audiotsm Documentation

Release 0.1.0

Author

Sep 21, 2017

Contents

AudioTSM is a python library for real-time audio time-scale modification procedures, i.e. algorithms that change the speed of an audio signal without changing its pitch.

Documentation: https://audiotsm.readthedocs.io/

Source code repository and issue tracker: https://github.com/Muges/audiotsm/

Python Package Index: https://pypi.python.org/pypi/audiotsm/

License: MIT – see the file `LICENSE` for details.

CHAPTER 1

Installation

Audiotsm should work with python 2.7 and python 3.4+.

You can install the latest version of audiotsm with pip:

```
pip install audiotsm
```

You may also need to install the sounddevice library in order to run the examples or to use a `StreamWriter`:

```
pip install sounddevice
```

CHAPTER 2

Basic usage

The audiotsm package implements several time-scale modification procedures:

- OLA (Overlap-Add), which should only be used for percussive audio signals;
- WSOLA (Waveform Similarity-based Overlap-Add), an amelioration of the OLA procedure which should give good results on most inputs.

Below is a basic example showing how to reduce the speed of a wav file by half using the WSOLA procedure:

```
from audiotsm import wsola
from audiotsm.io.wav import WavReader, WavWriter

with WavReader(input_filename) as reader:
    with WavWriter(output_filename, reader.channels, reader.samplerate) as writer:
        tsm = wsola(reader.channels, speed=0.5)
        tsm.run(reader, writer)
```

CHAPTER 3

Thanks

If you are interested in time-scale modification procedures, I highly recommend reading A Review of Time-Scale Modification of Music Signals by Jonathan Driedger and Meinard Müller.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

CHAPTER 5

Time-Scale Modification

Time-Scale Modification procedures

The *audiotsm* module provides several time-scale modification procedures:

- *ola()* (Overlap-Add), which should only be used for percussive audio signals;
- *wsola()* (Waveform Similarity-based Overlap-Add), which should give good results on most inputs.

Note: If you are not sure which procedure and parameters to use, using *wsola()* with the default parameters should work in most cases.

Each of the function of this module returns a *TSM* object which implements a time-scale modification procedure.

audiotsm.**ola**(*channels*, *speed=1.0*, *frame_length=256*, *analysis_hop=None*, *synthesis_hop=None*)

Returns a *TSM* object implementing the OLA (Overlap-Add) time-scale modification procedure.

In most cases, you should not need to set the frame_length, the analysis_hop or the synthesis_hop. If you want to fine tune these parameters, you can check the documentation of the *AnalysisSynthesisTSM* class to see what they represent.

Parameters

- **channels** (*int*) – the number of channels of the input signal.
- **speed** (*float, optional*) – the speed ratio by which the speed of the signal will be multiplied (for example, if speed is set to 0.5, the output signal will be half as fast as the input signal).
- **frame_length** (*int, optional*) – the length of the frames.
- **analysis_hop** (*int, optional*) – the number of samples between two consecutive analysis frames (speed * synthesis_hop by default). If analysis_hop is set, the speed parameter will be ignored.

- **synthesis_hop** (*int, optional*) – the number of samples between two consecutive synthesis frames (`frame_length // 2` by default).

Returns a *audiotsm.base.tsm.TSM* object

`audiotsm.`**`wsola`**(*channels*, *speed=1.0*, *frame_length=1024*, *analysis_hop=None*, *synthesis_hop=None*, *tolerance=None*)

Returns a *TSM* object implementing the WSOLA (Waveform Similarity-based Overlap-Add) time-scale modification procedure.

In most cases, you should not need to set the `frame_length`, the `analysis_hop`, the `synthesis_hop`, or the `tolerance`. If you want to fine tune these parameters, you can check the documentation of the *AnalysisSynthesisTSM* class to see what the first three represent.

WSOLA works in the same way as OLA, with the exception that it allows slight shift (at most `tolerance`) of the position of the analysis frames.

