Tutorial

This section covers the fundamentals of developing with *librosa*, including a package overview, basic and advanced usage, and integration with the *scikit-learn* package. We will assume basic familiarity with Python and NumPy/SciPy.

Overview

The *librosa* package is structured as collection of submodules:

- librosa
  - librosa.beat

    Functions for estimating tempo and detecting beat events.

  - librosa.core

    Core functionality includes functions to load audio from disk, compute various spectrogram representations, and a variety of commonly used tools for music analysis. For convenience, all functionality in this submodule is directly accessible from the top-level `librosa.*` namespace.

  - librosa.decompose

    Functions for harmonic-percussive source separation (HPSS) and generic spectrogram decomposition using matrix decomposition methods implemented in *scikit-learn*.

  - librosa.display

    Visualization and display routines using `matplotlib`.

  - librosa.effects

    Time-domain audio processing, such as pitch shifting and time stretching. This submodule also provides time-domain wrappers for the *decompose* submodule.

  - librosa.feature

    Feature extraction and manipulation. This includes low-level feature extraction, such as chromagrams, pseudo-constant-Q (log-frequency) transforms, Mel spectrogram, MFCC, and tuning estimation. Also provided are feature manipulation methods, such as delta features, memory embedding, and event-synchronous feature alignment.

  - librosa.filters

Filter-bank generation (chroma, pseudo-CQT, CQT, etc.). These are primarily internal functions used by other parts of *librosa*.

- librosa.onset

  Onset detection and onset strength computation.

- librosa.output

  Text- and wav-file output. *(Deprecated)*

- librosa.segment

  Functions useful for structural segmentation, such as recurrence matrix construction, time-lag representation, and sequentially constrained clustering.

- librosa.sequence

  Functions for sequential modeling. Various forms of Viterbi decoding, and helper functions for constructing transition matrices.

- librosa.util

  Helper utilities (normalization, padding, centering, etc.)

Quickstart

Before diving into the details, we'll walk through a brief example program

```
# Beat tracking example
from __future__ import print_function
import librosa

# 1. Get the file path to the included audio example
filename = librosa.util.example_audio_file()

# 2. Load the audio as a waveform `y`
#    Store the sampling rate as `sr`
y, sr = librosa.load(filename)

# 3. Run the default beat tracker
tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)

print('Estimated tempo: {:.2f} beats per minute'.format(tempo))

# 4. Convert the frame indices of beat events into timestamps
beat_times = librosa.frames_to_time(beat_frames, sr=sr)
```

The first step of the program:

```
filename = librosa.util.example_audio_file()
```

gets the path to the audio example file included with *librosa*. After this step, `filename` will be a string variable containing the path to the example audio file. The example is encoded in OGG Vorbis format, so you will need the appropriate codec installed for audioread.

The second step:

```
y, sr = librosa.load(filename)
```

loads and decodes the audio as a time series `y`, represented as a one-dimensional NumPy floating point array. The variable `sr` contains the sampling rate of `y`, that is, the number of samples per second of audio. By default, all audio is mixed to mono and resampled to 22050 Hz at load time. This behavior can be overridden by supplying additional arguments to `librosa.load()`.

Next, we run the beat tracker:

```
tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)
```

The output of the beat tracker is an estimate of the tempo (in beats per minute), and an array of frame numbers corresponding to detected beat events.

Frames here correspond to short windows of the signal (`y`), each separated by `hop_length = 512` samples. Since v0.3, *librosa* uses centered frames, so that the *k*th frame is centered around sample `k * hop_length`.

The next operation converts the frame numbers `beat_frames` into timings:

```
beat_times = librosa.frames_to_time(beat_frames, sr=sr)
```

Now, `beat_times` will be an array of timestamps (in seconds) corresponding to detected beat events.

The contents of `beat_times` should look something like this:

```
7.43
8.29
9.218
10.124
...
```

Advanced usage

Here we'll cover a more advanced example, integrating harmonic-percussive separation, multiple spectral features, and beat-synchronous feature aggregation.

```
# Feature extraction example
import numpy as np
import librosa

# Load the example clip
y, sr = librosa.load(librosa.util.example_audio_file())

# Set the hop length; at 22050 Hz, 512 samples ~= 23ms
hop_length = 512

# Separate harmonics and percussives into two waveforms
```

```
y_harmonic, y_percussive = librosa.effects.hpss(y)

# Beat track on the percussive signal
tempo, beat_frames = librosa.beat.beat_track(y=y_percussive,
                                             sr=sr)

# Compute MFCC features from the raw signal
mfcc = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length, n_mfcc=13)

# And the first-order differences (delta features)
mfcc_delta = librosa.feature.delta(mfcc)

# Stack and synchronize between beat events
# This time, we'll use the mean value (default) instead of median
beat_mfcc_delta = librosa.util.sync(np.vstack([mfcc, mfcc_delta]),
                                    beat_frames)

# Compute chroma features from the harmonic signal
chromagram = librosa.feature.chroma_cqt(y=y_harmonic,
                                        sr=sr)

# Aggregate chroma features between beat events
# We'll use the median value of each feature between beat frames
beat_chroma = librosa.util.sync(chromagram,
                                beat_frames,
                                aggregate=np.median)

# Finally, stack all beat-synchronous features together
beat_features = np.vstack([beat_chroma, beat_mfcc_delta])
```

This example builds on tools we've already covered in the quickstart example, so here we'll focus just on the new parts.

The first difference is the use of the effects module for time-series harmonic-percussive separation:

```
y_harmonic, y_percussive = librosa.effects.hpss(y)
```

The result of this line is that the time series `y` has been separated into two time series, containing the harmonic (tonal) and percussive (transient) portions of the signal. Each of `y_harmonic` and `y_percussive` have the same shape and duration as `y`.

The motivation for this kind of operation is two-fold: first, percussive elements tend to be stronger indicators of rhythmic content, and can help provide more stable beat tracking results; second, percussive elements can pollute tonal feature representations (such as chroma) by contributing energy across all frequency bands, so we'd be better off without them.

Next, we introduce the feature module and extract the Mel-frequency cepstral coefficients from the raw signal `y`:

```
mfcc = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length, n_mfcc=13)
```

The output of this function is the matrix `mfcc`, which is an *numpy.ndarray* of size `(n_mfcc, T)` (where `T` denotes the track duration in frames). Note that we use the same `hop_length` here as in the beat tracker, so the detected `beat_frames` values correspond to columns of `mfcc`.

The first type of feature manipulation we introduce is `delta`, which computes (smoothed) first-order differences among columns of its input:

```
mfcc_delta = librosa.feature.delta(mfcc)
```

The resulting matrix `mfcc_delta` has the same shape as the input `mfcc`.

The second type of feature manipulation is `sync`, which aggregates columns of its input between sample indices (e.g., beat frames):

```
beat_mfcc_delta = librosa.util.sync(np.vstack([mfcc, mfcc_delta]),
                                    beat_frames)
```

Here, we've vertically stacked the `mfcc` and `mfcc_delta` matrices together. The result of this operation is a matrix `beat_mfcc_delta` with the same number of rows as its input, but the number of columns depends on `beat_frames`. Each column `beat_mfcc_delta[:, k]` will be the *average* of input columns between `beat_frames[k]` and `beat_frames[k+1]`. (`beat_frames` will be expanded to span the full range `[0, T]` so that all data is accounted for.)

Next, we compute a chromagram using just the harmonic component:

```
chromagram = librosa.feature.chroma_cqt(y=y_harmonic,
                                        sr=sr)
```

After this line, `chromagram` will be a *numpy.ndarray* of size `(12, T)`, and each row corresponds to a pitch class (e.g., *C*, *C#*, etc.). Each column of `chromagram` is normalized by its peak value, though this behavior can be overridden by setting the `norm` parameter.

Once we have the chromagram and list of beat frames, we again synchronize the chroma between beat events:

```
beat_chroma = librosa.util.sync(chromagram,
                                beat_frames,
                                aggregate=np.median)
```

This time, we've replaced the default aggregate operation (*average*, as used above for MFCCs) with the *median*. In general, any statistical summarization function can be supplied here, including *np.max()*, *np.min()*, *np.std()*, etc.

Finally, the all features are vertically stacked again:

```
beat_features = np.vstack([beat_chroma, beat_mfcc_delta])
```

resulting in a feature matrix `beat_features` of dimension `(12 + 13 + 13, # beat intervals)`.

More examples

More example scripts are provided in the advanced examples section.

Core IO and DSP

Audio processing

load(path[, sr, mono, offset, duration, …])	Load an audio file as a floating point time series.
stream(path, block_length, frame_length, …)	Stream audio in fixed-length buffers.
to_mono(y)	Force an audio signal down to mono by averaging samples across channels.
resample(y, orig_sr, target_sr[, res_type, …])	Resample a time series from orig_sr to target_sr
get_duration([y, sr, S, n_fft, hop_length, …])	Compute the duration (in seconds) of an audio time series, feature matrix, or filename.
get_samplerate(path)	Get the sampling rate for a given file.
autocorrelate(y[, max_size, axis])	Bounded auto-correlation
lpc(y, order)	Linear Prediction Coefficients via Burg's method
zero_crossings(y[, threshold, …])	Find the zero-crossings of a signal *y*: indices *i* such that *sign(y[i]) != sign(y[j])*.
clicks([times, frames, sr, hop_length, …])	Returns a signal with the signal `click` placed at each specified time
tone(frequency[, sr, length, duration, phi])	Returns a pure tone signal.
chirp(fmin, fmax[, sr, length, duration, …])	Returns a chirp signal that goes from frequency *fmin* to frequency *fmax*

Spectral representations

stft(y[, n_fft, hop_length, win_length, …])	Short-time Fourier transform (STFT).
istft(stft_matrix[, hop_length, win_length, …])	Inverse short-time Fourier transform (ISTFT).
reassigned_spectrogram(y[, sr, S, n_fft, …])	Time-frequency reassigned spectrogram.
cqt(y[, sr, hop_length, fmin, n_bins, …])	Compute the constant-Q transform of an audio signal.
icqt(C[, sr, hop_length, fmin, …])	Compute the inverse constant-Q transform.
hybrid_cqt(y[, sr, hop_length, fmin, …])	Compute the hybrid constant-Q transform of an audio signal.
pseudo_cqt(y[, sr, hop_length, fmin, …])	Compute the pseudo constant-Q transform of an audio signal.
iirt(y[, sr, win_length, hop_length, …])	Time-frequency representation using IIR filters [Rd4077732470d-1].
fmt(y[, t_min, n_fmt, kind, beta, …])	The fast Mellin transform (FMT) [R6343f8d4cac9-1] of a uniformly sampled signal y.
griffinlim(S[, n_iter, hop_length, …])	Approximate magnitude spectrogram inversion using the "fast" Griffin-Lim algorithm [R047f50301c96-1] [R047f50301c96-2].

griffinlim_cqt(C[, n_iter, sr, hop_length, …])	Approximate constant-Q magnitude spectrogram inversion using the “fast” Griffin-Lim algorithm [Re33fb425db1f-1] [Re33fb425db1f-2].
interp_harmonics(x, freqs, h_range[, kind, …])	Compute the energy at harmonics of time-frequency representation.
salience(S, freqs, h_range[, weights, …])	Harmonic salience function.
phase_vocoder(D, rate[, hop_length])	Phase vocoder.
magphase(D[, power])	Separate a complex-valued spectrogram D into its magnitude (S) and phase (P) components, so that $D = S * P$.
get_fftlib()	Get the FFT library currently used by librosa
set_fftlib([lib])	Set the FFT library used by librosa.

Magnitude scaling

amplitude_to_db(S[, ref, amin, top_db])	Convert an amplitude spectrogram to dB-scaled spectrogram.
db_to_amplitude(S_db[, ref])	Convert a dB-scaled spectrogram to an amplitude spectrogram.
power_to_db(S[, ref, amin, top_db])	Convert a power spectrogram (amplitude squared) to decibel (dB) units
db_to_power(S_db[, ref])	Convert a dB-scale spectrogram to a power spectrogram.
perceptual_weighting(S, frequencies, **kwargs)	Perceptual weighting of a power spectrogram:
A_weighting(frequencies[, min_db])	Compute the A-weighting of a set of frequencies.
pcen(S[, sr, hop_length, gain, bias, power, …])	Per-channel energy normalization (PCEN) [Rb388d53f6b92-1]

Time and frequency conversion

frames_to_samples(frames[, hop_length, n_fft])	Converts frame indices to audio sample indices.
frames_to_time(frames[, sr, hop_length, n_fft])	Converts frame counts to time (seconds).
samples_to_frames(samples[, hop_length, n_fft])	Converts sample indices into STFT frames.
samples_to_time(samples[, sr])	Convert sample indices to time (in seconds).
time_to_frames(times[, sr, hop_length, n_fft])	Converts time stamps into STFT frames.
time_to_samples(times[, sr])	Convert timestamps (in seconds) to sample indices.
blocks_to_frames(blocks, block_length)	Convert block indices to frame indices
blocks_to_samples(blocks, block_length, …)	Convert block indices to sample indices
blocks_to_time(blocks, block_length, …)	Convert block indices to time (in seconds)
hz_to_note(frequencies, **kwargs)	Convert one or more frequencies (in Hz) to the nearest note names.
hz_to_midi(frequencies)	Get MIDI note number(s) for given frequencies
midi_to_hz(notes)	Get the frequency (Hz) of MIDI note(s)
midi_to_note(midi[, octave, cents])	Convert one or more MIDI numbers to note strings.

note_to_hz(note, **kwargs)	Convert one or more note names to frequency (Hz)
note_to_midi(note[, round_midi])	Convert one or more spelled notes to MIDI number(s).
hz_to_mel(frequencies[, htk])	Convert Hz to Mels
hz_to_octs(frequencies[, tuning, …])	Convert frequencies (Hz) to (fractional) octave numbers.
mel_to_hz(mels[, htk])	Convert mel bin numbers to frequencies
octs_to_hz(octs[, tuning, bins_per_octave, A440])	Convert octaves numbers to frequencies.
fft_frequencies([sr, n_fft])	Alternative implementation of *np.fft.fftfreq*
cqt_frequencies(n_bins, fmin[, …])	Compute the center frequencies of Constant-Q bins.
mel_frequencies([n_mels, fmin, fmax, htk])	Compute an array of acoustic frequencies tuned to the mel scale.
tempo_frequencies(n_bins[, hop_length, sr])	Compute the frequencies (in beats per minute) corresponding to an onset auto-correlation or tempogram matrix.
fourier_tempo_frequencies([sr, win_length, …])	Compute the frequencies (in beats per minute) corresponding to a Fourier tempogram matrix.
samples_like(X[, hop_length, n_fft, axis])	Return an array of sample indices to match the time axis from a feature matrix.
times_like(X[, sr, hop_length, n_fft, axis])	Return an array of time values to match the time axis from a feature matrix.

