varying tempo, this was altered that the length of the longest continuously tracked segment must be at least 1/4 of the total length *[R7]*.

References

[R6], *[R7]*

`madmom.evaluation.beats.`**`cml`**(*detections*, *annotations*, *phase_tolerance=0.175*, *tempo_tolerance=0.175*)

Calculate the cmlc and cmlt scores for the given detections and annotations.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

phase_tolerance : float, optional

Allowed phase tolerance.

tempo_tolerance : float, optional

Allowed tempo tolerance.

Returns **cmlc** : float

Longest continuous segment of correct detections normalized by the maximum length of both sequences (detection and annotations).

cmlt : float

Same as cmlc, but no continuity required.

References

[R8], *[R9]*

madmom.evaluation.beats.**continuity**(*detections*, *annotations*, *phase_tolerance=0.175*, *tempo_tolerance=0.175*, *offbeat=True*, *double=True*, *triple=True*)

Calculate the cmlc, cmlt, amlc and amlt scores for the given detections and annotations.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

phase_tolerance : float, optional

Allowed phase tolerance.

tempo_tolerance : float, optional

Allowed tempo tolerance.

offbeat : bool, optional

Include offbeat variation.

double : bool, optional

Include double and half tempo variations (and offbeat thereof).

triple : bool, optional

Include triple and third tempo variations (and offbeats thereof).

Returns **cmlc** : float

Tracking accuracy, continuity at the correct metrical level required.

cmlt : float

Same as cmlc, continuity at the correct metrical level not required.

amlc : float

Same as cmlc, alternate metrical levels allowed.

amlt : float

Same as cmlt, alternate metrical levels allowed.

See also:

cml()

madmom.evaluation.beats.**information_gain**(*detections*, *annotations*, *num_bins=40*)

Calculate information gain for the given detections and annotations.

Parameters **detections** : list or numpy array

Detected beats.

annotations : list or numpy array

Annotated beats.

num_bins : int, optional

Number of bins for the beat error histogram.

Returns **information_gain** : float

Information gain.

error_histogram : numpy array

Error histogram.

References

[R10]

class madmom.evaluation.beats.**BeatEvaluation**(*detections*, *annotations*, *fmeasure_window=0.07*, *pscore_tolerance=0.2*, *cemgil_sigma=0.04*, *goto_threshold=0.175*, *goto_sigma=0.1*, *goto_mu=0.1*, *continuity_phase_tolerance=0.175*, *continuity_tempo_tolerance=0.175*, *information_gain_bins=40*, *offbeat=True*, *double=True*, *triple=True*, *skip=0*, *downbeats=False*, ***kwargs*)

Beat evaluation class.

Parameters **detections** : str, list or numpy array

Detected beats.

annotations : str, list or numpy array

Annotated ground truth beats.

fmeasure_window : float, optional

F-measure evaluation window [seconds]

pscore_tolerance : float, optional

P-Score tolerance [fraction of the median beat interval].

cemgil_sigma : float, optional

Sigma of Gaussian window for Cemgil accuracy.

goto_threshold : float, optional

Threshold for Goto error.

goto_sigma : float, optional

Sigma for Goto error.

goto_mu : float, optional

Mu for Goto error.

continuity_phase_tolerance : float, optional

Continuity phase tolerance.

continuity_tempo_tolerance : float, optional

Ccontinuity tempo tolerance.

information_gain_bins : int, optional

Number of bins for for the information gain beat error histogram.

offbeat : bool, optional

Include offbeat variation.

double : bool, optional

Include double and half tempo variations (and offbeat thereof).

triple : bool, optional

Include triple and third tempo variations (and offbeats thereof).

skip : float, optional

Skip the first *skip* seconds for evaluation.

downbeats : bool, optional

Evaluate downbeats instead of beats.

Notes

The *offbeat*, *double*, and *triple* variations of the beat sequences are used only for AMLc/AMLt.

global_information_gain
Global information gain.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.beats.**BeatMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Class for averaging beat evaluation scores.

fmeasure
F-measure.

pscore
P-score.

cemgil
Cemgil accuracy.

goto
Goto accuracy.

cmlc
CMLc.

cmlt
CMLt.

amlc
AMLc.

amlt
AMLt.

information_gain
Information gain.

error_histogram
Error histogram.

global_information_gain
Global information gain.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.beats.**add_parser**(*parser*)
Add a beat evaluation sub-parser to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

Returns **sub_parser** : argparse sub-parser instance

Beat evaluation sub-parser.

parser_group : argparse argument group

Beat evaluation argument group.

8.1.3 madmom.evaluation.notes

This module contains note evaluation functionality.

madmom.evaluation.notes.**load_notes**(**args*, ***kwargs*)
Load the notes from the given values or file.

Parameters **values: str, file handle, list of tuples or numpy array**

Notes values.

Returns numpy array

Notes.

Notes

Expected file/tuple/row format:

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

madmom.evaluation.notes.**remove_duplicate_notes**(*data*)
Remove duplicate rows from the array.

Parameters **data** : numpy array

Data.

Returns numpy array

Data array with duplicate rows removed.

Notes

This function removes only exact duplicates.

madmom.evaluation.notes.**note_onset_evaluation**(*detections*, *annotations*, *window=0.025*)

Determine the true/false positive/negative note onset detections.

Parameters **detections** : numpy array

Detected notes.

annotations : numpy array

Annotated ground truth notes.

window : float, optional

Evaluation window [seconds].

Returns **tp** : numpy array, shape (num_tp, 2)

True positive detections.

fp : numpy array, shape (num_fp, 2)

False positive detections.

tn : numpy array, shape (0, 2)

True negative detections (empty, see notes).

fn : numpy array, shape (num_fn, 2)

False negative detections.

errors : numpy array, shape (num_tp, 2)

Errors of the true positive detections wrt. the annotations.

Notes

The expected note row format is:

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

The returned true negative array is empty, because we are not interested in this class, since it is magnitudes bigger than true positives array.

class madmom.evaluation.notes.**NoteEvaluation**(*detections*, *annotations*, *window=0.025*, *delay=0*, ***kwargs*)

Evaluation class for measuring Precision, Recall and F-measure of notes.

Parameters **detections** : str, list or numpy array

Detected notes.

annotations : str, list or numpy array

Annotated ground truth notes.

window : float, optional

F-measure evaluation window [seconds]

delay : float, optional

Delay the detections *delay* seconds for evaluation.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(*notes=False*, ***kwargs*)

Parameters **notes** : bool, optional

Display detailed output for all individual notes.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.notes.**NoteSumEvaluation**(*eval_objects*, *name=None*)
Class for summing note evaluations.

errors
Errors of the true positive detections wrt. the ground truth.

class madmom.evaluation.notes.**NoteMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Class for averaging note evaluations.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.notes.**add_parser**(*parser*)
Add a note evaluation sub-parser to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

Returns **sub_parser** : argparse sub-parser instance

Note evaluation sub-parser.

parser_group : argparse argument group

Note evaluation argument group.