Parameters

- **channels** (*int*) – the number of channels of the input signal.
- **speed** (*float, optional*) – the speed ratio by which the speed of the signal will be multiplied (for example, if `speed` is set to 0.5, the output signal will be half as fast as the input signal).
- **frame_length** (*int, optional*) – the length of the frames.
- **analysis_hop** (*int, optional*) – the number of samples between two consecutive analysis frames (`speed * synthesis_hop` by default). If `analysis_hop` is set, the `speed` parameter will be ignored.
- **synthesis_hop** (*int, optional*) – the number of samples between two consecutive synthesis frames (`frame_length // 2` by default).
- **tolerance** (*int*) – the maximum number of samples that the analysis frame can be shifted.

Returns a *audiotsm.base.tsm.TSM* object

TSM Object

The *audiotsm.base.tsm* module provides an abstract class for real-time audio time-scale modification procedures.

class `audiotsm.base.tsm.`**`TSM`**

An abstract class for real-time audio time-scale modification procedures.

If you want to use a *TSM* object to run a TSM procedure on a signal, you should use the *run()* method in most cases.

`clear`()

Clears the state of the *TSM* object, making it ready to be used on another signal (or another part of a signal).

This method should be called before processing a new file, or seeking to another part of a signal.

`flush_to`(*writer*)

Writes as many output samples as possible to `writer`, assuming that there are no remaining samples that will be added to the input (i.e. that the *write_to()* method will not be called), and returns the number of samples that were written.

Parameters **writer** – a *audiotsm.io.base.Writer*.

Returns

a tuple (`n`, `finished`), with:

- `n` the number of samples that were written to `writer`
- `finished` a boolean that is `True` when there are no samples remaining to flush.

Return type (int, bool)

`read_from`(*reader*)

Reads as many samples as possible from `reader`, processes them, and returns the number of samples that were read.

Parameters **`reader`** – a *`audiotsm.io.base.Reader`*.

Returns the number of samples that were read from `reader`.

`run`(*reader*, *writer*)

Runs the TSM procedure on the content of `reader` and writes the output to `writer`.

Parameters

- **`reader`** – a *`audiotsm.io.base.Reader`*.
- **`writer`** – a *`audiotsm.io.base.Writer`*.

`set_speed`(*speed*)

Sets the speed ratio.

Parameters **`speed`** (*`float`*) – the speed ratio by which the speed of the signal will be multiplied (for example, if `speed` is set to 0.5, the output signal will be half as fast as the input signal).

`write_to`(*writer*)

Writes as many result samples as possible to `writer`.

Parameters **`writer`** – a *`audiotsm.io.base.Writer`*.

Returns

a tuple (`n`, `finished`), with:

- `n` the number of samples that were written to `writer`
- `finished` a boolean that is `True` when there are no samples remaining to write. In this case, the *`read_from()`* method should be called to add new input samples, or, if there are no remaining input samples, the *`flush_to()`* method should be called to get the last output samples.

Return type (int, bool)

CHAPTER 6

Readers and Writers

TSM objects use *Reader* objects as input and *Writer* objects as output.

The *audiotsm.io* package provides Readers and Writers allowing to use *numpy arrays* or *wav files* as input or output of a *TSM*, to play the output in real-time, as well as base classes to implement your own Readers and Writers.

Numpy arrays

The *audiotsm.io.array* module provides a Reader and Writers allowing to use a `numpy.ndarray` as input or output of a *TSM* object.

class `audiotsm.io.array.`**`ArrayReader`**`(`*data*`)`

Bases: *audiotsm.io.base.Reader*

A *Reader* allowing to use `numpy.ndarray` as input of a *TSM* object.