Pitch and tuning

estimate_tuning([y, sr, S, n_fft, …])	Estimate the tuning of an audio time series or spectrogram input.
pitch_tuning(frequencies[, resolution, …])	Given a collection of pitches, estimate its tuning offset (in fractions of a bin) relative to A440=440.0Hz.
piptrack([y, sr, S, n_fft, hop_length, …])	Pitch tracking on thresholded parabolically-interpolated STFT.

Deprecated

ifgram(y[, sr, n_fft, hop_length, …])	Compute the instantaneous

librosa.core.load

`librosa.core.load`(*path*, *sr=22050*, *mono=True*, *offset=0.0*, *duration=None*, *dtype=<class 'numpy.float32'>*, *res_type='kaiser_best'*)[source]

Load an audio file as a floating point time series.

Audio will be automatically resampled to the given rate (default *sr=22050*).

To preserve the native sampling rate of the file, use *sr=None*.

Parameters: path : string, int, or file-like object

path to the input file.

Any codec supported by `soundfile` or `audioread` will work.

If the codec is supported by `soundfile`, then *path* can also be an open file descriptor (int), or any object implementing Python's file interface.

If the codec is not supported by `soundfile` (e.g., MP3), then only string file paths are supported.

sr : number > 0 [scalar]

target sampling rate

'None' uses the native sampling rate

mono : bool

convert signal to mono

offset : float

start reading after this time (in seconds)

duration : float

only load up to this much audio (in seconds)

dtype : numeric type

data type of *y*

res_type : str

resample type (see note)

Note

By default, this uses `resampy`'s high-quality mode ('kaiser_best').

For alternative resampling modes, see `resample`

Note

`audioread` may truncate the precision of the audio data to 16 bits.

See https://librosa.github.io/librosa/ioformats.html for alternate loading methods.

Returns: **y** : np.ndarray [shape=(n,) or (2, n)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

Examples

```
>>> # Load an ogg vorbis file
>>> filename = librosa.util.example_audio_file()
>>> y, sr = librosa.load(filename)
>>> y
array([ -4.756e-06,  -6.020e-06, ...,  -1.040e-06,   0.000e+00],
dtype=float32)
>>> sr
22050

>>> # Load a file and resample to 11 KHz
>>> filename = librosa.util.example_audio_file()
>>> y, sr = librosa.load(filename, sr=11025)
>>> y
array([ -2.077e-06,  -2.928e-06, ...,  -4.395e-06,   0.000e+00],
dtype=float32)
>>> sr
11025

>>> # Load 5 seconds of a file, starting 15 seconds in
>>> filename = librosa.util.example_audio_file()
>>> y, sr = librosa.load(filename, offset=15.0, duration=5.0)
>>> y
array([ 0.069,  0.1  , ..., -0.101,  0.   ], dtype=float32)
>>> sr
22050
```

librosa.core.stream

`librosa.core.stream`(*path*, *block_length*, *frame_length*, *hop_length*, *mono=True*, *offset=0.0*, *duration=None*, *fill_value=None*, *dtype=<class 'numpy.float32'>*)[source]

Stream audio in fixed-length buffers.

This is primarily useful for processing large files that won't fit entirely in memory at once.

Instead of loading the entire audio signal into memory (as in *load()*, this function produces *blocks* of audio spanning a fixed number of frames at a specified frame length and hop length.

While this function strives for similar behavior to `load`, there are a few caveats that users should be aware of:

1. This function does not return audio buffers directly. It returns a generator, which you can iterate over to produce blocks of audio. A *block*, in this context, refers to a buffer of audio which spans a given number of (potentially overlapping) frames.
2. Automatic sample-rate conversion is not supported. Audio will be streamed in its native sample rate, so no default values are provided for *frame_length* and *hop_length*. It is recommended that you first get the sampling rate for the file in question, using *get_samplerate()*, and set these parameters accordingly.
3. Many analyses require access to the entire signal to behave correctly, such as `resample`, `cqt`, or *beat_track*, so these methods will not be appropriate for streamed data.
4. The *block_length* parameter specifies how many frames of audio will be produced per block. Larger values will consume more memory, but will be more efficient to process down-stream. The best value will ultimately depend on your application and other system constraints.
5. By default, most librosa analyses (e.g., short-time Fourier transform) assume centered frames, which requires padding the signal at the beginning and end. This will not work correctly when the signal is carved into blocks, because it would introduce padding in the middle of the signal. To disable this feature, use *center=False* in all frame-based analyses.

See the examples below for proper usage of this function.

Parameters: path : string, int, or file-like object

path to the input file to stream.

Any codec supported by `soundfile` is permitted here.

block_length : int > 0

The number of frames to include in each block.

Note that at the end of the file, there may not be enough data to fill an

entire block, resulting in a shorter block by default. To pad the signal out so that blocks are always full length, set *fill_value* (see below).

frame_length : int > 0

The number of samples per frame.

hop_length : int > 0

The number of samples to advance between frames.

Note that by when *hop_length* < *frame_length*, neighboring frames will overlap. Similarly, the last frame of one *block* will overlap with the first frame of the next *block*.

mono : bool

Convert the signal to mono during streaming

offset : float

Start reading after this time (in seconds)

duration : float

Only load up to this much audio (in seconds)

fill_value : float [optional]

If padding the signal to produce constant-length blocks, this value will be used at the end of the signal.

In most cases, *fill_value=0* (silence) is expected, but you may specify any value here.

dtype : numeric type

data type of audio buffers to be produced

Yields: **y** : np.ndarray

An audio buffer of (at most) *block_length * (hop_length-1) + frame_length* samples.

See also

`load`
`get_samplerate`
`soundfile.blocks`

Examples

Apply a short-term Fourier transform to blocks of 256 frames at a time. Note that streaming operation requires left-aligned frames, so we must set *center=False* to avoid padding artifacts.

```
>>> filename = librosa.util.example_audio_file()
>>> sr = librosa.get_samplerate(filename)
>>> stream = librosa.stream(filename,
...                         block_length=256,
...                         frame_length=4096,
...                         hop_length=1024)
>>> for y_block in stream:
...     D_block = librosa.stft(y_block, center=False)
```

Or compute a mel spectrogram over a stream, using a shorter frame and non-overlapping windows

```
>>> filename = librosa.util.example_audio_file()
>>> sr = librosa.get_samplerate(filename)
>>> stream = librosa.stream(filename,
...                          block_length=256,
...                          frame_length=2048,
...                          hop_length=2048)
>>> for y_block in stream:
...     m_block = librosa.feature.melspectrogram(y_block, sr=sr,
...                                              n_fft=2048,
...                                              hop_length=2048,
...                                              center=False)
```

librosa.core.to_mono

librosa.core.to_mono(*y*)[source]

Force an audio signal down to mono by averaging samples across channels.

Parameters: **y** : np.ndarray [shape=(2,n) or shape=(n,)]

audio time series, either stereo or mono

Returns: **y_mono** : np.ndarray [shape=(n,)]

y as a monophonic time-series

Notes

This function caches at level 20.

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), mono=False)
>>> y.shape
(2, 1355168)
>>> y_mono = librosa.to_mono(y)
>>> y_mono.shape
(1355168,)
```

librosa.core.resample

`librosa.core.resample`(*y*, *orig_sr*, *target_sr*, *res_type='kaiser_best'*, *fix=True*, *scale=False*, ***kwargs*)[source]

Resample a time series from orig_sr to target_sr

Parameters: **y** : np.ndarray [shape=(n,) or shape=(2, n)]

audio time series. Can be mono or stereo.

orig_sr : number > 0 [scalar]

original sampling rate of *y*

target_sr : number > 0 [scalar]

target sampling rate

res_type : str

resample type (see note)

Note

By default, this uses `resampy`'s high-quality mode ('kaiser_best').

To use a faster method, set *res_type='kaiser_fast'*.

To use `scipy.signal.resample`, set *res_type='fft'* or *res_type='scipy'*.

To use `scipy.signal.resample_poly`, set *res_type='polyphase'*.

Note

When using *res_type='polyphase'*, only integer sampling rates are supported.

fix : bool

adjust the length of the resampled signal to be of size exactly *ceil(target_sr * len(y) / orig_sr)*

scale : bool

Scale the resampled signal so that *y* and *y_hat* have approximately equal total energy.

kwargs : additional keyword arguments

If *fix==True*, additional keyword arguments to pass to `librosa.util.fix_length`.

Returns: **y_hat** : np.ndarray [shape=(n * target_sr / orig_sr,)]

y resampled from *orig_sr* to *target_sr*

Raises: ParameterError

If *res_type='polyphase'* and *orig_sr* or *target_sr* are not both integer-valued.

See also

`librosa.util.fix_length`
`scipy.signal.resample`
`resampy.resample`

Notes

This function caches at level 20.

Examples

Downsample from 22 KHz to 8 KHz

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), sr=22050)
>>> y_8k = librosa.resample(y, sr, 8000)
>>> y.shape, y_8k.shape
((1355168,), (491671,)
```

librosa.core.get_duration

`librosa.core.get_duration`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, center=True, filename=None*)[source]

Compute the duration (in seconds) of an audio time series, feature matrix, or filename.

Parameters: **y** : np.ndarray [shape=(n,), (2, n)] or None

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

STFT matrix, or any STFT-derived matrix (e.g., chromagram or mel spectrogram). Durations calculated from spectrogram inputs are only accurate up to the frame resolution. If high precision is required, it is better to use the audio time series directly.

n_fft : int > 0 [scalar]

FFT window size for *S*

hop_length : int > 0 [scalar]

number of audio samples between columns of *S*

center : boolean

- If *True*, *S[:, t]* is centered at *y[t * hop_length]*
- If *False*, then *S[:, t]* begins at *y[t * hop_length]*

filename : str

If provided, all other parameters are ignored, and the duration is calculated directly from the audio file. Note that this avoids loading the contents into memory, and is therefore useful for querying the duration of long files.

As in *load()*, this can also be an integer or open file-handle that can be processed by `soundfile`.

Returns: **d** : float >= 0

Duration (in seconds) of the input time series or spectrogram.

Raises: ParameterError

if none of *y*, *S*, or *filename* are provided.

Notes

`get_duration` can be applied to a file (*filename*), a spectrogram (*S*), or audio buffer (*y, sr*). Only one of these three options should be provided. If you do provide multiple options (e.g., *filename* and *S*), then *filename* takes precedence over *S*, and *S* takes precedence over *(y, sr)*.

Examples

```
>>> # Load the example audio file
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.get_duration(y=y, sr=sr)
61.45886621315193

>>> # Or directly from an audio file
>>> librosa.get_duration(filename=librosa.util.example_audio_file())
61.4

>>> # Or compute duration from an STFT matrix
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = librosa.stft(y)
>>> librosa.get_duration(S=S, sr=sr)
61.44

>>> # Or a non-centered STFT matrix
>>> S_left = librosa.stft(y, center=False)
>>> librosa.get_duration(S=S_left, sr=sr)
61.3471201814059
```

librosa.core.get_samplerate

`librosa.core.get_samplerate`(*path*)[source]

Get the sampling rate for a given file.

Parameters: path : string, int, or file-like

The path to the file to be loaded As in *load()*, this can also be an integer or open file-handle that can be processed by `soundfile`.

Returns: **sr** : number > 0

The sampling rate of the given audio file

Examples

Get the sampling rate for the included audio file

```
>>> path = librosa.util.example_audio_file()
>>> librosa.get_samplerate(path)
44100
```

librosa.core.autocorrelate

`librosa.core.autocorrelate`(*y*, *max_size=None*, *axis=-1*)[source]

Bounded auto-correlation

Parameters: y : np.ndarray

array to autocorrelate

max_size : int > 0 or None

maximum correlation lag. If unspecified, defaults to *y.shape[axis]* (unbounded)

axis : int

The axis along which to autocorrelate. By default, the last axis (-1) is taken.

Returns: **z** : np.ndarray

truncated autocorrelation *y*y* along the specified axis. If *max_size* is specified, then *z.shape[axis]* is bounded to *max_size*.

Notes

This function caches at level 20.