8.1.4 madmom.evaluation.onsets

This module contains onset evaluation functionality described in *[R11]*:

References

madmom.evaluation.onsets.**load_onsets**(**args*, ***kwargs*)
Load the onsets from the given values or file.

Parameters **values: str, file handle, list of tuples or numpy array**

Onsets values.

Returns numpy array, shape (num_onsets,)

Onsets.

Notes

Expected file/tuple/row format:

'onset_time' [additional information will be ignored]

`madmom.evaluation.onsets.`**`onset_evaluation`**(*detections*, *annotations*, *window=0.025*)

Determine the true/false positive/negative detections.

Parameters **detections** : numpy array

Detected notes.

annotations : numpy array

Annotated ground truth notes.

window : float, optional

Evaluation window [seconds].

Returns **tp** : numpy array, shape (num_tp,)

True positive detections.

fp : numpy array, shape (num_fp,)

False positive detections.

tn : numpy array, shape (0,)

True negative detections (empty, see notes).

fn : numpy array, shape (num_fn,)

False negative detections.

errors : numpy array, shape (num_tp,)

Errors of the true positive detections wrt. the annotations.

Notes

The returned true negative array is empty, because we are not interested in this class, since it is magnitudes bigger than true positives array.

class `madmom.evaluation.onsets.`**`OnsetEvaluation`**(*detections*, *annotations*, *window=0.025*, *combine=0*, *delay=0*, ***kwargs*)

Evaluation class for measuring Precision, Recall and F-measure of onsets.

Parameters **detections** : str, list or numpy array

Detected notes.

annotations : str, list or numpy array

Annotated ground truth notes.

window : float, optional

F-measure evaluation window [seconds]

combine : float, optional

Combine all annotated onsets within *combine* seconds.

delay : float, optional

Delay the detections *delay* seconds for evaluation.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.onsets.**OnsetSumEvaluation**(*eval_objects*, *name=None*)
Class for summing onset evaluations.

errors
Errors of the true positive detections wrt. the ground truth.

class madmom.evaluation.onsets.**OnsetMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Class for averaging onset evaluations.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.onsets.**add_parser**(*parser*)
Add an onset evaluation sub-parser to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

Returns **sub_parser** : argparse sub-parser instance

Onset evaluation sub-parser.

parser_group : argparse argument group

Onset evaluation argument group.

8.1.5 madmom.evaluation.tempo

This module contains tempo evaluation functionality.

madmom.evaluation.tempo.**load_tempo**(*values*, *split_value=1.0*, *sort=False*, *norm_strengths=False*, *max_len=None*)

Load tempo information from the given values or file.

Parameters **values** : str, file handle, list of tuples or numpy array

Tempo values or file name/handle.

split_value : float, optional

Value to distinguish between tempi and strengths. *values* > *split_value* are interpreted as tempi [bpm], *values* <= *split_value* are interpreted as strengths.

sort : bool, optional

Sort the tempi by their strength.

norm_strengths : bool, optional

Normalize the strengths to sum 1.

max_len : int, optional

Return at most *max_len* tempi.

Returns **tempi** : numpy array, shape (num_tempi, 2)

Array with tempi (rows, first column) and their relative strengths (second column).

Notes

The tempo must have the one of the following formats (separated by whitespace if loaded from file):

'tempo_one' 'tempo_two' 'relative_strength' (of the first tempo) 'tempo_one' 'tempo_two' 'strength_one' 'strength_two'

If no strengths are given, uniformly distributed strengths are returned.

madmom.evaluation.tempo.**tempo_evaluation**(*detections*, *annotations*, *tolerance=0.04*)

Calculate the tempo P-Score, at least one or both tempi correct.

Parameters **detections** : list of tuples or numpy array

Detected tempi (rows, first column) and their relative strengths (second column).

annotations : list or numpy array

Annotated tempi (rows, first column) and their relative strengths (second column).

tolerance : float, optional

Evaluation tolerance (max. allowed deviation).

Returns **pscore** : float

P-Score.

at_least_one : bool

At least one tempo correctly identified.

all : bool

All tempi correctly identified.

Notes

All given detections are evaluated against all annotations according to the relative strengths given. If no strengths are given, evenly distributed strengths are assumed. If the strengths do not sum to 1, they will be normalized.

References

[R12]

class madmom.evaluation.tempo.**TempoEvaluation**(*detections*, *annotations*, *tolerance=0.04*, *double=True*, *triple=True*, *sort=True*, *max_len=None*, *name=None*, ***kwargs*)

Tempo evaluation class.

Parameters **detections** : str, list of tuples or numpy array

Detected tempi (rows) and their strengths (columns). If a file name is given, load them from this file.

annotations : str, list or numpy array

Annotated ground truth tempi (rows) and their strengths (columns). If a file name is given, load them from this file.

tolerance : float, optional

Evaluation tolerance (max. allowed deviation).

double : bool, optional

Include double and half tempo variations.

triple : bool, optional

Include triple and third tempo variations.

sort : bool, optional

Sort the tempi by their strengths (descending order).

max_len : bool, optional

Evaluate at most *max_len* tempi.

name : str, optional

Name of the evaluation to be displayed.

Notes

For P-Score, the number of detected tempi will be limited to the number of annotations (if not further limited by *max_len*). For Accuracy 1 & 2 only one detected tempo is used. Depending on *sort*, this can be either the first or the strongest one.

tostring(***kwargs*)

Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

class madmom.evaluation.tempo.**TempoMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)

Class for averaging tempo evaluation scores.

pscore
P-Score.

any
At least one tempo correct.

all
All tempi correct.

acc1
Accuracy 1.

acc2
Accuracy 2.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns str

Evaluation metrics formatted as a human readable string.

madmom.evaluation.tempo.**add_parser**(*parser*)

Add a tempo evaluation sub-parser to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

Returns **sub_parser** : argparse sub-parser instance

Tempo evaluation sub-parser.

parser_group : argparse argument group

Tempo evaluation argument group.

CHAPTER 9

madmom.ml

Machine learning package.

9.1 Submodules

9.1.1 madmom.ml.gmm

This module contains functionality needed for fitting and scoring Gaussian Mixture Models (GMMs) (needed e.g. in madmom.features.beats_hmm).

The needed functionality is taken from sklearn.mixture.GMM which is released under the BSD license and was written by these authors:

- Ron Weiss <ronweiss@gmail.com>
- Fabian Pedregosa <fabian.pedregosa@inria.fr>
- Bertrand Thirion <bertrand.thirion@inria.fr>

This version works with sklearn v0.16 an onwards. All commits until 0650d5502e01e6b4245ce99729fc8e7a71aacff3 are incorporated.

madmom.ml.gmm.**logsumexp**(*arr*, *axis=0*)

Computes the sum of arr assuming arr is in the log domain.