Parameters **`data`** (`numpy.ndarray`) – a matrix of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer, where the samples will be read.

class `audiotsm.io.array.`**`ArrayWriter`**`(`*channels*`)`

Bases: *audiotsm.io.base.Writer*

A *Writer* allowing to get the output of a *TSM* object as a `numpy.ndarray`.

Writing to an *ArrayWriter* will add the data at the end of the *data* attribute.

Parameters **`channels`** (*int*) – the number of channels of the signal.

`data`

A `numpy.ndarray` of shape (`m`, `n`), with `m` the number of channels and `n` the length of the data, where the samples have written.

class `audiotsm.io.array.`**`FixedArrayWriter`**`(`*data*`)`

Bases: *audiotsm.io.base.Writer*

A *Writer* allowing to use `numpy.ndarray` as output of a TSM object.

Contrary to an *ArrayWriter*, a *FixedArrayWriter* takes the buffer in which the data will be written as a parameter of its constructor. The buffer is of fixed size, and it will not be possible to write more samples to the *FixedArrayWriter* than the buffer can contain.

Parameters **data** (numpy.ndarray) – a matrix of shape (m, n), with m the number of channels and n the length of the buffer, where the samples will be written.

Wav files

The *audiotsm.io.wav* module provides a *Reader* and a *Writer* allowing to use wav files as input or output of a *TSM* object.

class audiotsm.io.wav.**WavReader**(*filename*)

Bases: *audiotsm.io.base.Reader*

A *Reader* allowing to use a wav file as input of a *TSM* object.

You should close the *WavReader* after using it with the *close()* method, or use it in a with statement as follow:

```
with WavReader(filename) as reader:
    # use reader...
```

Parameters **filename** (*str*) – the name of an existing wav file.

close()

Close the wav file.

samplerate

The samplerate of the wav file.

samplewidth

The sample width in bytes of the wav file.

class audiotsm.io.wav.**WavWriter**(*filename*, *channels*, *samplerate*)

Bases: *audiotsm.io.base.Writer*

A *Writer* allowing to use a wav file as output of a *TSM* object.

You should close the *WavWriter* after using it with the *close()* method, or use it in a with statement as follow:

```
with WavWriter(filename, 2, 44100) as writer:
    # use writer...
```

Parameters

- **filename** (*str*) – the name of the wav file (it will be overwritten if it already exists).
- **channels** (*int*) – the number of channels of the signal.
- **samplerate** (*int*) – the sampling rate of the signal.

close()

Close the wav file.

Play in real-time

Implementing your own

The *audiotsm.io.base* module provides base classes for the input and output of *TSM* objects.

class audiotsm.io.base.**Reader**

An abstract class for the input of a *TSM* object.

channels

The number of channels of the *Reader*.

empty

True if there is no more data to read.

read(*buffer*)

Reads as many samples from the *Reader* as possible, write them to buffer, and returns the number of samples that were read.

Parameters buffer (numpy.ndarray) – a matrix of shape (m, n), with m the number of channels and n the length of the buffer, where the samples will be written.

Returns the number of samples that were read. It should always be equal to the length of the buffer, except when there is no more values to be read.

Raises ValueError – if the *Reader* and the buffer do not have the same number of channels

skip(*n*)

Try to skip n samples, an returns the number of samples that were actually skipped.

class audiotsm.io.base.**Writer**

An abstract class for the output of a *TSM* object.

channels

The number of channels of the *Writer*.

write(*buffer*)

Write as many samples from the *Writer* as possible from buffer, and returns the number of samples that were written.

Parameters buffer (numpy.ndarray) – a matrix of shape (m, n), with m the number of channels and n the length of the buffer, where the samples will be read.

Returns the number of samples that were written. It should always be equal to the length of the buffer, except when there is no more space in the *Writer*.