Examples

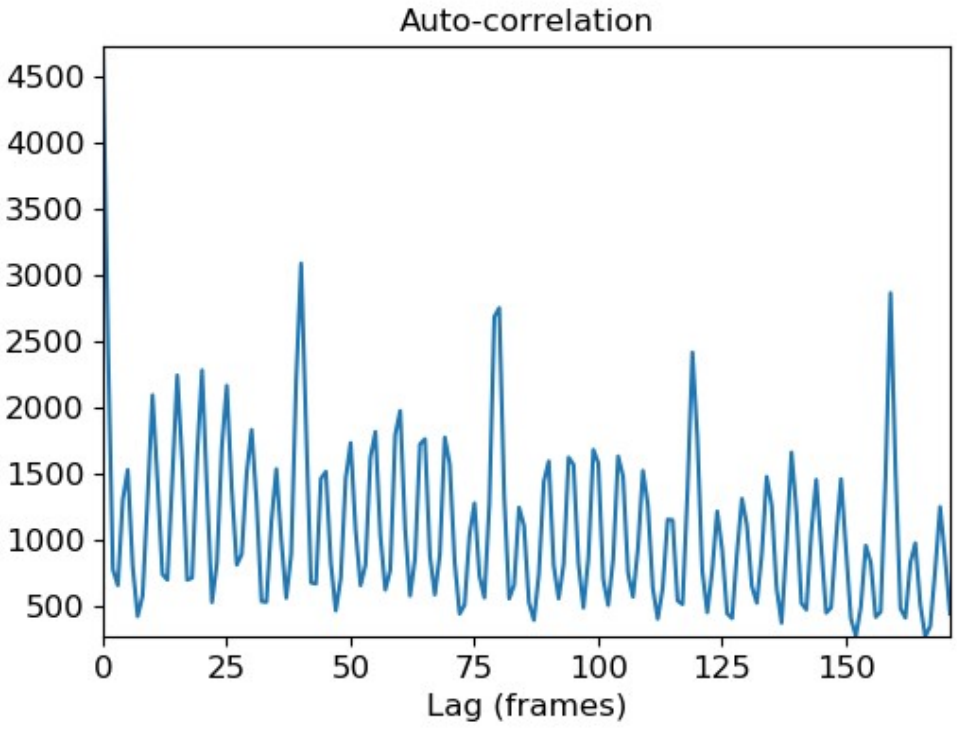
Compute full autocorrelation of y

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=20,
duration=10)
>>> librosa.autocorrelate(y)
array([  3.226e+03,   3.217e+03, ...,   8.277e-04,   3.575e-04],
dtype=float32)
```

Compute onset strength auto-correlation up to 4 seconds

```
>>> import matplotlib.pyplot as plt
>>> odf = librosa.onset.onset_strength(y=y, sr=sr, hop_length=512)
>>> ac = librosa.autocorrelate(odf, max_size=4* sr / 512)
>>> plt.plot(ac)
>>> plt.title('Auto-correlation')
>>> plt.xlabel('Lag (frames)')
>>> plt.show()
```

(Source code)



librosa.core.lpc

`librosa.core.lpc`(*y*, *order*)[source]

Linear Prediction Coefficients via Burg's method

This function applies Burg's method to estimate coefficients of a linear filter on *y* of order *order*. Burg's method is an extension to the Yule-Walker approach, which are both sometimes referred to as LPC parameter estimation by autocorrelation.

It follows the description and implementation approach described in the introduction in [1]. N.B. This paper describes a different method, which is not implemented here, but has been chosen for its clear explanation of Burg's technique in its introduction.

[1] Larry Marple A New Autoregressive Spectrum Analysis Algorithm IEEE Transactions on Accoustics, Speech, and Signal Processing vol 28, no. 4, 1980

Parameters: **y** : np.ndarray

Time series to fit

order : int > 0

Order of the linear filter

Returns: **a** : np.ndarray of length order + 1

LP prediction error coefficients, i.e. filter denominator polynomial

Raises: ParameterError

- If y is not valid audio as per *util.valid_audio*
- If order < 1 or not integer

FloatingPointError

- If y is ill-conditioned

See also

`scipy.signal.lfilter`

Examples

Compute LP coefficients of y at order 16 on entire series

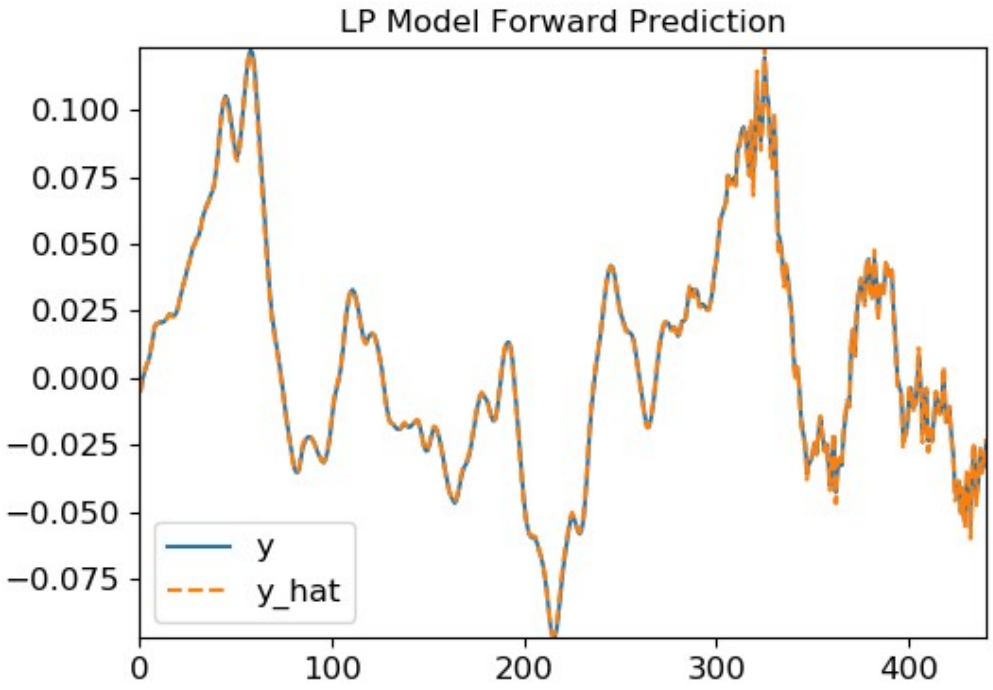
```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=30,
...                      duration=10)
>>> librosa.lpc(y, 16)
```

Compute LP coefficients, and plot LP estimate of original series

```
>>> import matplotlib.pyplot as plt
>>> import scipy
```

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=30,
...                      duration=0.020)
>>> a = librosa.lpc(y, 2)
>>> y_hat = scipy.signal.lfilter([0] + -1*a[1:], [1], y)
>>> plt.figure()
>>> plt.plot(y)
>>> plt.plot(y_hat, linestyle='--')
>>> plt.legend(['y', 'y_hat'])
>>> plt.title('LP Model Forward Prediction')
>>> plt.show()
```

(Source code)



librosa.core.zero_crossings

`librosa.core.zero_crossings`(*y*, *threshold=1e-10*, *ref_magnitude=None*, *pad=True*, *zero_pos=True*, *axis=-1*)[source]

Find the zero-crossings of a signal *y*: indices *i* such that *sign(y[i]) != sign(y[j])*.

If *y* is multi-dimensional, then zero-crossings are computed along the specified *axis*.

Parameters: **y** : np.ndarray

The input array

threshold : float > 0 or None

If specified, values where *-threshold <= y <= threshold* are clipped to 0.

ref_magnitude : float > 0 or callable

If numeric, the threshold is scaled relative to *ref_magnitude*.

If callable, the threshold is scaled relative to *ref_magnitude(np.abs(y))*.

pad : boolean

If *True*, then *y[0]* is considered a valid zero-crossing.

zero_pos : boolean

If *True* then the value 0 is interpreted as having positive sign.

If *False*, then 0, -1, and +1 all have distinct signs.

axis : int

Axis along which to compute zero-crossings.

Returns: **zero_crossings** : np.ndarray [shape=y.shape, dtype=boolean]

Indicator array of zero-crossings in *y* along the selected axis.

Notes

This function caches at level 20.

Examples

```
>>> # Generate a time-series
>>> y = np.sin(np.linspace(0, 4 * 2 * np.pi, 20))
>>> y
```

```
array([  0.000e+00,   9.694e-01,   4.759e-01,  -7.357e-01,
        -8.372e-01,   3.247e-01,   9.966e-01,   1.646e-01,
        -9.158e-01,  -6.142e-01,   6.142e-01,   9.158e-01,
        -1.646e-01,  -9.966e-01,  -3.247e-01,   8.372e-01,
         7.357e-01,  -4.759e-01,  -9.694e-01,  -9.797e-16])
>>> # Compute zero-crossings
>>> z = librosa.zero_crossings(y)
>>> z
array([ True, False, False,  True, False,  True, False, False,
        True, False,  True, False,  True, False, False,  True,
       False,  True, False,  True], dtype=bool)
>>> # Stack y against the zero-crossing indicator
>>> librosa.util.stack([y, z], axis=-1)
array([[  0.000e+00,   1.000e+00],
       [  9.694e-01,   0.000e+00],
       [  4.759e-01,   0.000e+00],
       [ -7.357e-01,   1.000e+00],
       [ -8.372e-01,   0.000e+00],
       [  3.247e-01,   1.000e+00],
       [  9.966e-01,   0.000e+00],
       [  1.646e-01,   0.000e+00],
       [ -9.158e-01,   1.000e+00],
       [ -6.142e-01,   0.000e+00],
       [  6.142e-01,   1.000e+00],
       [  9.158e-01,   0.000e+00],
       [ -1.646e-01,   1.000e+00],
       [ -9.966e-01,   0.000e+00],
       [ -3.247e-01,   0.000e+00],
       [  8.372e-01,   1.000e+00],
       [  7.357e-01,   0.000e+00],
       [ -4.759e-01,   1.000e+00],
       [ -9.694e-01,   0.000e+00],
       [ -9.797e-16,   1.000e+00]])
>>> # Find the indices of zero-crossings
>>> np.nonzero(z)
(array([ 0,  3,  5,  8, 10, 12, 15, 17, 19]),)
```

librosa.core.clicks

`librosa.core.clicks`(*times=None*, *frames=None*, *sr=22050*, *hop_length=512*, *click_freq=1000.0*, *click_duration=0.1*, *click=None*, *length=None*)[source]

Returns a signal with the signal `click` placed at each specified time

Parameters: **times** : np.ndarray or None

times to place clicks, in seconds

frames : np.ndarray or None

frame indices to place clicks

sr : number > 0

desired sampling rate of the output signal

hop_length : int > 0

if positions are specified by *frames*, the number of samples between frames.

click_freq : float > 0

frequency (in Hz) of the default click signal. Default is 1KHz.

click_duration : float > 0

duration (in seconds) of the default click signal. Default is 100ms.

click : np.ndarray or None

optional click signal sample to use instead of the default blip.

length : int > 0

desired number of samples in the output signal

Returns: **click_signal** : np.ndarray

Synthesized click signal

Raises: ParameterError

- If neither *times* nor *frames* are provided.
- If any of *click_freq*, *click_duration*, or *length* are out of range.

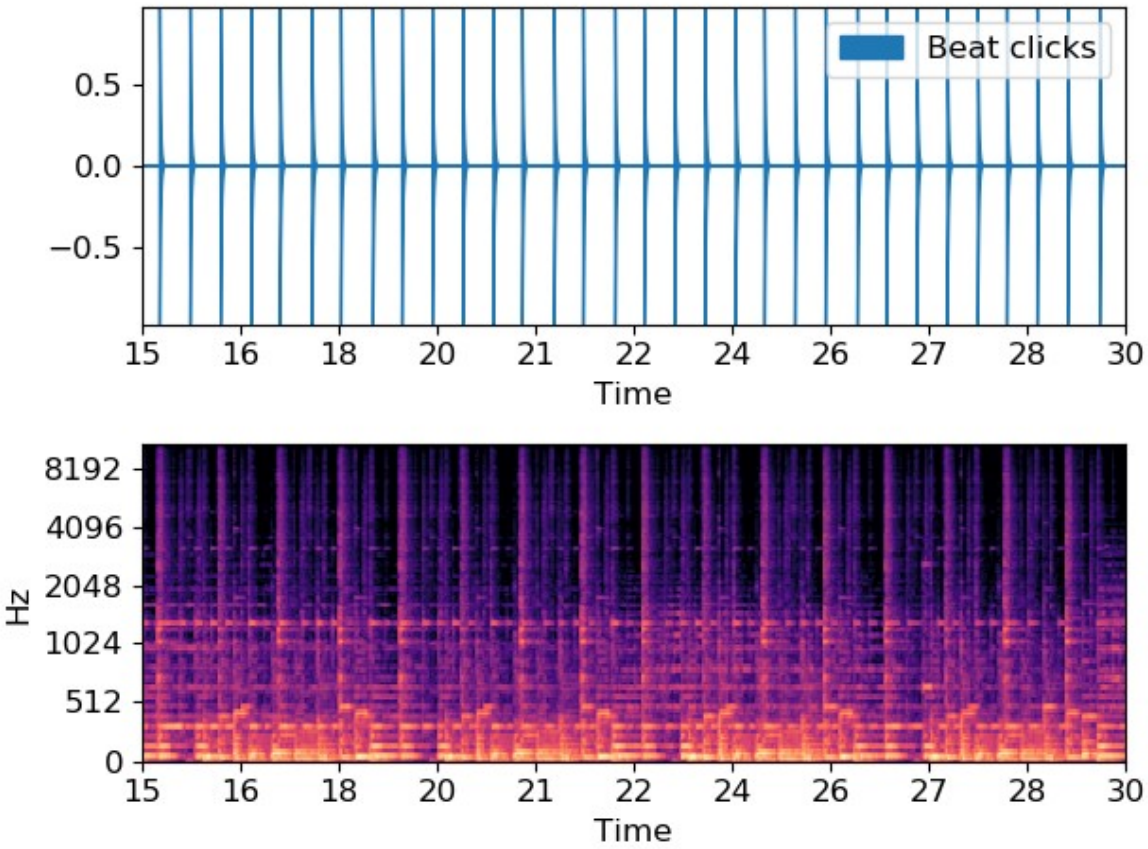
Examples

```
>>> # Sonify detected beat events
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
>>> y_beats = librosa.clicks(frames=beats, sr=sr)

>>> # Or generate a signal of the same length as y
>>> y_beats = librosa.clicks(frames=beats, sr=sr, length=len(y))

>>> # Or use timing instead of frame indices
>>> times = librosa.frames_to_time(beats, sr=sr)
>>> y_beat_times = librosa.clicks(times=times, sr=sr)

>>> # Or with a click frequency of 880Hz and a 500ms sample
>>> y_beat_times880 = librosa.clicks(times=times, sr=sr,
...                                  click_freq=880, click_duration=0.5)
```