Parameters **arr** : numpy array

Input data [log domain].

axis : int, optional

Axis to operate on.

Returns numpy array

log(sum(exp(arr))) while minimizing the possibility of over/underflow.

Notes

Function copied from sklearn.utils.extmath.

madmom.ml.gmm.**pinvh**(*a*, *cond=None*, *rcond=None*, *lower=True*)

Compute the (Moore-Penrose) pseudo-inverse of a hermetian matrix.

Calculate a generalized inverse of a symmetric matrix using its eigenvalue decomposition and including all 'large' eigenvalues.

Parameters **a** : array, shape (N, N)

Real symmetric or complex hermetian matrix to be pseudo-inverted.

cond, rcond : float or None

Cutoff for 'small' eigenvalues. Singular values smaller than rcond * largest_eigenvalue are considered zero. If None or -1, suitable machine precision is used.

lower : boolean

Whether the pertinent array data is taken from the lower or upper triangle of *a*.

Returns **B** : array, shape (N, N)

Raises **LinAlgError**

If eigenvalue does not converge

Notes

Function copied from sklearn.utils.extmath.

madmom.ml.gmm.**log_multivariate_normal_density**(*x*, *means*, *covars*, *covariance_type='diag'*)

Compute the log probability under a multivariate Gaussian distribution.

Parameters **x** : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

means : array_like, shape (n_components, n_features)

List of n_features-dimensional mean vectors for n_components Gaussians. Each row corresponds to a single mean vector.

covars : array_like

List of n_components covariance parameters for each Gaussian. The shape depends on *covariance_type*:

- (n_components, n_features) if 'spherical',
- (n_features, n_features) if 'tied',
- (n_components, n_features) if 'diag',
- (n_components, n_features, n_features) if 'full'.

covariance_type : {'diag', 'spherical', 'tied', 'full'}

Type of the covariance parameters. Defaults to 'diag'.

Returns **lpr** : array_like, shape (n_samples, n_components)

Array containing the log probabilities of each data point in *x* under each of the n_components multivariate Gaussian distributions.

class madmom.ml.gmm.**GMM**(*n_components=1*, *covariance_type='full'*)

Gaussian Mixture Model

Representation of a Gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

Parameters **n_components** : int, optional

Number of mixture components. Defaults to 1.

covariance_type : {'diag', 'spherical', 'tied', 'full'}

String describing the type of covariance parameters to use. Defaults to 'diag'.

Attributes

*weights_*	(array, shape (n_components,)) This attribute stores the mixing weights for each mixture component.
*means_*	(array, shape (n_components, n_features)) Mean parameters for each mixture component.
*covars_*	(array) Covariance parameters for each mixture component. The shape depends on *covariance_type*.:: - (n_components, n_features) if 'spherical', - (n_features, n_features) if 'tied', - (n_components, n_features) if 'diag', - (n_components, n_features, n_features) if 'full'.
*converged_*	(bool) True when convergence was reached in fit(), False otherwise.

score_samples(*x*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters **x: array_like, shape (n_samples, n_features)**

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns **logprob** : array_like, shape (n_samples,)

Log probabilities of each data point in *x*.

responsibilities : array_like, shape (n_samples, n_components)

Posterior probabilities of each mixture component for each observation.

score(*x*)

Compute the log probability under the model.

Parameters **x** : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns **logprob** : array_like, shape (n_samples,)

Log probabilities of each data point in *x*.

fit(*x*, *random_state=None*, *tol=0.001*, *min_covar=0.001*, *n_iter=100*, *n_init=1*, *params='wmc'*, *init_params='wmc'*)

Estimate model parameters with the expectation-maximization algorithm.

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string '' when creating the GMM object. Likewise, if you would like just to do an initialization, set n_iter=0.

Parameters **x** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

random_state: RandomState or an int seed (0 by default)

A random number generator instance.

min_covar : float, optional

Floor on the diagonal of the covariance matrix to prevent overfitting.

tol : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold.

n_iter : int, optional

Number of EM iterations to perform.

n_init : int, optional

Number of initializations to perform, the best results is kept.

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

init_params : string, optional

Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

9.1.2 madmom.ml.hmm

This module contains Hidden Markov Model (HMM) functionality.

Notes

If you want to change this module and use it interactively, use pyximport.

```
>>> import pyximport
>>> pyximport.install(reload_support=True,
                      setup_args={'include_dirs': np.get_include()})
```

class madmom.ml.hmm.**DiscreteObservationModel**

Simple discrete observation model that takes an observation matrix of the form (num_states x num_observations) containing P(observation | state).

Parameters **observation_probabilities** : numpy array

Observation probabilities as a 2D array of shape (num_observations, num_states). Has to sum to 1 over the second axis, since it represents P(observation | state).

densities(*self*, *observations*)

Densities of the observations.

Parameters **observations** : numpy array

Observations.

Returns numpy array

Densities of the observations.

log_densities(*self*, *observations*)

Log densities of the observations.

Parameters **observations** : numpy array

Observations.

Returns numpy array

Log densities of the observations.

madmom.ml.hmm.**HMM**
alias of *HiddenMarkovModel*

class madmom.ml.hmm.**HiddenMarkovModel**
Hidden Markov Model

To search for the best path through the state space with the Viterbi algorithm, the following parameters must be defined.

Parameters **transition_model** : *TransitionModel* instance

Transition model.

observation_model : *ObservationModel* instance

Observation model.

initial_distribution : numpy array, optional

Initial state distribution; if 'None' a uniform distribution is assumed.

forward(*self*, *observations*)
Compute the forward variables at each time step. Instead of computing in the log domain, we normalise at each step, which is faster for the forward algorithm.

Parameters **observations** : numpy array

Observations to compute the forward variables for.

Returns numpy array, shape (num_observations, num_states)

Forward variables.

forward_generator(*self*, *observations*, *block_size=None*)
Compute the forward variables at each time step. Instead of computing in the log domain, we normalise at each step, which is faster for the forward algorithm. This function is a generator that yields the forward variables for each time step individually to save memory. The observation densities are computed block-wise to save Python calls in the inner loops.

Parameters **observations** : numpy array

Observations to compute the forward variables for.

block_size : int, optional

Block size for the block-wise computation of observation densities. If 'None', all observation densities will be computed at once.

Yields numpy array, shape (num_states,)

Forward variables.

viterbi(*self*, *observations*)
Determine the best path with the Viterbi algorithm.

Parameters **observations** : numpy array

Observations to decode the optimal path for.

Returns **path** : numpy array

Best state-space path sequence.

log_prob : float

Corresponding log probability.

class madmom.ml.hmm.**ObservationModel**

Observation model class for a HMM.