Raises ValueError – if the *Writer* and the buffer do not have the same number of channels

CHAPTER 7

Internal API

Analysis-Synthesis based TSM procedures

The `audiotsm.base.analysis_synthesis` module provides a base class for real-time analysis-synthesis based audio time-scale modification procedures.

class `audiotsm.base.analysis_synthesis.`**`AnalysisSynthesisTSM`**(*converter*, *channels*, *frame_length*, *analysis_hop*, *synthesis_hop*, *analysis_window*, *synthesis_window*, *delta_before=0*, *delta_after=0*)

A `audiotsm.base.tsm.TSM` for real-time analysis-synthesis based time-scale modification procedures.

The basic principle of an analysis-synthesis based TSM procedure is to first decompose the input signal into short overlapping frames, called the analysis frames. The frames have a fixed length `frame_length`, and are separated by `analysis_hop` samples, as illustrated below:

```
          <--------frame_length--------><-analysis_hop->
Frame 1:  [~~~~~~~~~~~~~~~~~~~~~~~~~~~~]
Frame 2:                  [~~~~~~~~~~~~~~~~~~~~~~~~~~~~]
Frame 3:                                  [~~~~~~~~~~~~~~~~~~~~~~~~~~~~]
...
```

It then relocates the frames on the time axis by changing the distance between them (to `synthesis_hop`), as illustrated below:

```
          <--------frame_length--------><----synthesis_hop---->
Frame 1:  [~~~~~~~~~~~~~~~~~~~~~~~~~~~~]
Frame 2:                         [~~~~~~~~~~~~~~~~~~~~~~~~~~~~]
Frame 3:                                                [~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~]
...
```

This changes the speed of the signal by the ratio `analysis_hop / synthesis_hop` (for example, if the `synthesis_hop` is twice the `analysis_hop`, the output signal will be half as fast as the input signal).

However this simple method introduces artifacts to the signal. These artifacts can be reduced by modifying the analysis frames by various methods. This is done by a `converter` object, which converts the analysis frames into modified frames called the synthesis frames.

To further reduce the artifacts, window functions (the `analysis_window` and the `synthesis_window`) can be applied to the analysis frames and the synthesis frames in order to smooth the signal.

Some TSM procedures (e.g. WSOLA-like methods) may need to have access to some samples preceeding or following an analysis frame to generate the synthesis frame. The *delta_before* and *delta_after* parameters allow to specify the numbers of samples needed before and after the analysis frame, so that they are available to the `converter`.

For more details on Time-Scale Modification procedures, I recommend reading "A Review of Time-Scale Modification of Music Signals" by Jonathan Driedger and Meinard Müller.

Parameters

- **`converter`** (*`Converter`*) – an object that implements the conversion of the analysis frames into synthesis frames.
- **`channels`** (*`int`*) – the number of channels of the input signal.
- **`frame_length`** (*`int`*) – the length of the frames.
- **`analysis_hop`** (*`int`*) – the number of samples between two consecutive analysis frames.
- **`synthesis_hop`** (*`int`*) – the number of samples between two consecutive synthesis frames.
- **`analysis_window`** (`numpy.ndarray`) – a window applied to the analysis frames
- **`synthesis_window`** (`numpy.ndarray`) – a window applied to the synthesis frames
- **`delta_before`** (*`int`*) – the number of samples preceding an analysis frame that the converter requires (this is usually 0, except for WSOLA-like methods)
- **`delta_after`** (*`int`*) – the number of samples following an analysis frame that the converter requires (this is usually 0, except for WSOLA-like methods)

class `audiotsm.base.analysis_synthesis.`**`Converter`**

A base class for objects implementing the conversion of analysis frames into synthesis frames.

`clear``()`

Clears the state of the Converter, making it ready to be used on another signal (or another part of a signal). It is called by the *`clear()`* method and the constructor of *`AnalysisSynthesisTSM`*.

`convert_frame``(`*analysis_frame*`)`

Converts an analysis frame into a synthesis frame.