Display click waveform next to the spectrogram

```
>>> import matplotlib.pyplot as plt
```

```
>>> plt.figure()
>>> S = librosa.feature.melspectrogram(y=y, sr=sr)
>>> ax = plt.subplot(2,1,2)
>>> librosa.display.specshow(librosa.power_to_db(S, ref=np.max),
...                          x_axis='time', y_axis='mel')
>>> plt.subplot(2,1,1, sharex=ax)
>>> librosa.display.waveplot(y_beat_times, sr=sr, label='Beat clicks')
>>> plt.legend()
>>> plt.xlim(15, 30)
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.tone

`librosa.core.tone`(*frequency*, *sr=22050*, *length=None*, *duration=None*, *phi=None*)[source]

Returns a pure tone signal. The signal generated is a cosine wave.

Parameters: **frequency** : float > 0

frequency

sr : number > 0

desired sampling rate of the output signal

length : int > 0

desired number of samples in the output signal. When both *duration* and *length* are defined, *length* would take priority.

duration : float > 0

desired duration in seconds. When both *duration* and *length* are defined, *length* would take priority.

phi : float or None

phase offset, in radians. If unspecified, defaults to *-np.pi * 0.5*.

Returns: **tone_signal** : np.ndarray [shape=(length,), dtype=float64]

Synthesized pure sine tone signal

Raises: ParameterError

- If *frequency* is not provided.
- If neither *length* nor *duration* are provided.

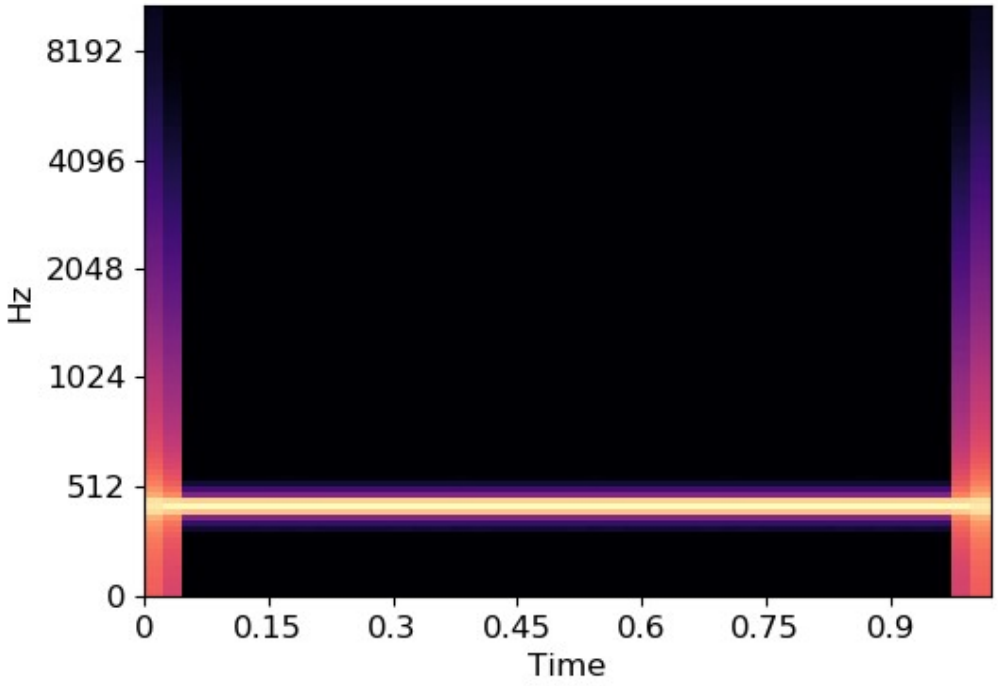
Examples

```
>>> # Generate a pure sine tone A4
>>> tone = librosa.tone(440, duration=1)

>>> # Or generate the same signal using `length`
>>> tone = librosa.tone(440, sr=22050, length=22050)
```

Display spectrogram

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> S = librosa.feature.melspectrogram(y=tone)
>>> librosa.display.specshow(librosa.power_to_db(S, ref=np.max),
...                          x_axis='time', y_axis='mel')
>>> plt.show()
```

(Source code)



librosa.core.chirp

`librosa.core.chirp`(*fmin, fmax, sr=22050, length=None, duration=None, linear=False, phi=None*)[source]

Returns a chirp signal that goes from frequency *fmin* to frequency *fmax*

Parameters: **fmin** : float > 0

initial frequency

fmax : float > 0

final frequency

sr : number > 0

desired sampling rate of the output signal

length : int > 0

desired number of samples in the output signal. When both *duration* and *length* are defined, *length* would take priority.

duration : float > 0

desired duration in seconds. When both *duration* and *length* are defined, *length* would take priority.

linear : boolean

- If *True*, use a linear sweep, i.e., frequency changes linearly with time
- If *False*, use a exponential sweep.

Default is *False*.

phi : float or None

phase offset, in radians. If unspecified, defaults to *-np.pi * 0.5*.

Returns: **chirp_signal** : np.ndarray [shape=(length,), dtype=float64]

Synthesized chirp signal

Raises: ParameterError

- If either *fmin* or *fmax* are not provided.
- If neither *length* nor *duration* are provided.

See also

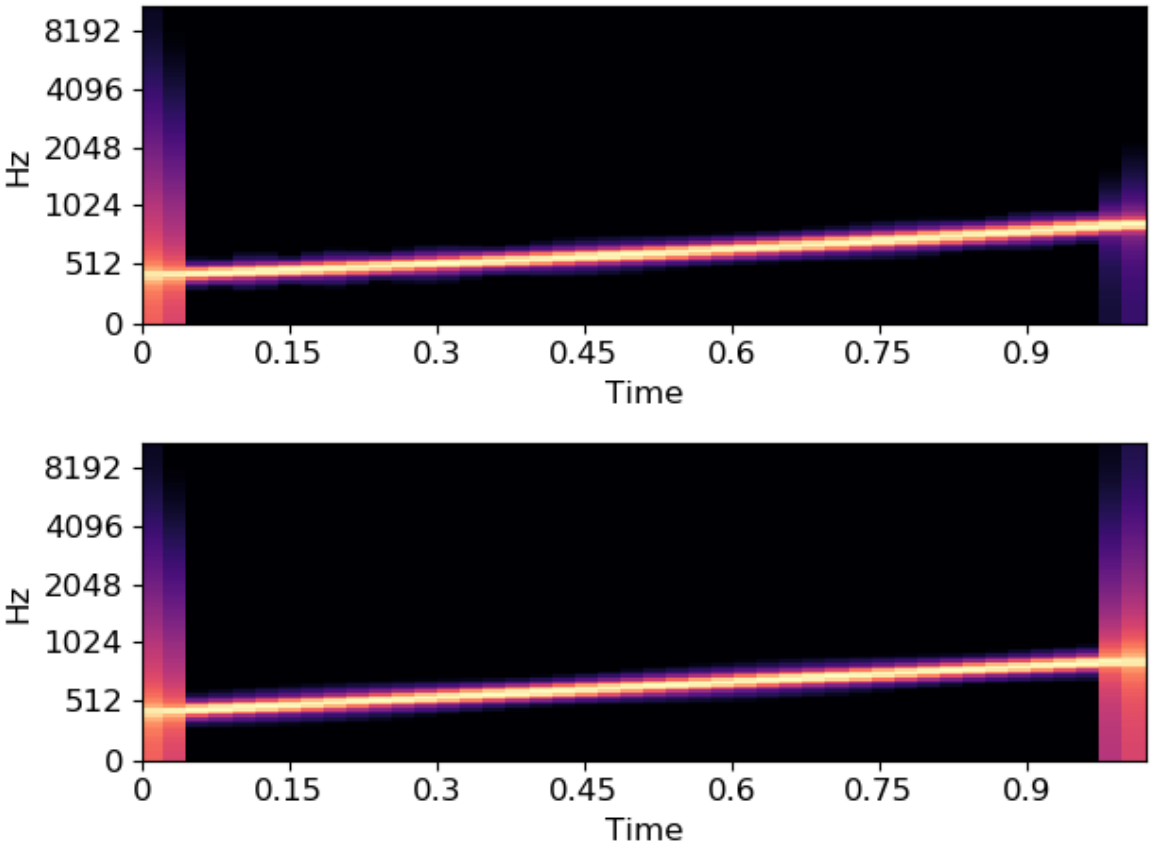
scipy.signal.chirp

Examples

```
>>> # Generate a exponential chirp from A4 to A5
>>> exponential_chirp = librosa.chirp(440, 880, duration=1)

>>> # Or generate the same signal using `length`
>>> exponential_chirp = librosa.chirp(440, 880, sr=22050, length=22050)

>>> # Or generate a linear chirp instead
>>> linear_chirp = librosa.chirp(440, 880, duration=1, linear=True)
```

Display spectrogram for both exponential and linear chirps

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> S_exponential = librosa.feature.melspectrogram(y=exponential_chirp)
>>> ax = plt.subplot(2,1,1)
>>> librosa.display.specshow(librosa.power_to_db(S_exponential,
ref=np.max),
...                          x_axis='time', y_axis='mel')
>>> plt.subplot(2,1,2, sharex=ax)
>>> S_linear = librosa.feature.melspectrogram(y=linear_chirp)
>>> librosa.display.specshow(librosa.power_to_db(S_linear, ref=np.max),
...                          x_axis='time', y_axis='mel')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.stft

`librosa.core.stft`(*y, n_fft=2048, hop_length=None, win_length=None, window='hann', center=True, dtype=<class 'numpy.complex64'>, pad_mode='reflect'*)[source]

Short-time Fourier transform (STFT). [1] (chapter 2)

The STFT represents a signal in the time-frequency domain by computing discrete Fourier transforms (DFT) over short overlapping windows.

This function returns a complex-valued matrix D such that

- *np.abs(D[f, t])* is the magnitude of frequency bin *f* at frame *t*, and
- *np.angle(D[f, t])* is the phase of frequency bin *f* at frame *t*.

The integers *t* and *f* can be converted to physical units by means of the utility functions *frames_to_sample* and `fft_frequencies`.

[1] 13.Müller. “Fundamentals of Music Processing.” Springer, 2015

Parameters:

y : np.ndarray [shape=(n,)], real-valued

input signal

n_fft : int > 0 [scalar]

length of the windowed signal after padding with zeros. The number of rows in the STFT matrix *D* is (1 + n_fft/2). The default value, n_fft=2048 samples, corresponds to a physical duration of 93 milliseconds at a sample rate of 22050 Hz, i.e. the default sample rate in librosa. This value is well adapted for music signals. However, in speech processing, the recommended value is 512, corresponding to 23 milliseconds at a sample rate of 22050 Hz. In any case, we recommend setting *n_fft* to a power of two for optimizing the speed of the fast Fourier transform (FFT) algorithm.

hop_length : int > 0 [scalar]

number of audio samples between adjacent STFT columns.

Smaller values increase the number of columns in *D* without affecting the frequency resolution of the STFT.

If unspecified, defaults to *win_length / 4* (see below).

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()* of length *win_length* and then padded with zeros to match *n_fft*.

Smaller values improve the temporal resolution of the STFT (i.e. the ability to discriminate impulses that are closely spaced in time) at the expense of frequency resolution (i.e. the ability to discriminate pure tones that are closely spaced in frequency). This effect is known as the time-frequency localization tradeoff and needs to be adjusted according to the properties of the input signal *y*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

Either:

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

Defaults to a raised cosine window ("hann"), which is adequate for most applications in audio signal processing.

center : boolean

If *True*, the signal *y* is padded so that frame *D[:, t]* is centered at *y[t * hop_length]*.

If *False*, then *D[:, t]* begins at *y[t * hop_length]*.

Defaults to *True*, which simplifies the alignment of *D* onto a time grid by means of `librosa.core.frames_to_samples`. Note, however, that *center* must be set to *False* when analyzing signals with `librosa.stream`.

dtype : numeric type

Complex numeric type for *D*. Default is single-precision floating-point complex (*np.complex64*).

pad_mode : string or function

If *center=True*, this argument is passed to *np.pad* for padding the edges of the signal *y*. By default (*pad_mode="reflect"*), *y* is padded on both sides with its own reflection, mirrored around its first and last sample respectively. If *center=False*, this argument is ignored.

Returns: **D** : np.ndarray [shape=(1 + n_fft/2, n_frames), dtype=dtype]

Complex-valued matrix of short-term Fourier transform coefficients.

See also

`istft`
: Inverse STFT

`reassigned_spectrogram`
: Time-frequency reassigned spectrogram

Notes

This function caches at level 20.