The observation model is defined as a plain 1D numpy arrays *pointers* and the methods *log_densities()* and *densities()* which return 2D numpy arrays with the (log) densities of the observations.

Parameters **pointers** : numpy array (num_states,)

Pointers from HMM states to the correct densities. The length of the array must be equal to the number of states of the HMM and pointing from each state to the corresponding column of the array returned by one of the *log_densities()* or *densities()* methods. The *pointers* type must be np.uint32.

See also:

ObservationModel.log_densities, *ObservationModel.densities*

densities(*self*, *observations*)

Densities (or probabilities) of the observations for each state.

This defaults to computing the exp of the *log_densities*. You can provide a special implementation to speed-up everything.

Parameters **observations** : numpy array

Observations.

Returns numpy array

Densities as a 2D numpy array with the number of rows being equal to the number of observations and the columns representing the different observation log probability densities. The type must be np.float.

log_densities(*self*, *observations*)

Log densities (or probabilities) of the observations for each state.

Parameters **observations** : numpy array

Observations.

Returns numpy array

Log densities as a 2D numpy array with the number of rows being equal to the number of observations and the columns representing the different observation log probability densities. The type must be np.float.

class madmom.ml.hmm.**TransitionModel**

Transition model class for a HMM.

The transition model is defined similar to a scipy compressed sparse row matrix and holds all transition probabilities from one state to an other. This allows an efficient Viterbi decoding of the HMM.

Parameters **states** : numpy array

All states transitioning to state s are stored in: states[pointers[s]:pointers[s+1]]

pointers : numpy array

Pointers for the *states* array for state s.

probabilities : numpy array

The corresponding transition are stored in: probabilities[pointers[s]:pointers[s+1]].

See also:

`scipy.sparse.csr_matrix`

Notes

This class should be either used for loading saved transition models or being sub-classed to define a specific transition model.

classmethod `from_dense`(*cls*, *states*, *prev_states*, *probabilities*)
Instantiate a TransitionModel from dense transitions.

Parameters **states** : numpy array, shape (num_transitions,)

Array with states (i.e. destination states).

prev_states : numpy array, shape (num_transitions,)

Array with previous states (i.e. origination states).

probabilities : numpy array, shape (num_transitions,)

Transition probabilities.

Returns *TransitionModel* instance

TransitionModel instance.

`log_probabilities`
Transition log probabilities.

`make_sparse`(*states*, *prev_states*, *probabilities*)
Return a sparse representation of dense transitions.

This method removes all duplicate states and thus allows an efficient Viterbi decoding of the HMM.

Parameters **states** : numpy array, shape (num_transitions,)

Array with states (i.e. destination states).

prev_states : numpy array, shape (num_transitions,)

Array with previous states (i.e. origination states).

probabilities : numpy array, shape (num_transitions,)

Transition probabilities.

Returns **states** : numpy array

All states transitioning to state s are returned in: states[pointers[s]:pointers[s+1]]

pointers : numpy array

Pointers for the *states* array for state s.

probabilities : numpy array

The corresponding transition are returned in: probabilities[pointers[s]:pointers[s+1]].

See also:

TransitionModel

Notes

Three 1D numpy arrays of same length must be given. The indices correspond to each other, i.e. the first entry of all three arrays define the transition from the state defined prev_states[0] to that defined in states[0] with the probability defined in probabilities[0].

num_states
Number of states.

num_transitions
Number of transitions.

9.1.3 madmom.ml.io

9.1.4 madmom.ml.rnn

This module contains recurrent neural network (RNN) related functionality.

It's main purpose is to serve as a substitute for testing neural networks which were trained by other ML packages or programs without requiring these packages or programs as dependencies.

The only allowed dependencies are Python + numpy + scipy.

The structure reflects just the needed functionality for testing networks. This module is not meant to be a general purpose RNN with lots of functionality. Just use one of the many NN/ML packages out there if you need training or any other stuff.

class madmom.ml.rnn.**BidirectionalLayer**
Bidirectional network layer.

Parameters **fwd_layer** : Layer instance

Forward layer.

bwd_layer : Layer instance

Backward layer.

activate()
Activate the layer.

After activating the *fwd_layer* with the data and the *bwd_layer* with the data in reverse temporal order, the two activations are stacked and returned.

Parameters **data** : numpy array

Activate with this data.

Returns numpy array

Activations for this data.

class madmom.ml.rnn.**Cell**
Cell as used by LSTM units.

Parameters **weights** : numpy array, shape ()

Weights.

bias : scalar or numpy array, shape ()

Bias.

recurrent_weights : numpy array, shape ()

Recurrent weights.

transfer_fn : numpy ufunc, optional

Transfer function.

activate()

Activate the cell / gate with the given data, state (if peephole connections are used) and the output (if recurrent connections are used).

Parameters **data** : scalar or numpy array, shape ()

Input data for the cell.

prev : scalar or numpy array, shape ()

Output data of the previous time step.

state : scalar or numpy array, shape ()

State data of the {current | previous} time step.

Returns numpy array

Activations of the gate for this data.

class madmom.ml.rnn.**FeedForwardLayer**

Feed-forward network layer.

Parameters **weights** : numpy array, shape ()

Weights.

bias : scalar or numpy array, shape ()

Bias.

transfer_fn : numpy ufunc

Transfer function.

activate()

Activate the layer.

Parameters **data** : numpy array

Activate with this data.

Returns numpy array

Activations for this data.

class madmom.ml.rnn.**Gate**

Gate as used by LSTM units.

Parameters **weights** : numpy array, shape ()

Weights.

bias : scalar or numpy array, shape ()

Bias.

recurrent_weights : numpy array, shape ()

Recurrent weights.

peephole_weights : numpy array, shape ()

Peephole weights.

transfer_fn : numpy ufunc, optional

Transfer function.

class madmom.ml.rnn.**LSTMLayer**

Recurrent network layer with Long Short-Term Memory units.

Parameters **weights** : numpy array, shape ()

Weights.

bias : scalar or numpy array, shape ()

Bias.

recurrent_weights : numpy array, shape ()

Recurrent weights.

peephole_weights : numpy array, shape ()

Peephole weights.

transfer_fn : numpy ufunc, optional

Transfer function.

activate()

Activate the LSTM layer.

Parameters **data** : numpy array

Activate with this data.

Returns numpy array

Activations for this data.

madmom.ml.rnn.**RNN**

alias of *RecurrentNeuralNetwork*

class madmom.ml.rnn.**RNNProcessor**

Recurrent Neural Network (RNN) processor class.

Parameters **nn_files** : list

List of files with the RNN models.

num_threads : int, optional

Number of parallel working threads.

add_arguments

Add recurrent neural network testing options to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

nn_files : list

RNN model files.

Returns argparse argument group

Recurrent neural network argument parser group.

class madmom.ml.rnn.**RecurrentLayer**
Recurrent network layer.