Parameters **`analysis_frame`** (`numpy.ndarray`) – a matrix of shape (`m`, `delta_before + frame_length + delta_after`), with `m` the number of channels, containing the analysis frame and some samples before and after (as specified by the `delta_before` and `delta_after` parameters of the *`AnalysisSynthesisTSM`* calling the *`Converter`*).

`analysis_frame[:, delta_before:-delta_after]` contains the actual analysis frame (without the samples preceeding and following it).

Returns a synthesis frame represented as a `numpy.ndarray` of shape (`m`, `frame_length`), with `m` the number of channels.

Circular buffers

The *audiotsm.utils* module provides utility functions and classes used in the implementation of time-scale modification procedures.

class audiotsm.utils.**CBuffer**(*channels*, *max_length*)

A *CBuffer* is a circular buffer used to store multichannel audio data.

It can be seen as a variable-size buffer whose length is bounded by `max_length`. The *CBuffer.write()* and *CBuffer.right_pad()* methods allow to add samples at the end of the buffer, while the *CBuffer.read()* and *CBuffer.remove()* methods allow to remove samples from the beginning of the buffer.

Contrary to the samples added by the *CBuffer.write()* and *CBuffer.read_from()*, those added by the *CBuffer.right_pad()* method are considered not to be ready to be read. Effectively, this means that they can be modified by the *CBuffer.add()* and *CBuffer.divide()* methods, but have to be marked as ready to be read with the *CBuffer.set_ready()* method before being read with the *CBuffer.peek()*, *CBuffer.read()*, or *CBuffer.write_to()* methods.

Parameters

- **channels** (*int*) – the number of channels of the buffer.
- **max_length** (*int*) – the maximum length of the buffer (i.e. the maximum number of samples that can be stored in each channel).

add(*buffer*)

Adds a `buffer` element-wise to the *CBuffer*.

Parameters **buffer** (`numpy.ndarray`) – a matrix of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer.

Raises **ValueError** – if the *CBuffer* and the `buffer` do not have the same number of channels or the *CBuffer* is smaller than the `buffer` (`self.length < n`).

divide(*array*)

Divides each channel of the *CBuffer* element-wise by the `array`.

Parameters **array** (`numpy.ndarray`) – an array of shape (`n`,).

Raises **ValueError** – if the length of the *CBuffer* is smaller than the length of the array (`self.length < n`).

length

The number of samples of each channel of the *CBuffer*.

peek(*buffer*)

Reads as many samples from the *CBuffer* as possible, without removing them from the *CBuffer*, writes them to the `buffer`, and returns the number of samples that were read.

The samples need to be marked as ready to be read with the *CBuffer.set_ready()* method in order to be read. This is done automatically by the *CBuffer.write()* and *CBuffer.read_from()* methods.

Parameters **buffer** (`numpy.ndarray`) – a matrix of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer, where the samples will be written.

Returns the number of samples that were read from the *CBuffer*.

Raises **ValueError** – if the *CBuffer* and the `buffer` do not have the same number of channels.

read(*buffer*)

Reads as many samples from the `CBuffer` as possible, removes them from the `CBuffer`, writes them to the `buffer`, and returns the number of samples that were read.

The samples need to be marked as ready to be read with the `CBuffer.set_ready()` method in order to be read. This is done automatically by the `CBuffer.write()` and `CBuffer.read_from()` methods.

Parameters **buffer** (`numpy.ndarray`) – a matrix of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer, where the samples will be written.

Returns the number of samples that were read from the `CBuffer`.

Raises **ValueError** – if the `CBuffer` and the `buffer` do not have the same number of channels.

read_from(*reader*)

Reads as many samples as possible from `reader`, writes them to the `CBuffer`, and returns the number of samples that were read.

The written samples are marked as ready to be read.

Parameters **reader** – a `audiotsm.io.base.Reader`.

Returns the number of samples that were read from `reader`.