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> D = np.abs(librosa.stft(y))
>>> D
array([[2.58028018e-03, 4.32422794e-02, 6.61255598e-01, ...,
        6.82710262e-04, 2.51654536e-04, 7.23036574e-05],
       [2.49403086e-03, 5.15930466e-02, 6.00107312e-01, ...,
        3.48026224e-04, 2.35853557e-04, 7.54836728e-05],
       [7.82410789e-04, 1.05394892e-01, 4.37517226e-01, ...,
        6.29352580e-04, 3.38571583e-04, 8.38094638e-05],
       ...,
       [9.48568513e-08, 4.74725084e-07, 1.50052492e-05, ...,
        1.85637656e-08, 2.89708542e-08, 5.74304337e-09],
       [1.25165826e-07, 8.58259284e-07, 1.11157215e-05, ...,
        3.49099771e-08, 3.11740926e-08, 5.29926236e-09],
       [1.70630571e-07, 8.92518756e-07, 1.23656537e-05, ...,
        5.33256745e-08, 3.33264900e-08, 5.13272980e-09]], dtype=float32)
```

Use left-aligned frames, instead of centered frames

```
>>> D_left = np.abs(librosa.stft(y, center=False))
```

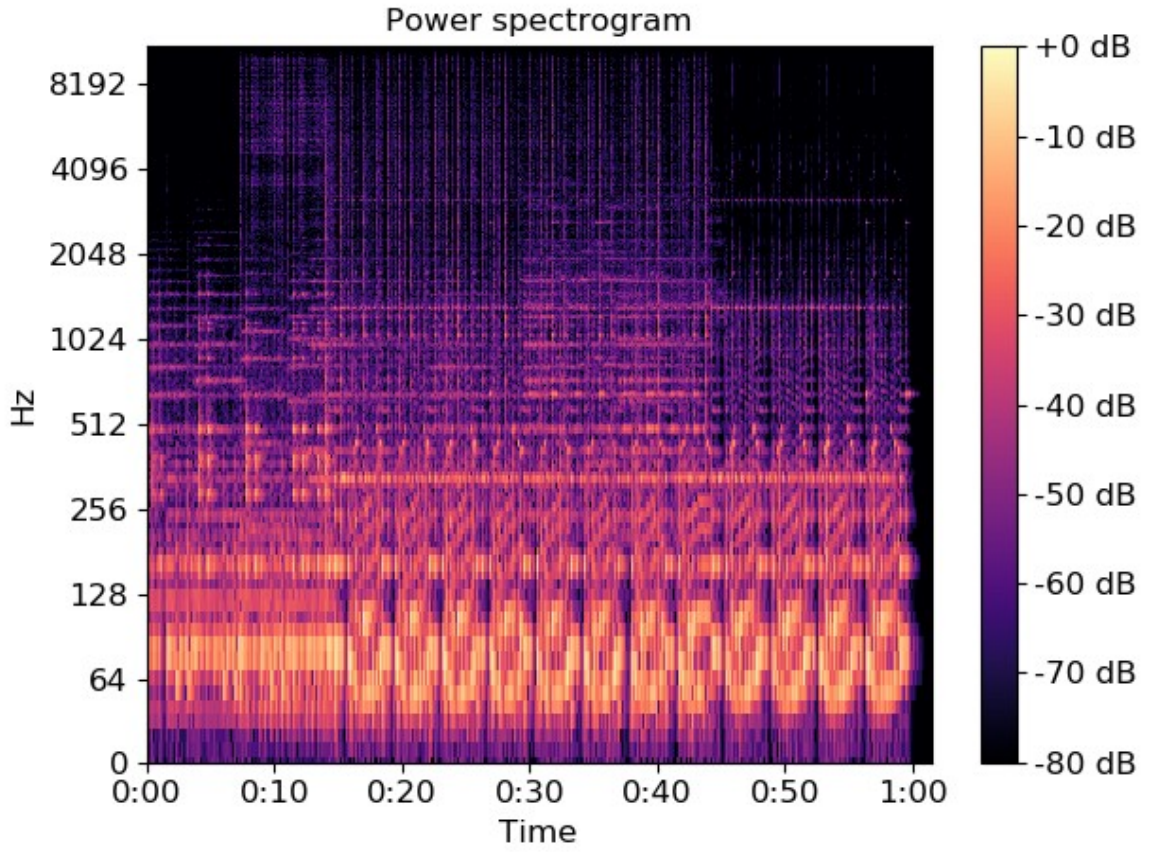
Use a shorter hop length

```
>>> D_short = np.abs(librosa.stft(y, hop_length=64))
```

Display a spectrogram

```
>>> import matplotlib.pyplot as plt
>>> librosa.display.specshow(librosa.amplitude_to_db(D,
...                                                  ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('Power spectrogram')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.tight_layout()
>>> plt.show()
```

(Source code)



librosa.core.istft

`librosa.core.istft`(*stft_matrix*, *hop_length=None*, *win_length=None*, *window='hann'*, *center=True*, *dtype=<class 'numpy.float32'>*, *length=None*)[source]

Inverse short-time Fourier transform (ISTFT).

Converts a complex-valued spectrogram *stft_matrix* to time-series *y* by minimizing the mean squared error between *stft_matrix* and STFT of *y* as described in [1] up to Section 2 (reconstruction from MSTFT).

In general, window function, hop length and other parameters should be same as in stft, which mostly leads to perfect reconstruction of a signal from unmodified *stft_matrix*.

[1] D. W. Griffin and J. S. Lim, “Signal estimation from modified short-time Fourier transform,” IEEE Trans. ASSP, vol.32, no.2, pp.236–243, Apr. 1984.

Parameters: stft_matrix : np.ndarray [shape=(1 + n_fft/2, t)]

STFT matrix from `stft`

hop_length : int > 0 [scalar]

Number of frames between STFT columns. If unspecified, defaults to *win_length / 4*.

win_length : int <= n_fft = 2 * (stft_matrix.shape[0] - 1)

When reconstructing the time series, each frame is windowed and each sample is normalized by the sum of squared window according to the *window* function (see below).

If unspecified, defaults to *n_fft*.

window : string, tuple, number, function, np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a user-specified window vector of length *n_fft*

center : boolean

- If *True*, *D* is assumed to have centered frames.
- If *False*, *D* is assumed to have left-aligned frames.

dtype : numeric type

Real numeric type for *y*. Default is 32-bit float.

length : int > 0, optional

If provided, the output *y* is zero-padded or clipped to exactly *length* samples.

Returns: **y** : np.ndarray [shape=(n,)]

time domain signal reconstructed from *stft_matrix*

See also

`stft`
Short-time Fourier Transform

Notes

This function caches at level 30.

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> D = librosa.stft(y)
>>> y_hat = librosa.istft(D)
>>> y_hat
array([ -4.812e-06,  -4.267e-06, ...,   6.271e-06,   2.827e-07],
dtype=float32)
```

Exactly preserving length of the input signal requires explicit padding. Otherwise, a partial frame at the end of *y* will not be represented.

```
>>> n = len(y)
>>> n_fft = 2048
>>> y_pad = librosa.util.fix_length(y, n + n_fft // 2)
>>> D = librosa.stft(y_pad, n_fft=n_fft)
```

```
>>> y_out = librosa.istft(D, length=n)
>>> np.max(np.abs(y - y_out))
1.4901161e-07
```

librosa.core.reassigned_spectrogram

`librosa.core.reassigned_spectrogram`(*y, sr=22050, S=None, n_fft=2048, hop_length=None, win_length=None, window='hann', center=True, reassign_frequencies=True, reassign_times=True, ref_power=1e-06, fill_nan=False, clip=True, dtype=<class 'numpy.complex64'>, pad_mode='reflect'*)[source]

Time-frequency reassigned spectrogram.

The reassignment vectors are calculated using equations 5.20 and 5.23 in [1]:

$$\omega\hat{} = \omega - \mathrm{I}(SdhSh)t\hat{} = t + \mathrm{R}(SthSh)$$

where *S_h* is the complex STFT calculated using the original window, *S_dh* is the complex STFT calculated using the derivative of the original window, and *S_th* is the complex STFT calculated using the original window multiplied by the time offset from the window center. See [2] for additional algorithms, and [3] and [4] for history and discussion of the method.

It is recommended to use *center=False* with this function rather than the librosa default *True*. Unlike `stft`, reassigned times are not aligned to the left or center of each frame, so padding the signal does not affect the meaning of the reassigned times. However, reassignment assumes that the energy in each FFT bin is associated with exactly one signal component and impulse event. The default *center=True* with reflection padding can thus invalidate the reassigned estimates in the half-reflected frames at the beginning and end of the signal.

If *reassign_times* is *False*, the frame times that are returned will be aligned to the left or center of the frame, depending on the value of *center*. In this case, if *center* is *True*, then *pad_mode="wrap"* is recommended for valid estimation of the instantaneous frequencies in the boundary frames.

[1] Flandrin, P., Auger, F., & Chassande-Mottin, E. (2002). Time-Frequency reassignment: From principles to algorithms. In Applications in Time-Frequency Signal Processing (Vol. 10, pp. 179-204). CRC Press.

[2] Fulop, S. A., & Fitz, K. (2006). Algorithms for computing the time-corrected instantaneous frequency (reassigned) spectrogram, with applications. The Journal of the Acoustical Society of America, 119(1), 360. doi:10.1121/1.2133000

[3] Auger, F., Flandrin, P., Lin, Y.-T., McLaughlin, S., Meignen, S., Oberlin, T., & Wu, H.-T. (2013). Time-Frequency Reassignment and Synchrosqueezing: An Overview. IEEE Signal Processing Magazine, 30(6), 32-41. doi:10.1109/MSP.2013.2265316

[4] Hainsworth, S., Macleod, M. (2003). Time-frequency reassignment: a review and analysis. Tech. Rep. CUED/FINFENG/TR.459, Cambridge University Engineering Department

Parameters

y : np.ndarray [shape=(n,)], real-valued

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) complex STFT calculated using the other arguments provided to `reassigned_spectrogram`

n_fft : int > 0 [scalar]

FFT window size. Defaults to 2048.

hop_length : int > 0 [scalar]

hop length, number samples between subsequent frames. If not supplied, defaults to *win_length / 4*.

win_length : int > 0, <= n_fft

Window length. Defaults to *n_fft*. See `stft` for details.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a user-specified window vector of length *n_fft*

See `stft` for details.

center : boolean

- If *True* (default), the signal *y* is padded so that frame *D[:, t]* is centered at *y[t * hop_length]*.
- If *False*, then *D[:, t]* begins at *y[t * hop_length]*

reassign_frequencies : boolean

- If *True* (default), the returned frequencies will be instantaneous frequency estimates.
- If *False*, the returned frequencies will be a read-only view of the STFT bin frequencies for all frames.

reassign_times : boolean

- If *True* (default), the returned times will be corrected (reassigned) time estimates for each bin.
- If *False*, the returned times will be a read-only view of the STFT frame times for all bins.

ref_power : float >= 0 or callable

Minimum power threshold for estimating time-frequency reassignments. Any bin with *np.abs(S[f, t])**2 < ref_power* will be returned as *np.nan* in both frequency and time, unless *fill_nan* is *True*. If 0 is provided, then only bins with zero power will be returned as *np.nan* (unless *fill_nan=True*).

fill_nan : boolean

- If *False* (default), the frequency and time reassignments for bins below the power threshold provided in *ref_power* will be returned as *np.nan*.
- If *True*, the frequency and time reassignments for these bins will be returned as the bin center frequencies and frame times.

clip : boolean

- If *True* (default), estimated frequencies outside the range *[0, 0.5 * sr]* or times outside the range *[0, len(y) / sr]* will be clipped to those ranges.
- If *False*, estimated frequencies and times beyond the bounds of the spectrogram may be returned.

dtype : numeric type

Complex numeric type for STFT calculation. Default is 64-bit complex.

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

Returns: **freqs** : np.ndarray [shape=(1 + n_fft/2, t), dtype=real]

Instantaneous frequencies: *freqs[f, t]* is the frequency for bin *f*, frame *t* If *reassign_frequencies=False*, this will instead be a read-only array of the same shape containing the bin center frequencies for all frames.

times : np.ndarray [shape=(1 + n_fft/2, t), dtype=real]

Reassigned times: *times[f, t]* is the time for bin *f*, frame *t* If *reassign_times=False*, this will instead be a read-only array of the same shape containing the frame times for all bins.

mags : np.ndarray [shape=(1 + n_fft/2, t), dtype=real]

Magnitudes from short-time Fourier transform: *mags[f, t]* is the magnitude for bin *f*, frame *t*

Warns: RuntimeWarning

Frequency or time estimates with zero support will produce a divide-by-zero warning, and will be returned as *np.nan* unless *fill_nan=True*.

See also

`stft`
Short-time Fourier Transform

Examples

```
>>> amin = 1e-10
>>> n_fft = 64
>>> sr = 4000
>>> y = 1e-3 * librosa.clicks(times=[0.3], sr=sr, click_duration=1.0,
...                           click_freq=1200.0, length=8000) +\
```

```
...     1e-3 * librosa.clicks(times=[1.5], sr=sr, click_duration=0.5,
...                           click_freq=400.0, length=8000) +\
...     1e-3 * librosa.chirp(200, 1600, sr=sr, duration=2.0) +\
...     1e-6 * np.random.randn(2*sr)
>>> freqs, times, mags = librosa.reassigned_spectrogram(y=y, sr=sr, n_fft=n_fft)
>>> mags_db = librosa.power_to_db(mags, amin=amin)
>>> ax = plt.subplot(2, 1, 1)
>>> librosa.display.specshow(mags_db, x_axis="s", y_axis="linear", sr=sr,
...                          hop_length=n_fft//4, cmap="gray_r")
>>> plt.title("Spectrogram")
>>> plt.tick_params(axis='x', labelbottom=False)
>>> plt.xlabel("")
>>> plt.subplot(2, 1, 2, sharex=ax, sharey=ax)
>>> plt.scatter(times, freqs, c=mags_db, alpha=0.05, cmap="gray_r")
>>> plt.clim(10*np.log10(amin), np.max(mags_db))
>>> plt.title("Reassigned spectrogram"))
```

librosa.core.cqt

`librosa.core.cqt`(*y*, *sr=22050*, *hop_length=512*, *fmin=None*, *n_bins=84*, *bins_per_octave=12*, *tuning=0.0*, *filter_scale=1*, *norm=1*, *sparsity=0.01*, *window='hann'*, *scale=True*, *pad_mode='reflect'*, *res_type=None*)[source]

Compute the constant-Q transform of an audio signal.