Parameters **weights** : numpy array, shape ()

Weights.

bias : scalar or numpy array, shape ()

Bias.

recurrent_weights : numpy array, shape ()

Recurrent weights.

transfer_fn : numpy ufunc

Transfer function.

activate()
Activate the layer.

Parameters **data** : numpy array

Activate with this data.

Returns numpy array

Activations for this data.

class madmom.ml.rnn.**RecurrentNeuralNetwork**
Recurrent Neural Network (RNN) class.

Parameters **layers** : list

Layers of the RNN.

classmethod **load**()
Instantiate a RecurrentNeuralNetwork from a .npz model file.

Parameters **filename** : str

Name of the .npz file with the RNN model.

Returns *RecurrentNeuralNetwork* instance

RNN instance

process()
Process the given data with the RNN.

Parameters **data** : numpy array

Activate the network with this data.

Returns numpy array

Network predictions for this data.

madmom.ml.rnn.**average_predictions**
Returns the average of all predictions.

Parameters **predictions** : list

Predictions (i.e. NN activation functions).

Returns numpy array

Averaged prediction.

madmom.ml.rnn.**linear**
Linear function.

Parameters **x** : numpy array

Input data.

out : numpy array, optional

Array to hold the output data.

Returns numpy array

Unaltered input data.

madmom.ml.rnn.**relu**
Rectified linear (unit) transfer function.

Parameters **x** : numpy array

Input data.

out : numpy array, optional

Array to hold the output data.

Returns numpy array

Rectified linear of input data.

madmom.ml.rnn.**sigmoid**
Logistic sigmoid function.

Parameters **x** : numpy array

Input data.

out : numpy array, optional

Array to hold the output data.

Returns numpy array

Logistic sigmoid of input data.

madmom.ml.rnn.**softmax**
Softmax transfer function.

Parameters **x** : numpy array

Input data.

out : numpy array, optional

Array to hold the output data.

Returns numpy array

Softmax of input data.

CHAPTER 10

madmom.utils

Utility package.

madmom.utils.**suppress_warnings**(*function*)
Decorate the given function to suppress any warnings.

Parameters **function** : function

Function to be decorated.

Returns decorated function

Decorated function.

madmom.utils.**filter_files**(*files*, *suffix*)
Filter the list to contain only files matching the given *suffix*.

Parameters **files** : list

List of files to be filtered.

suffix : str

Return only files matching this suffix.

Returns list

List of files.

madmom.utils.**search_path**(*path*, *recursion_depth=0*)
Returns a list of files in a directory (recursively).

Parameters **path** : str or list

Directory to be searched.

recursion_depth : int, optional

Recursively search sub-directories up to this depth.

Returns list

List of files.

madmom.utils.**search_files**(*files*, *suffix=None*, *recursion_depth=0*)
Returns the files matching the given *suffix*.

Parameters **files** : str or list

File, path or a list thereof to be searched / filtered.

suffix : str, optional

Return only files matching this suffix.

recursion_depth : int, optional

Recursively search sub-directories up to this depth.

Returns list

List of files.

Notes

The list of returned files is sorted.

madmom.utils.**strip_suffix**(*filename*, *suffix=None*)

Strip off the suffix of the given filename or string.

Parameters **filename** : str

Filename or string to strip.

suffix : str, optional

Suffix to be stripped off (e.g. '.txt' including the dot).

Returns str

Filename or string without suffix.

madmom.utils.**match_file**(*filename*, *match_list*, *suffix=None*, *match_suffix=None*)

Match a filename or string against a list of other filenames or strings.

Parameters **filename** : str

Filename or string to strip.

match_list : list

Match to this list of filenames or strings.

suffix : str, optional

Suffix of *filename* to be ignored.

match_suffix

Match only files from *match_list* with this suffix.

Returns list

List of matched files.

madmom.utils.**load_events**(**args*, ***kwargs*)

Load a events from a text file, one floating point number per line.

Parameters **filename** : str or file handle

File to load the events from.

Returns numpy array

Events.

Notes

Comments (lines starting with '#') and additional columns are ignored, i.e. only the first column is returned.

madmom.utils.**write_events**(*events*, *filename*, *fmt='%.3f'*, *header=''*)
Write events to a text file, one floating point number per line.

Parameters **events** : numpy array

Events.

filename : str or file handle

File to write the events to.

fmt : str, optional

How to format the events.

header : str, optional

Header to be written (as a comment).

Returns numpy array

Events.

Notes

This function is just a wrapper to `np.savetxt`, but reorders the arguments in a way it can be used as an *processors.OutputProcessor*.

madmom.utils.**combine_events**(*events*, *delta*)
Combine all events within a certain range.

Parameters **events** : list or numpy array

Events to be combined.

delta : float

Combination delta. All events within this *delta* are combined, i.e. replaced by the mean of the two events.

Returns numpy array

Combined events.

madmom.utils.**quantize_events**(*events*, *fps*, *length=None*, *shift=None*)
Quantize the events with the given resolution.

Parameters **events** : numpy array

Events to be quantized.

fps : float

Quantize with *fps* frames per second.

length : int, optional

Length of the returned array.

shift : float, optional

Shift the events by this value before quantisation

Returns numpy array

Quantized events.

class madmom.utils.**OverrideDefaultListAction**(*sep=None*, **args*, ***kwargs*)

An argparse action that works similarly to the regular 'append' action. The default value is deleted when a new value is specified. The 'append' action would append the new value to the default.

Parameters **sep** : str, optional

Separator to be used if multiple values should be parsed from a list.

madmom.utils.**segment_axis**(*signal*, *frame_size*, *hop_size*, *axis=None*, *end='cut'*, *end_value=0*)

Generate a new array that chops the given array along the given axis into (overlapping) frames.

Parameters **signal** : numpy array

Signal.

frame_size : int

Size of each frame [samples].

hop_size : int

Hop size between adjacent frames [samples].

axis : int, optional

Axis to operate on; if 'None', operate on the flattened array.

end : {'cut', 'wrap', 'pad'}, optional

What to do with the last frame, if the array is not evenly divisible into pieces; possible values:

- 'cut' simply discard the extra values,
- 'wrap' copy values from the beginning of the array,
- 'pad' pad with a constant value.

end_value : float, optional

Value used to pad if *end* is 'pad'.

Returns numpy array, shape (num_frames, frame_size)

Array with overlapping frames

Notes

The array is not copied unless necessary (either because it is unevenly strided and being flattened or because end is set to 'pad' or 'wrap').

The returned array is always of type np.ndarray.