Raises **ValueError** – if the `CBuffer` and `reader` do not have the same number of channels.

ready

The number of samples that can be read.

remaining_length

The number of samples that can be added to the `CBuffer`.

remove(*n*)

Removes the first `n` samples of the `CBuffer`, preventing them to be read again, and leaving more space for new samples to be written.

Parameters **n** (`int`) – the number of samples to remove.

Returns the number of samples that were removed.

right_pad(*n*)

Add zeros at the end of the `CBuffer`.

The added samples are not marked as ready to be read. The `CBuffer.set_ready()` will need to be called in order to be able to read them.

Parameters **n** (`int`) – the number of zeros to add.

Raises **ValueError** – if there is not enough space to add the zeros.

set_ready(*n*)

Mark the next `n` samples as ready to be read.

Parameters **n** (`int`) – the number of samples to mark as ready to be read.

Raises **ValueError** – if there is less than `n` samples that are not ready yet.

to_array()

Returns an array containing the same data as the `CBuffer`.

Returns a `numpy.ndarray` of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer.

write (*buffer*)

Writes as many samples from the `buffer` to the *CBuffer* as possible, and returns the number of samples that were read.

The written samples are marked as ready to be read.

Parameters **buffer** (`numpy.ndarray`) – a matrix of shape (`m`, `n`), with `m` the number of channels and `n` the length of the buffer, where the samples will be read.

Returns the number of samples that were written to the *CBuffer*.

Raises **ValueError** – if the *CBuffer* and the `buffer` do not have the same number of channels.

write_to (*writer*)

Writes as many samples as possible to `writer`, deletes them from the *CBuffer*, and returns the number of samples that were written.

The samples need to be marked as ready to be read with the *CBuffer.set_ready()* method in order to be read. This is done automatically by the *CBuffer.write()* and *CBuffer.read_from()* methods.

Parameters **writer** – a *audiotsm.io.base.Writer*.

Returns the number of samples that were written to `writer`.

Raises **ValueError** – if the *CBuffer* and `writer` do not have the same number of channels.

class audiotsm.utils.**NormalizeBuffer** (*length*)

A *NormalizeBuffer* is a mono-channel circular buffer, used to normalize audio buffers.

Parameters **length** (*int*) – the length of the *NormalizeBuffer*.

add (*window*)

Adds a window element-wise to the *NormalizeBuffer*.

Parameters **window** (`numpy.ndarray`) – an array of shape (`n`,).

Raises **ValueError** – if the window is larger than the buffer (`n > self.length`).

length

The length of the CBuffer.

remove (*n*)

Removes the first `n` values of the *NormalizeBuffer*.

Parameters **n** (*int*) – the number of values to remove.

to_array (*start=0*, *end=None*)

Returns an array containing the same data as the *NormalizeBuffer*, from index `start` (included) to index `end` (exluded).

Returns `numpy.ndarray`

Window functions

The *audiotsm.utils.windows* module contains window functions used for digital signal processing.

audiotsm.utils.windows.**apply** (*buffer*, *window*)

Applies a window to a buffer.

Parameters

- **buffer** (numpy.ndarray) – a matrix of shape (m, n), with m the number of channels and n the length of the buffer.
- **window** – a numpy.ndarray of shape (n,).

audiotsm.utils.windows.**hanning**(*length*)

Returns a periodic Hanning window.

Contrary to numpy.hanning(), which returns the symetric Hanning window, *hanning()* returns a periodic Hanning window, which is better for spectral analysis.

Parameters **length** (int) – the number of points of the Hanning window

Returns the window as a numpy.ndarray of shape (length,).

audiotsm.utils.windows.**product**(*window1*, *window2*)

Returns the product of two windows.

Parameters

- **window1** – a numpy.ndarray of shape (n,) or None.
- **window2** – a numpy.ndarray of shape (n,) or None.

Returns the product of the two windows. If one of the windows is equal to None, the other is returned, and if the two are equal to None, None is returned.

Python Module Index

Index