This implementation is based on the recursive sub-sampling method described by [1].

[1] Schoerkhuber, Christian, and Anssi Klapuri. “Constant-Q transform toolbox for music processing.” 7th Sound and Music Computing Conference, Barcelona, Spain. 2010.

Parameters **y** : np.ndarray [shape=(n,)]

:

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

hop_length : int > 0 [scalar]

number of samples between successive CQT columns.

fmin : float > 0 [scalar]

Minimum frequency. Defaults to C1 ~= 32.70 Hz

n_bins : int > 0 [scalar]

Number of frequency bins, starting at *fmin*

bins_per_octave : int > 0 [scalar]

Number of bins per octave

tuning : None or float

Tuning offset in fractions of a bin.

If *None*, tuning will be automatically estimated from the signal.

The minimum frequency of the resulting CQT will be modified to *fmin * 2**(tuning / bins_per_octave)*.

filter_scale : float > 0

Filter scale factor. Small values (<1) use shorter windows for improved time resolution.

norm : {inf, -inf, 0, float > 0}

Type of norm to use for basis function normalization. See `librosa.util.normalize`.

sparsity : float in [0, 1)

Sparsify the CQT basis by discarding up to *sparsity* fraction of the energy in each basis.

Set *sparsity=0* to disable sparsification.

window : str, tuple, number, or function

Window specification for the basis filters. See *filters.get_window* for details.

scale : bool

If *True*, scale the CQT response by square-root the length of each channel's filter. This is analogous to *norm='ortho'* in FFT.

If *False*, do not scale the CQT. This is analogous to *norm=None* in FFT.

pad_mode : string

Padding mode for centered frame analysis.

See also: `librosa.core.stft` and *np.pad*.

res_type : string [optional]

The resampling mode for recursive downsampling.

By default, `cqt` will adaptively select a resampling mode which trades off accuracy at high frequencies for efficiency at low frequencies.

You can override this by specifying a resampling mode as supported by librosa.core.resample. For example, *res_type='fft'* will use a high-quality, but potentially slow FFT-based down-sampling, while *res_type='polyphase'* will use a fast, but potentially inaccurate down-sampling.

Returns: **CQT** : np.ndarray [shape=(n_bins, t), dtype=np.complex or np.float]

Constant-Q value each frequency at each time.

Raises: ParameterError

If *hop_length* is not an integer multiple of *2**(n_bins / bins_per_octave)*

Or if *y* is too short to support the frequency range of the CQT.

See also

librosa.core.resample
librosa.util.normalize

Notes

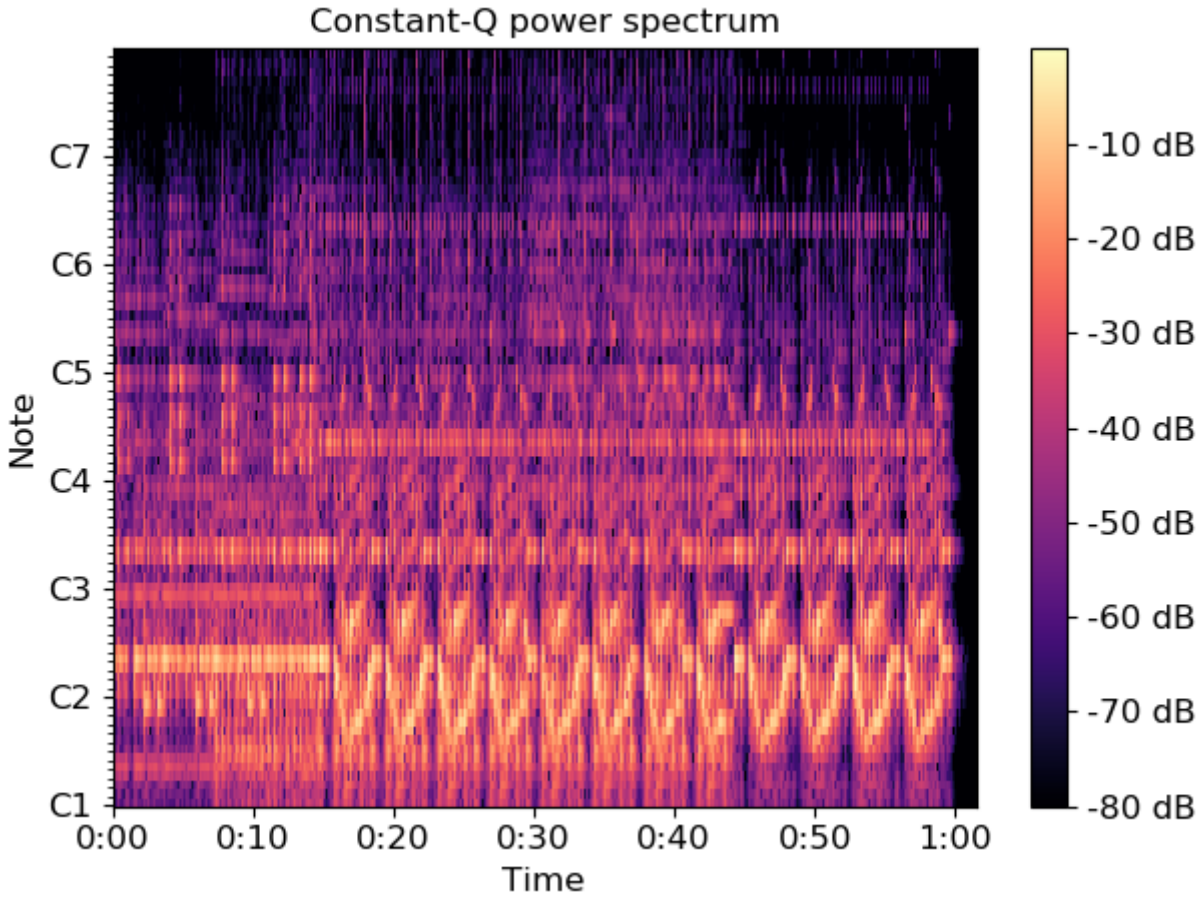
This function caches at level 20.

Examples

Generate and plot a constant-Q power spectrum

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> C = np.abs(librosa.cqt(y, sr=sr))
>>> librosa.display.specshow(librosa.amplitude_to_db(C, ref=np.max),
...                          sr=sr, x_axis='time', y_axis='cqt_note')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Constant-Q power spectrum')
>>> plt.tight_layout()
>>> plt.show()
```

(Source code)



Limit the frequency range

```
>>> C = np.abs(librosa.cqt(y, sr=sr, fmin=librosa.note_to_hz('C2'),
...                 n_bins=60))
>>> C
array([[  8.827e-04,   9.293e-04, ...,   3.133e-07,   2.942e-07],
       [  1.076e-03,   1.068e-03, ...,   1.153e-06,   1.148e-06],
       ...,
       [  1.042e-07,   4.087e-07, ...,   1.612e-07,   1.928e-07],
       [  2.363e-07,   5.329e-07, ...,   1.294e-07,   1.611e-07]])
```

Using a higher frequency resolution

```
>>> C = np.abs(librosa.cqt(y, sr=sr, fmin=librosa.note_to_hz('C2'),
...                 n_bins=60 * 2, bins_per_octave=12 * 2))
>>> C
array([[  1.536e-05,   5.848e-05, ...,   3.241e-07,   2.453e-07],
       [  1.856e-03,   1.854e-03, ...,   2.397e-08,   3.549e-08],
       ...,
       [  2.034e-07,   4.245e-07, ...,   6.213e-08,   1.463e-07],
       [  4.896e-08,   5.407e-07, ...,   9.176e-08,   1.051e-07]])
```

librosa.core.icqt

`librosa.core.icqt`(*C, sr=22050, hop_length=512, fmin=None, bins_per_octave=12, tuning=0.0, filter_scale=1, norm=1, sparsity=0.01, window='hann', scale=True, length=None, amin=<DEPRECATED parameter>, res_type='fft', dtype=<class 'numpy.float32'>*)[source]

Compute the inverse constant-Q transform.

Given a constant-Q transform representation *C* of an audio signal *y*, this function produces an approximation *y_hat*

Parameters: C : np.ndarray, [shape=(n_bins, n_frames)]

Constant-Q representation as produced by *core.cqt*

hop_length : int > 0 [scalar]

number of samples between successive frames

fmin : float > 0 [scalar]

Minimum frequency. Defaults to C1 ~= 32.70 Hz

tuning : float [scalar]

Tuning offset in fractions of a bin.

The minimum frequency of the CQT will be modified to *fmin * 2**(tuning / bins_per_octave)*.

filter_scale : float > 0 [scalar]

Filter scale factor. Small values (<1) use shorter windows for improved time resolution.

norm : {inf, -inf, 0, float > 0}

Type of norm to use for basis function normalization. See `librosa.util.normalize`.

sparsity : float in [0, 1)

Sparsify the CQT basis by discarding up to *sparsity* fraction of the energy in each basis.

Set *sparsity=0* to disable sparsification.

window : str, tuple, number, or function

Window specification for the basis filters. See *filters.get_window* for

details.

scale : bool

If *True*, scale the CQT response by square-root the length of each channel's filter. This is analogous to *norm='ortho'* in FFT.

If *False*, do not scale the CQT. This is analogous to *norm=None* in FFT.

length : int > 0, optional

If provided, the output *y* is zero-padded or clipped to exactly *length* samples.

amin : float or None [DEPRECATED]

Note

This parameter is deprecated in 0.7.0 and will be removed in 0.8.0.

res_type : string

Resampling mode. By default, this uses `fft` mode for high-quality reconstruction, but this may be slow depending on your signal duration. See `librosa.resample` for supported modes.

dtype : numeric type

Real numeric type for *y*. Default is 32-bit float.

Returns: **y** : np.ndarray, [shape=(n_samples), dtype=np.float]

Audio time-series reconstructed from the CQT representation.

See also

`cqt`
`core.resample`

Notes

This function caches at level 40.

Examples

Using default parameters

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=15)
>>> C = librosa.cqt(y=y, sr=sr)
>>> y_hat = librosa.icqt(C=C, sr=sr)
```

Or with a different hop length and frequency resolution:

```
>>> hop_length = 256
>>> bins_per_octave = 12 * 3
>>> C = librosa.cqt(y=y, sr=sr, hop_length=256, n_bins=7*bins_per_octave,
...                 bins_per_octave=bins_per_octave)
>>> y_hat = librosa.icqt(C=C, sr=sr, hop_length=hop_length,
...                 bins_per_octave=bins_per_octave)
```

librosa.core.hybrid_cqt

`librosa.core.hybrid_cqt`(*y, sr=22050, hop_length=512, fmin=None, n_bins=84, bins_per_octave=12, tuning=0.0, filter_scale=1, norm=1, sparsity=0.01, window='hann', scale=True, pad_mode='reflect', res_type=None*)[source]

Compute the hybrid constant-Q transform of an audio signal.

Here, the hybrid CQT uses the pseudo CQT for higher frequencies where the hop_length is longer than half the filter length and the full CQT for lower frequencies.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

hop_length : int > 0 [scalar]

number of samples between successive CQT columns.

fmin : float > 0 [scalar]

Minimum frequency. Defaults to C1 ~= 32.70 Hz

n_bins : int > 0 [scalar]

Number of frequency bins, starting at *fmin*

bins_per_octave : int > 0 [scalar]

Number of bins per octave

tuning : None or float

Tuning offset in fractions of a bin.

If *None*, tuning will be automatically estimated from the signal.

The minimum frequency of the resulting CQT will be modified to *fmin * 2**(tuning / bins_per_octave)*.

filter_scale : float > 0

Filter filter_scale factor. Larger values use longer windows.

sparsity : float in [0, 1)

Sparsify the CQT basis by discarding up to *sparsity* fraction of the energy in each basis.

Set *sparsity=0* to disable sparsification.

window : str, tuple, number, or function

Window specification for the basis filters. See *filters.get_window* for details.

pad_mode : string

Padding mode for centered frame analysis.

See also: `librosa.core.stft` and *np.pad*.

res_type : string

Resampling mode. See `librosa.core.cqt` for details.

Returns: **CQT** : np.ndarray [shape=(n_bins, t), dtype=np.float]

Constant-Q energy for each frequency at each time.

Raises: ParameterError

If *hop_length* is not an integer multiple of *2**(n_bins / bins_per_octave)*

Or if *y* is too short to support the frequency range of the CQT.

See also

`cqt`
`pseudo_cqt`

Notes

This function caches at level 20.

librosa.core.pseudo_cqt

`librosa.core.pseudo_cqt`(*y, sr=22050, hop_length=512, fmin=None, n_bins=84, bins_per_octave=12, tuning=0.0, filter_scale=1, norm=1, sparsity=0.01, window='hann', scale=True, pad_mode='reflect'*)[source]

Compute the pseudo constant-Q transform of an audio signal.

This uses a single fft size that is the smallest power of 2 that is greater than or equal to the max of:

1. The longest CQT filter
2. 2x the hop_length

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

hop_length : int > 0 [scalar]

number of samples between successive CQT columns.

fmin : float > 0 [scalar]

Minimum frequency. Defaults to C1 ~= 32.70 Hz

n_bins : int > 0 [scalar]

Number of frequency bins, starting at *fmin*

bins_per_octave : int > 0 [scalar]

Number of bins per octave

tuning : None or float

Tuning offset in fractions of a bin.

If *None*, tuning will be automatically estimated from the signal.