Examples

```
>>> segment_axis(np.arange(10), 4, 2)
array([[0, 1, 2, 3],
       [2, 3, 4, 5],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

10.1 Submodules

10.1.1 madmom.utils.midi

This module contains MIDI functionality.

Almost all code is taken from Giles Hall's python-midi package: https://github.com/vishnubob/python-midi

It combines the complete package in a single file, to make it easier to distribute. Most notable changes are *MIDITrack* and *MIDIFile* classes which handle all data i/o and provide a interface which allows to read/display all notes as simple numpy arrays. Also, the EventRegistry is handled differently.

The last merged commit is 3053fefe.

Since then the following commits have been added functionality-wise:

- 0964c0b (prevent multiple tick conversions)
- c43bf37 (add pitch and value properties to AfterTouchEvent)
- 40111c6 (add 0x08 MetaEvent: ProgramNameEvent)
- 43de818 (handle unknown MIDI meta events gracefully)

Additionally, the module has been updated to work with Python3.

madmom.utils.midi.**byte2int**(*byte*)
Convert a byte-character to an integer.

madmom.utils.midi.**read_variable_length**(*data*)
Read a variable length variable from the given data.

Parameters **data** : bytearray

Data of variable length.

Returns **length** : int

Length in bytes.

madmom.utils.midi.**write_variable_length**(*value*)
Write a variable length variable.

Parameters **value** : bytearray

Value to be encoded as a variable of variable length.

Returns bytearray

Variable with variable length.

class madmom.utils.midi.**EventRegistry**
Class for registering Events.

Event classes should be registered manually by calling EventRegistry.register_event(EventClass) after the class definition.

classmethod register_event(*event*)
Registers an event in the registry.

Parameters **event** : *Event* instance

Event to be registered.

Raises **ValueError**

For unknown events.

class madmom.utils.midi.**AbstractEvent**(***kwargs*)
Abstract Event.

class madmom.utils.midi.**Event**(***kwargs*)
Event.

classmethod is_event(*status_msg*)
Indicates whether the given status message belongs to this event.

Parameters **status_msg** : int

Status message.

Returns bool

True if the given status message belongs to this event.

class madmom.utils.midi.**MetaEvent**(***kwargs*)
MetaEvent is a special subclass of Event that is not meant to be used as a concrete class. It defines a subset of Events known as the Meta events.

classmethod is_event(*status_msg*)
Indicates whether the given status message belongs to this event.

Parameters **status_msg** : int

Status message.

Returns bool

True if the given status message belongs to this event.

class madmom.utils.midi.**MetaEventWithText**(***kwargs*)
Meta Event With Text.

class madmom.utils.midi.**NoteEvent**(***kwargs*)
NoteEvent is a special subclass of Event that is not meant to be used as a concrete class. It defines the generalities of NoteOn and NoteOff events.

pitch
Pitch of the note event.

velocity
Velocity of the note event.

class madmom.utils.midi.**NoteOnEvent**(***kwargs*)
Note On Event.

class madmom.utils.midi.**NoteOffEvent**(***kwargs*)
Note Off Event.

class madmom.utils.midi.**AfterTouchEvent**(***kwargs*)
After Touch Event.

pitch
Pitch of the after touch event.

value
Value of the after touch event.

class madmom.utils.midi.**ControlChangeEvent**(***kwargs*)
Control Change Event.

control
Control ID.

value
Value of the controller.

class madmom.utils.midi.**ProgramChangeEvent**(***kwargs*)
Program Change Event.

value
Value of the Program Change Event.

class madmom.utils.midi.**ChannelAfterTouchEvent**(***kwargs*)
Channel After Touch Event.

value
Value of the Channel After Touch Event.

class madmom.utils.midi.**PitchWheelEvent**(***kwargs*)
Pitch Wheel Event.

pitch
Pitch of the Pitch Wheel Event.

class madmom.utils.midi.**SysExEvent**(***kwargs*)
System Exclusive Event.

classmethod is_event(*status_msg*)
Indicates whether the given status message belongs to this event.

Parameters **status_msg** : int

Status message.

Returns bool

True if the given status message belongs to this event.

class madmom.utils.midi.**SequenceNumberMetaEvent**(***kwargs*)
Sequence Number Meta Event.

class madmom.utils.midi.**TextMetaEvent**(***kwargs*)
Text Meta Event.

class madmom.utils.midi.**CopyrightMetaEvent**(***kwargs*)
Copyright Meta Event.

class madmom.utils.midi.**TrackNameEvent**(***kwargs*)
Track Name Event.

class madmom.utils.midi.**InstrumentNameEvent**(***kwargs*)
Instrument Name Event.

class madmom.utils.midi.**LyricsEvent**(***kwargs*)
Lyrics Event.

class madmom.utils.midi.**MarkerEvent**(***kwargs*)
Marker Event.

class madmom.utils.midi.**CuePointEvent**(***kwargs*)
Cue Point Event.

class madmom.utils.midi.**ProgramNameEvent**(***kwargs*)
Program Name Event.

class madmom.utils.midi.**UnknownMetaEvent**(***kwargs*)
Unknown Meta Event.

The *meta_command* class variable must be set by the constructor of inherited classes.

Parameters **meta_command** : int

Value of the meta command.

class madmom.utils.midi.**ChannelPrefixEvent**(***kwargs*)
Channel Prefix Event.

class madmom.utils.midi.**PortEvent**(***kwargs*)
Port Event.

class madmom.utils.midi.**TrackLoopEvent**(***kwargs*)
Track Loop Event.

class madmom.utils.midi.**EndOfTrackEvent**(***kwargs*)
End Of Track Event.

class madmom.utils.midi.**SetTempoEvent**(***kwargs*)
Set Tempo Event.

microseconds_per_quarter_note
Microseconds per quarter note.

class madmom.utils.midi.**SmpteOffsetEvent**(***kwargs*)
SMPTE Offset Event.

class madmom.utils.midi.**TimeSignatureEvent**(***kwargs*)
Time Signature Event.

numerator
Numerator of the time signature.

denominator
Denominator of the time signature.

metronome
Metronome.

thirty_seconds
Thirty-seconds of the time signature.

class madmom.utils.midi.**KeySignatureEvent**(***kwargs*)
Key Signature Event.

alternatives
Alternatives of the key signature.

minor
Major / minor.

class madmom.utils.midi.**SequencerSpecificEvent**(***kwargs*)
Sequencer Specific Event.

class madmom.utils.midi.**MIDITrack**(*events=None*)
MIDI Track.

Parameters **events** : list

MIDI events.

data_stream
MIDI data stream representation of the track.

classmethod **from_file**(*midi_stream*)
Create a MIDI track by reading the data from a stream.

Parameters **midi_stream** : open file handle

MIDI file stream (e.g. open MIDI file handle)

Returns *MIDITrack* instance

MIDITrack instance

classmethod **from_notes**(*notes*, *resolution=480*)
Create a MIDI track from the given notes.

Parameters **notes** : numpy array

Array with the notes, one per row. The columns must be: (onset time, pitch, duration, velocity).

resolution : int

Resolution (i.e. microseconds per quarter note) of the MIDI track.

Returns *MIDITrack* instance

MIDITrack instance

class madmom.utils.midi.**MIDIFile**(*tracks=None*, *resolution=480*, *file_format=0*)
MIDI File.

Parameters **tracks** : list

List of *MIDITrack* instances.

resolution : int, optional

Resolution (i.e. microseconds per quarter note).

file_format : int, optional

Format of the MIDI file.

ticks_per_quarter_note
Number of ticks per quarter note.

tempi()
Tempi of the MIDI file.

Returns **tempi** : numpy array

Array with tempi (tick, seconds per tick, cumulative time).

time_signatures()
Time signatures of the MIDI file.

Returns **time_signatures** : numpy array

Array with time signatures (tick, numerator, denominator).

notes(*note_time_unit='s'*)
Notes of the MIDI file.

Parameters **note_time_unit** : {'s', 'b'}