The minimum frequency of the resulting CQT will be modified to *fmin * 2**(tuning / bins_per_octave)*.

filter_scale : float > 0

Filter filter_scale factor. Larger values use longer windows.

sparsity : float in [0, 1)

Sparsify the CQT basis by discarding up to *sparsity* fraction of the energy in each basis.

Set *sparsity=0* to disable sparsification.

window : str, tuple, number, or function

Window specification for the basis filters. See *filters.get_window* for details.

pad_mode : string

Padding mode for centered frame analysis.

See also: `librosa.core.stft` and *np.pad*.

Returns: **CQT** : np.ndarray [shape=(n_bins, t), dtype=np.float]

Pseudo Constant-Q energy for each frequency at each time.

Raises: ParameterError

If *hop_length* is not an integer multiple of *2**(n_bins / bins_per_octave)*

Or if *y* is too short to support the frequency range of the CQT.

Notes

This function caches at level 20.

librosa.core.iirt

`librosa.core.iirt`(*y*, *sr=22050*, *win_length=2048*, *hop_length=None*, *center=True*, *tuning=0.0*, *pad_mode='reflect'*, *flayout='sos'*, ***kwargs*)[source]

Time-frequency representation using IIR filters [1].

This function will return a time-frequency representation using a multirate filter bank consisting of IIR filters. First, *y* is resampled as needed according to the provided *sample_rates*. Then, a filterbank with with *n* band-pass filters is designed. The resampled input signals are processed by the filterbank as a whole. (`scipy.signal.filtfilt` resp. *sosfiltfilt* is used to make the phase linear.) The output of the filterbank is cut into frames. For each band, the short-time mean-square power (STMSP) is calculated by summing *win_length* subsequent filtered time samples.

When called with the default set of parameters, it will generate the TF-representation as described in [1] (pitch filterbank):

- 85 filters with MIDI pitches [24, 108] as *center_freqs*.
- each filter having a bandwith of one semitone.

[1] *(1, 2)* Müller, Meinard. “Information Retrieval for Music and Motion.” Springer Verlag. 2007.

Parameters :

y : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

win_length : int > 0, <= n_fft

Window length.

hop_length : int > 0 [scalar]

Hop length, number samples between subsequent frames. If not supplied, defaults to *win_length / 4*.

center : boolean

- If *True*, the signal *y* is padded so that frame *D[:, t]* is centered at *y[t * hop_length]*.
- If *False*, then *D[:, t]* begins at *y[t * hop_length]*

tuning : float [scalar]

Tuning deviation from A440 in fractions of a bin.

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, this function uses reflection padding.

flayout : string

- If *sos* (default), a series of second-order filters is used for filtering with `scipy.signal.sosfiltfilt`. Minimizes numerical precision errors for high-order filters, but is slower.
- If *ba*, the standard difference equation is used for filtering with `scipy.signal.filtfilt`. Can be unstable for high-order filters.

kwargs : additional keyword arguments

Additional arguments for *librosa.filters.semitone_filterbank()* (e.g., could be used to provide another set of *center_freqs* and *sample_rates*).

Returns:

bands_power : np.ndarray [shape=(n, t), dtype=dtype]

Short-time mean-square power for the input signal.

Raises:

ParameterError

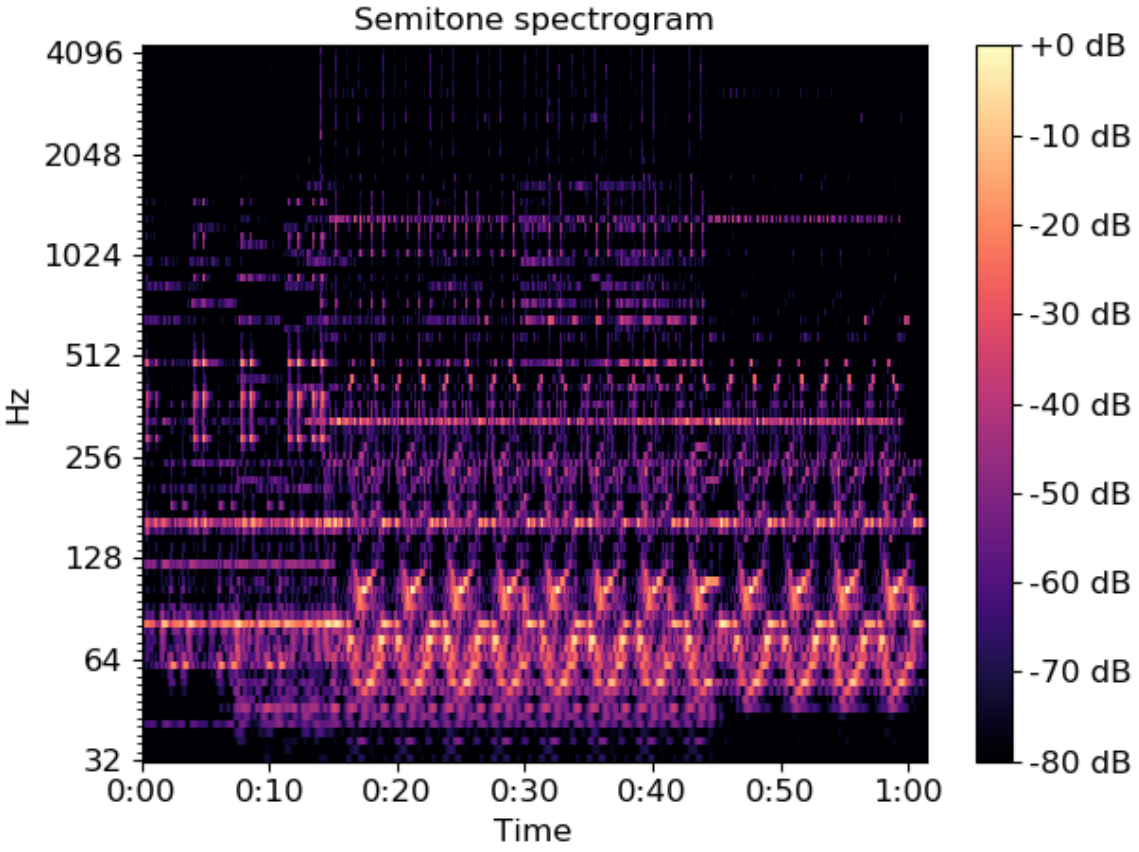
If *flayout* is not None, *ba*, or *sos*.

See also

librosa.filters.semitone_filterbank
librosa.filters._multirate_fb
librosa.filters.mr_frequencies
librosa.core.cqt
scipy.signal.filtfilt
scipy.signal.sosfiltfilt

Examples

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> D = np.abs(librosa.iirt(y))
>>> librosa.display.specshow(librosa.amplitude_to_db(D, ref=np.max),
...                          y_axis='cqt_hz', x_axis='time')
>>> plt.title('Semitone spectrogram')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.fmt

`librosa.core.fmt`(*y, t_min=0.5, n_fmt=None, kind='cubic', beta=0.5, over_sample=1, axis=-1*)[source]

The fast Mellin transform (FMT) [1] of a uniformly sampled signal y.

When the Mellin parameter (beta) is 1/2, it is also known as the scale transform [2]. The scale transform can be useful for audio analysis because its magnitude is invariant to scaling of the

domain (e.g., time stretching or compression). This is analogous to the magnitude of the Fourier transform being invariant to shifts in the input domain.

[1] De Sena, Antonio, and Davide Rocchesso. “A fast Mellin and scale transform.” EURASIP Journal on Applied Signal Processing 2007.1 (2007): 75-75.

[2] Cohen, L. “The scale representation.” IEEE Transactions on Signal Processing 41, no. 12 (1993): 3275-3292.

Parameters: **y** : np.ndarray, real-valued

The input signal(s). Can be multidimensional. The target axis must contain at least 3 samples.

t_min : float > 0

The minimum time spacing (in samples). This value should generally be less than 1 to preserve as much information as possible.

n_fmt : int > 2 or None

The number of scale transform bins to use. If None, then *n_bins = over_sample * ceil(n * log((n-1)/t_min))* is taken, where *n = y.shape[axis]*

kind : str

The type of interpolation to use when re-sampling the input. See `scipy.interpolate.interp1d` for possible values.

Note that the default is to use high-precision (cubic) interpolation. This can be slow in practice; if speed is preferred over accuracy, then consider using *kind=’linear’*.

beta : float

The Mellin parameter. *beta=0.5* provides the scale transform.

over_sample : float >= 1

Over-sampling factor for exponential resampling.

axis : int

The axis along which to transform *y*

Returns: **x_scale** : np.ndarray [dtype=complex]

The scale transform of *y* along the *axis* dimension.

Raises: ParameterError

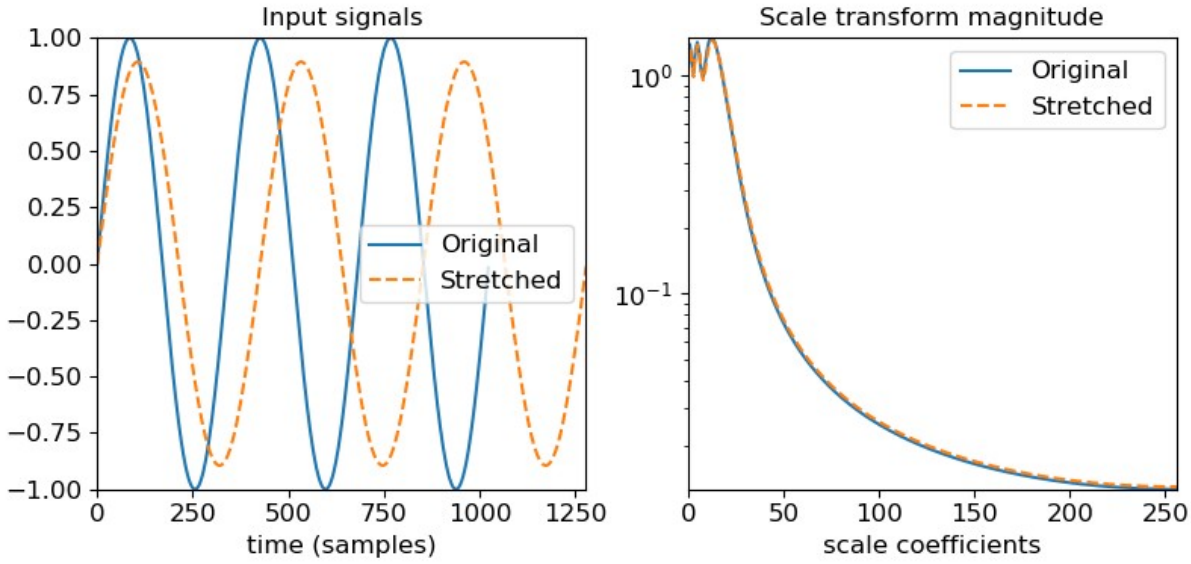
if *n_fmt* < *2* or *t_min* <= *0* or if *y* is not finite or if *y.shape[axis]* < *3*.

Notes

This function caches at level 30.

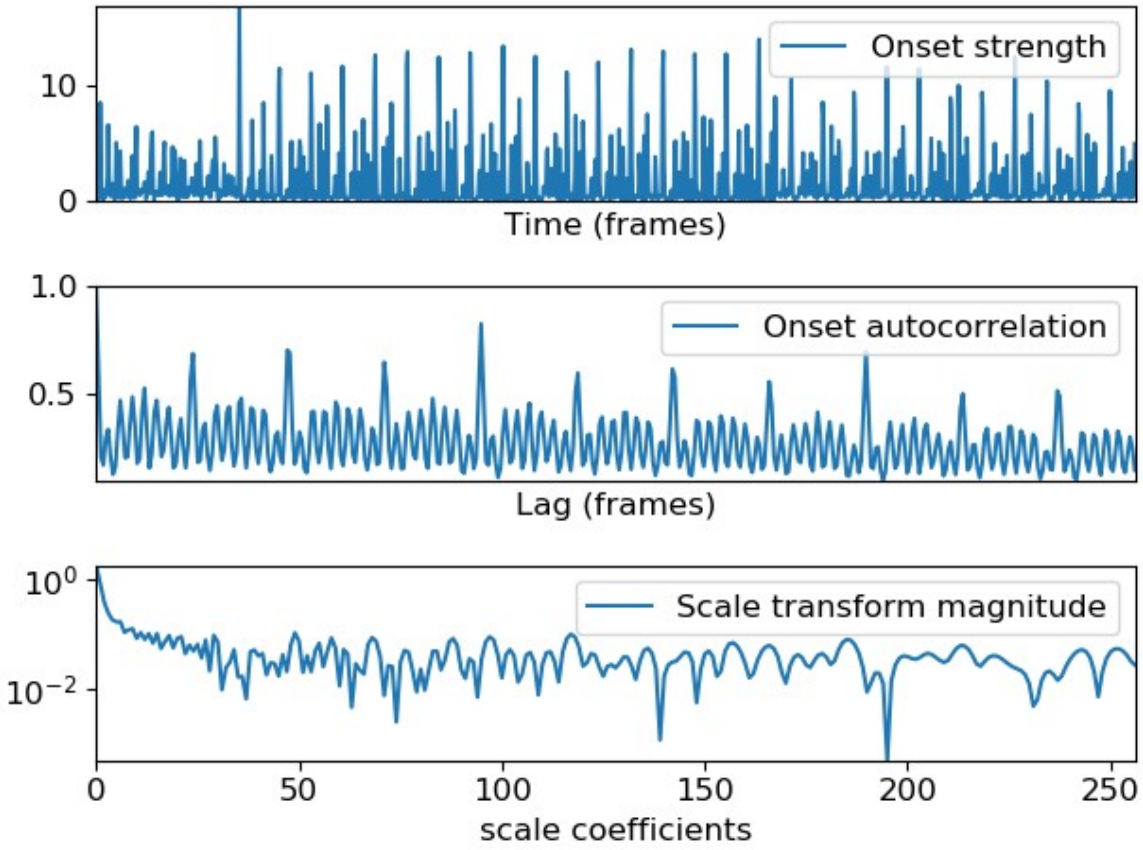
Examples

```
>>> # Generate a signal and time-stretch it (with energy normalization)
>>> scale = 1.25
>>> freq = 3.0
>>> x1 = np.linspace(0, 1, num=1024, endpoint=False)
>>> x2 = np.linspace(0, 1, num=scale * len(x1), endpoint=False)
>>> y1 = np.sin(2 * np.pi * freq * x1)
>>> y2 = np.sin(2 * np.pi * freq * x2) / np.sqrt(scale)
>>> # Verify that the two signals have the same energy
>>> np.sum(np.abs(y1)**2), np.sum(np.abs(y2)**2)
    (255.99999999999997, 255.99999999999969)
>>> scale1 = librosa.fmt(y1, n_fmt=512)
>>> scale2 = librosa.fmt(y2, n_fmt=512)
>>> # And plot the results
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(1, 2, 1)
>>> plt.plot(y1, label='Original')
>>> plt.plot(y2, linestyle='--', label='Stretched')
>>> plt.xlabel('time (samples)')
>>> plt.title('Input signals')
>>> plt.legend(frameon=True)
>>> plt.axis('tight')
>>> plt.subplot(1, 2, 2)
>>> plt.semilogy(np.abs(scale1), label='Original')
>>> plt.semilogy(np.abs(scale2), linestyle='--', label='Stretched')
>>> plt.xlabel('scale coefficients')
>>> plt.title('Scale transform magnitude')
>>> plt.legend(frameon=True)
>>> plt.axis('tight')
>>> plt.tight_layout()
>>> plt.show()
```




```
>>> # Plot the scale transform of an onset strength autocorrelation
```