Time unit for notes, seconds ('s') or beats ('b').

Returns **notes** : numpy array

Array with notes (onset time, pitch, duration, velocity).

data_stream
MIDI data stream representation of the MIDI file.

write(*midi_file*)
Write a MIDI file.

Parameters **midi_file** : str

The MIDI file name.

classmethod **from_file**(*midi_file*)
Create a MIDI file instance from a .mid file.

Parameters **midi_file** : str

Name of the .mid file to load.

Returns *MIDIFile* instance

MIDIFile instance

classmethod **from_notes**(*notes*)
Create a MIDIFile instance from a numpy array with notes.

Parameters **notes** : numpy array or list of tuples

Notes (onset, pitch, offset, velocity).

Returns *MIDIFile* instance

MIDIFile instance with all notes collected in one track.

static **add_arguments**(*parser*, *length=None*, *velocity=None*)
Add MIDI related arguments to an existing parser object.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

length : float, optional

Default length of the notes [seconds].

velocity : int, optional

Default velocity of the notes.

Returns argparse argument group

MIDI argument parser group object.

madmom.utils.midi.**process_notes**(*data*, *output=None*)

This is a simple processing function. It either loads the notes from a MIDI file and or writes the notes to a file.

The behaviour depends on the presence of the *output* argument, if 'None' is given, the notes are read, otherwise the notes are written to file.

Parameters **data** : str or numpy array

MIDI file to be loaded (if *output* is 'None') / notes to be written.

output : str, optional

Output file name. If set, the notes given by *data* are written.

Returns **notes** : numpy array

Notes read/written.

10.1.2 madmom.utils.stats

CHAPTER 11

madmom.processors

This module contains all processor related functionality.

11.1 Notes

All features should be implemented as classes which inherit from Processor (or provide a XYZProcessor(Processor) variant). This way, multiple Processor objects can be chained/combined to achieve the wanted functionality.

class madmom.processors.**Processor**
Abstract base class for processing data.

static load(*infile*)
Instantiate a new Processor from a file.

This method un-pickles a saved Processor object. Subclasses should overwrite this method with a better performing solution if speed is an issue.

Parameters **infile** : str or file handle

Pickled processor.

Returns *Processor* instance

Processor.

dump(*outfile*)
Save the Processor to a file.

This method pickles a Processor object and saves it. Subclasses should overwrite this method with a better performing solution if speed is an issue.

Parameters **outfile** : str or file handle

Output file for pickling the processor.

process(*data*)
Process the data.

This method must be implemented by the derived class and should process the given data and return the processed output.

Parameters **data** : depends on the implementation of subclass

Data to be processed.

Returns depends on the implementation of subclass

Processed data.

class madmom.processors.**OutputProcessor**
Class for processing data and/or feeding it into some sort of output.

process(*data*, *output*)
Processes the data and feed it to the output.

This method must be implemented by the derived class and should process the given data and return the processed output.

Parameters **data** : depends on the implementation of subclass

Data to be processed (e.g. written to file).

output : str or file handle

Output file name or file handle.

Returns depends on the implementation of subclass

Processed data.

class madmom.processors.**SequentialProcessor**(*processors*)
Processor class for sequential processing of data.

Parameters **processors** : list

Processor instances to be processed sequentially.

Notes

If the *processors* list contains lists or tuples, these get wrapped as a SequentialProcessor itself.

insert(*index*, *processor*)
Insert a Processor at the given processing chain position.

Parameters **index** : int

Position inside the processing chain.

processor : *Processor*

Processor to insert.

append(*other*)
Append another Processor to the processing chain.

Parameters **other** : *Processor*

Processor to append to the processing chain.

extend(*other*)
Extend the processing chain with a list of Processors.

Parameters **other** : list

Processors to be appended to the processing chain.

process(*data*)
Process the data sequentially with the defined processing chain.

Parameters **data** : depends on the first processor of the processing chain

Data to be processed.

Returns depends on the last processor of the processing chain

Processed data.

class madmom.processors.**ParallelProcessor**(*processors*, *num_threads=1*)

Processor class for parallel processing of data.

Parameters **processors** : list

Processor instances to be processed in parallel.

num_threads : int, optional

Number of parallel working threads.

Notes

If the *processors* list contains lists or tuples, these get wrapped as a *SequentialProcessor*.

process(*data*)

Process the data in parallel.

Parameters **data** : depends on the processors

Data to be processed.

Returns list

Processed data.

classmethod **add_arguments**(*parser*, *num_threads=1*)

Add parallel processing options to an existing parser object.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

num_threads : int, optional

Number of parallel working threads.

Returns argparse argument group

Parallel processing argument parser group.

Notes

The group is only returned if only if *num_threads* is not 'None'. Setting it smaller or equal to 0 sets it the number of CPU cores.

class madmom.processors.**IOProcessor**(*in_processor*, *out_processor=None*)

Input/Output Processor which processes the input data with the input processor and pipes everything into the given output processor.

All Processors defined in the input chain are sequentially called with the 'data' argument only. The output Processor is the only one ever called with two arguments ('data', 'output').

Parameters **in_processor** : *Processor*, function, tuple or list

Input processor. Can be a *Processor* (or subclass thereof like *SequentialProcessor* or *ParallelProcessor*), a function accepting a single argument ('data'). If a tuple or list is given, it is wrapped as a *SequentialProcessor*.

out_processor : *OutputProcessor*, function, tuple or list

OutputProcessor or function accepting two arguments ('data', 'output'). If a tuple or list is given, it is wrapped in an *IOProcessor* itself with the last element regarded as the *out_processor* and all others as *in_processor*.

process(*data*, *output=None*)

Processes the data with the input processor and pipe everything into the output processor, which also pipes it to *output*.

Parameters **data** : depends on the input processors

Data to be processed.

output: str or file handle

Output file (handle).

Returns depends on the output processors

Processed data.

madmom.processors.**process_single**(*processor*, *infile*, *outfile*, ***kwargs*)

Process a single file with the given Processor.

Parameters **processor** : *Processor* instance

Processor to be processed.

infile : str or file handle

Input file (handle).

outfile : str or file handle

Output file (handle).

madmom.processors.**process_batch**(*processor*, *files*, *output_dir=None*, *output_suffix=None*, *strip_ext=True*, *num_workers=4*, ***kwargs*)

Process a list of files with the given Processor in batch mode.

Parameters **processor** : *Processor* instance

Processor to be processed.

files : list

Input file(s) (handles).

output_dir : str, optional

Output directory.

output_suffix : str, optional

Output suffix (e.g. '.txt' including the dot).

strip_ext : bool, optional

Strip off the extension from the input files.

num_workers : int, optional

Number of parallel working threads.

Notes

Either *output_dir* and/or *output_suffix* must be set. If *strip_ext* is True, the extension of the input file names is stripped off before the *output_suffix* is appended to the input file names.

madmom.processors.**pickle_processor**(*processor*, *outfile*, ***kwargs*)
Pickle the Processor to a file.

Parameters **processor** : *Processor* instance

Processor to be pickled.

outfile : str or file handle

Output file (handle) where to pickle it.

madmom.processors.**io_arguments**(*parser*, *output_suffix='.txt'*, *pickle=True*)
Add input / output related arguments to an existing parser.

Parameters **parser** : argparse parser instance

Existing argparse parser object.

output_suffix : str, optional

Suffix appended to the output files.

pickle : bool, optional

Add a 'pickle' sub-parser to the parser?

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

CHAPTER 13

Acknowledgements

Supported by the European Commission through the GiantSteps project (FP7 grant agreement no. 610591) and the Phenicx project (FP7 grant agreement no. 601166) as well as the Austrian Science Fund (FWF) project Z159.

[R1] Sebastian Böck and Gerhard Widmer "Maximum Filter Vibrato Suppression for Onset Detection" Proceedings of the 16th International Conference on Digital Audio Effects (DAFx), 2013.

[R13] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[R14] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[R15] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[R16] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R18] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[R19] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R21] Filip Korzeniowski, Sebastian Böck and Gerhard Widmer, "Probabilistic Extraction of Beat Positions from a Beat Activation Function", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[R22] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[R23] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R24] Florian Krebs, Sebastian Böck and Gerhard Widmer, "Rhythmic Pattern Modeling for Beat and Downbeat Tracking in Musical Audio", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[R25] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R26] Filip Korzeniowski, Sebastian Böck and Gerhard Widmer, "Probabilistic Extraction of Beat Positions from a Beat Activation Function", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[R27] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R28] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R29] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R30] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R31] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R32] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R33] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[R34] Florian Krebs, Sebastian Böck and Gerhard Widmer, "Rhythmic Pattern Modeling for Beat and Downbeat Tracking in Musical Audio", Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[R35] Paul Masri, "Computer Modeling of Sound for Transformation and Synthesis of Musical Signals", PhD thesis, University of Bristol, 1996.

[R36] Chris Duxbury, Mark Sandler and Matthew Davis, "A hybrid approach to musical note onset detection", Proceedings of the 5th International Conference on Digital Audio Effects (DAFx), 2002.

[R37] Paul Masri, "Computer Modeling of Sound for Transformation and Synthesis of Musical Signals", PhD thesis, University of Bristol, 1996.

[R38] Sebastian Böck and Gerhard Widmer, "Maximum Filter Vibrato Suppression for Onset Detection", Proceedings of the 16th International Conference on Digital Audio Effects (DAFx), 2013.

[R39] Sebastian Böck and Gerhard Widmer, "Local group delay based vibrato and tremolo suppression for onset detection", Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[R40] Paul Brossier, "Automatic Annotation of Musical Audio for Interactive Applications", PhD thesis, Queen Mary University of London, 2006.

[R41] Stephen Hainsworth and Malcolm Macleod, "Onset Detection in Musical Audio Signals", Proceedings of the International Computer Music Conference (ICMC), 2003.

[R42] Juan Pablo Bello, Chris Duxbury, Matthew Davies and Mark Sandler, "On the use of phase and energy for musical onset detection in the complex domain", IEEE Signal Processing Letters, Volume 11, Number 6, 2004.

[R43] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[R44] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[R45] Juan Pablo Bello, Chris Duxbury, Matthew Davies and Mark Sandler, "On the use of phase and energy for musical onset detection in the complex domain", IEEE Signal Processing Letters, Volume 11, Number 6, 2004.

[R46] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[R47] Sebastian Böck, Florian Krebs and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods", Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[R48] Sebastian Böck, Florian Krebs and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods", Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[R49] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[R50] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R2] Arshia Cont, Diemo Schwarz, Norbert Schnell and Christopher Raphael, "Evaluation of Real-Time Audio-to-Score Alignment", Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR), 2007.

[R3] Matthew E. P. Davies, Norberto Degara, and Mark D. Plumbley, "Evaluation Methods for Musical Audio Beat Tracking Algorithms", Technical Report C4DM-TR-09-06, Centre for Digital Music, Queen Mary University of London, 2009.

[R4] M. McKinney, D. Moelants, M. Davies and A. Klapuri, "Evaluation of audio beat tracking and music tempo extraction algorithms", Journal of New Music Research, vol. 36, no. 1, 2007.

[R5] A.T. Cemgil, B. Kappen, P. Desain, and H. Honing, "On tempo tracking: Tempogram representation and Kalman filtering", Journal Of New Music Research, vol. 28, no. 4, 2001.

[R6] M. Goto and Y. Muraoka, "Issues in evaluating beat tracking systems", Working Notes of the IJCAI-97 Workshop on Issues in AI and Music - Evaluation and Assessment, 1997.

[R7] Matthew E. P. Davies, Norberto Degara, and Mark D. Plumbley, "Evaluation Methods for Musical Audio Beat Tracking Algorithms", Technical Report C4DM-TR-09-06, Centre for Digital Music, Queen Mary University of London, 2009.

[R8] S. Hainsworth, "Techniques for the automated analysis of musical audio", PhD. dissertation, Department of Engineering, Cambridge University, 2004.

[R9] A.P. Klapuri, A. Eronen, and J. Astola, "Analysis of the meter of acoustic musical signals", IEEE Transactions on Audio, Speech and Language Processing, vol. 14, no. 1, 2006.

[R10] M. E.P. Davies, N. Degara and M. D. Plumbley, "Measuring the performance of beat tracking algorithms algorithms using a beat error histogram", IEEE Signal Processing Letters, vol. 18, vo. 3, 2011.

[R11] Sebastian Böck, Florian Krebs and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods", Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[R12] M. McKinney, D. Moelants, M. Davies and A. Klapuri, “Evaluation of audio beat tracking and music tempo extraction algorithms”, Journal of New Music Research, vol. 36, no. 1, 2007.

m

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W