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10.0, duration=30.0)
>>> odf = librosa.onset.onset_strength(y=y, sr=sr)
>>> # Auto-correlate with up to 10 seconds lag
>>> odf_ac = librosa.autocorrelate(odf, max_size=10 * sr // 512)
>>> # Normalize
>>> odf_ac = librosa.util.normalize(odf_ac, norm=np.inf)
>>> # Compute the scale transform
>>> odf_ac_scale = librosa.fmt(librosa.util.normalize(odf_ac), n_fmt=512)
>>> # Plot the results
>>> plt.figure()
>>> plt.subplot(3, 1, 1)
>>> plt.plot(odf, label='Onset strength')
>>> plt.axis('tight')
>>> plt.xlabel('Time (frames)')
>>> plt.xticks([])
>>> plt.legend(frameon=True)
>>> plt.subplot(3, 1, 2)
>>> plt.plot(odf_ac, label='Onset autocorrelation')
>>> plt.axis('tight')
>>> plt.xlabel('Lag (frames)')
>>> plt.xticks([])
>>> plt.legend(frameon=True)
>>> plt.subplot(3, 1, 3)
>>> plt.semilogy(np.abs(odf_ac_scale), label='Scale transform magnitude')
>>> plt.axis('tight')
>>> plt.xlabel('scale coefficients')
>>> plt.legend(frameon=True)
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.griffinlim

`librosa.core.griffinlim`(*S, n_iter=32, hop_length=None, win_length=None, window='hann', center=True, dtype=<class 'numpy.float32'>, length=None, pad_mode='reflect', momentum=0.99, init='random', random_state=None*)[source]

Approximate magnitude spectrogram inversion using the “fast” Griffin-Lim algorithm [1] [2].

Given a short-time Fourier transform magnitude matrix (*S*), the algorithm randomly initializes phase estimates, and then alternates forward- and inverse-STFT operations. Note that this assumes reconstruction of a real-valued time-domain signal, and that *S* contains only the non-negative frequencies (as computed by *core.stft*).

[1] *(1, 2)* Perraudin, N., Balazs, P., & Søndergaard, P. L. “A fast Griffin-Lim algorithm,” IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (pp. 1-4), Oct. 2013.

[2] D. W. Griffin and J. S. Lim, “Signal estimation from modified short-time Fourier transform,” IEEE Trans. ASSP, vol.32, no.2, pp.236–243, Apr. 1984.

Parameters: S : np.ndarray [shape=(n_fft / 2 + 1, t), non-negative]

An array of short-time Fourier transform magnitudes as produced by *core.stft*.

n_iter : int > 0

The number of iterations to run

hop_length : None or int > 0

The hop length of the STFT. If not provided, it will default to *n_fft // 4*

win_length : None or int > 0

The window length of the STFT. By default, it will equal *n_fft*

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

A window specification as supported by `stft` or `istft`

center : boolean

If *True*, the STFT is assumed to use centered frames. If *False*, the STFT is assumed to use left-aligned frames.

dtype : np.dtype

Real numeric type for the time-domain signal. Default is 32-bit float.

length : None or int > 0

If provided, the output *y* is zero-padded or clipped to exactly *length* samples.

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

momentum : number >= 0

The momentum parameter for fast Griffin-Lim. Setting this to 0 recovers the original Griffin-Lim method [1]. Values near 1 can lead to faster convergence, but above 1 may not converge.

init : None or ‘random’ [default]

If ‘random’ (the default), then phase values are initialized randomly according to *random_state*. This is recommended when the input *S* is a magnitude spectrogram with no initial phase estimates.

If *None*, then the phase is initialized from *S*. This is useful when an initial guess for phase can be provided, or when you want to resume Griffin-Lim from a previous output.

random_state : None, int, or np.random.RandomState

If int, random_state is the seed used by the random number generator for phase initialization.

If *np.random.RandomState* instance, the random number generator itself.

If *None*, defaults to the current *np.random* object.

Returns: **y** : np.ndarray [shape=(n,)]

time-domain signal reconstructed from *S*

See also

`stft`
`istft`
`magphase`
`filters.get_window`

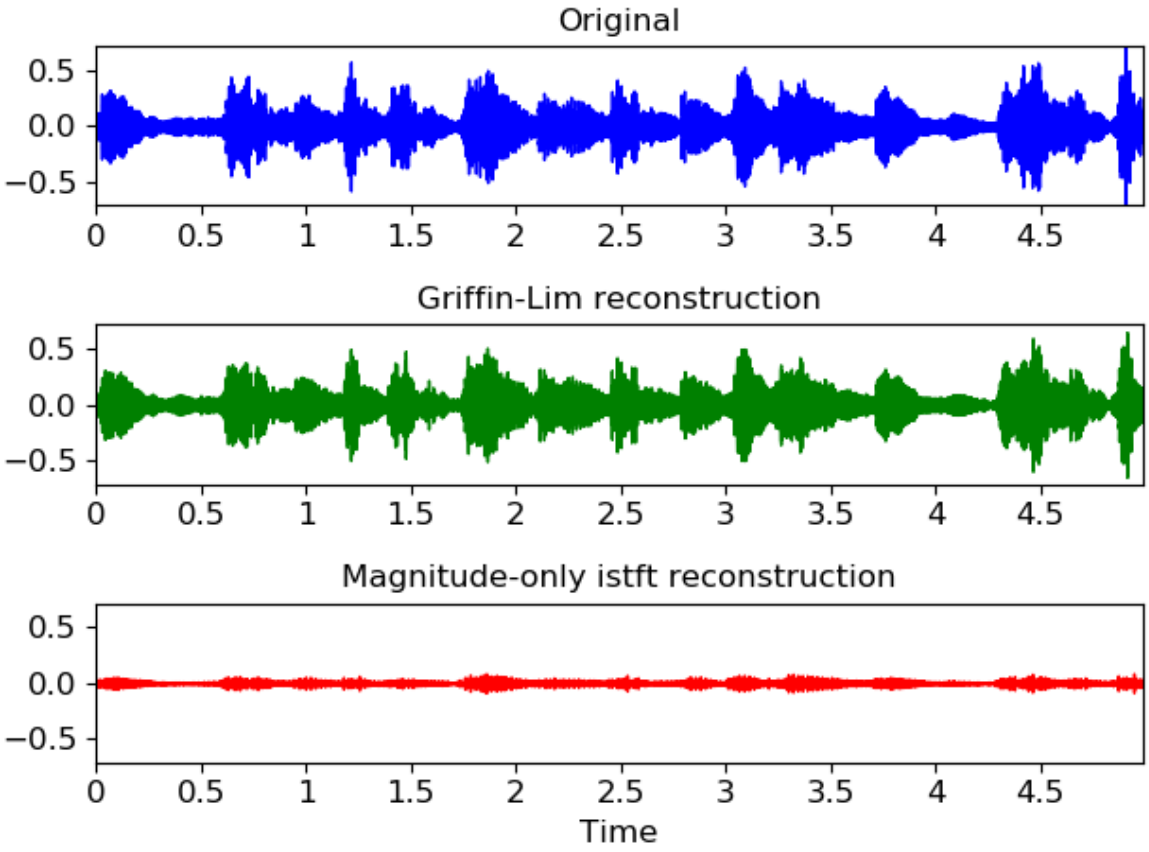
Examples

A basic STFT inverse example

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=5,
offset=30)
>>> # Get the magnitude spectrogram
>>> S = np.abs(librosa.stft(y))
>>> # Invert using Griffin-Lim
>>> y_inv = librosa.griffinlim(S)
>>> # Invert without estimating phase
>>> y_istft = librosa.istft(S)
```

Wave-plot the results

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> ax = plt.subplot(3,1,1)
>>> librosa.display.waveplot(y, sr=sr, color='b')
>>> plt.title('Original')
>>> plt.xlabel('')
>>> plt.subplot(3,1,2, sharex=ax, sharey=ax)
>>> librosa.display.waveplot(y_inv, sr=sr, color='g')
>>> plt.title('Griffin-Lim reconstruction')
>>> plt.xlabel('')
>>> plt.subplot(3,1,3, sharex=ax, sharey=ax)
>>> librosa.display.waveplot(y_istft, sr=sr, color='r')
>>> plt.title('Magnitude-only istft reconstruction')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.griffinlim_cqt

`librosa.core.griffinlim_cqt`(*C*, *n_iter=32*, *sr=22050*, *hop_length=512*, *fmin=None*, *bins_per_octave=12*, *tuning=0.0*, *filter_scale=1*, *norm=1*, *sparsity=0.01*, *window='hann'*, *scale=True*, *pad_mode='reflect'*, *res_type='kaiser_fast'*, *dtype=<class 'numpy.float32'>*, *length=None*, *momentum=0.99*, *init='random'*, *random_state=None*)[source]

Approximate constant-Q magnitude spectrogram inversion using the “fast” Griffin-Lim algorithm [1] [2].

Given the magnitude of a constant-Q spectrogram (*C*), the algorithm randomly initializes phase estimates, and then alternates forward- and inverse-CQT operations.

This implementation is based on the Griffin-Lim method for Short-time Fourier Transforms, but adapted for use with constant-Q spectrograms.

[1] *(1, 2)* Perraudin, N., Balazs, P., & Søndergaard, P. L. “A fast Griffin-Lim algorithm,” IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (pp. 1-4), Oct. 2013.

[2] D. W. Griffin and J. S. Lim, “Signal estimation from modified short-time Fourier transform,” IEEE Trans. ASSP, vol.32, no.2, pp.236–243, Apr. 1984.

Parameters: C : np.ndarray [shape=(n_bins, n_frames)]

The constant-Q magnitude spectrogram

n_iter : int > 0

The number of iterations to run

sr : number > 0

Audio sampling rate

hop_length : int > 0

The hop length of the CQT

fmin : number > 0

Minimum frequency for the CQT.

If not provided, it defaults to C1.

bins_per_octave : int > 0

Number of bins per octave

tuning : float

Tuning deviation from A440, in fractions of a bin

filter_scale : float > 0

Filter scale factor. Small values (<1) use shorter windows for improved time resolution.

norm : {inf, -inf, 0, float > 0}

Type of norm to use for basis function normalization. See `librosa.util.normalize`.

sparsity : float in [0, 1)

Sparsify the CQT basis by discarding up to *sparsity* fraction of the energy in each basis.

Set *sparsity=0* to disable sparsification.

window : str, tuple, or function

Window specification for the basis filters. See *filters.get_window* for details.

scale : bool

If *True*, scale the CQT response by square-root the length of each

channel's filter. This is analogous to *norm='ortho'* in FFT.

If *False*, do not scale the CQT. This is analogous to *norm=None* in FFT.

pad_mode : string

Padding mode for centered frame analysis.

See also: `librosa.core.stft` and *np.pad*

res_type : string

The resampling mode for recursive downsampling.

By default, CQT uses an adaptive mode selection to trade accuracy at high frequencies for efficiency at low frequencies.

Griffin-Lim uses the efficient (fast) resampling mode by default.

See `librosa.core.resample` for a list of available options.

dtype : numeric type

Real numeric type for *y*. Default is 32-bit float.

length : int > 0, optional

If provided, the output *y* is zero-padded or clipped to exactly *length* samples.

momentum : float > 0

The momentum parameter for fast Griffin-Lim. Setting this to 0 recovers the original Griffin-Lim method [1]. Values near 1 can lead to faster convergence, but above 1 may not converge.

init : None or 'random' [default]

If 'random' (the default), then phase values are initialized randomly according to *random_state*. This is recommended when the input *C* is a magnitude spectrogram with no initial phase estimates.

If *None*, then the phase is initialized from *C*. This is useful when an initial guess for phase can be provided, or when you want to resume Griffin-Lim from a previous output.

random_state : None, int, or np.random.RandomState

If int, random_state is the seed used by the random number generator for phase initialization.

If *np.random.RandomState* instance, the random number generator itself.

If *None*, defaults to the current *np.random* object.

Returns: **y** : np.ndarray [shape=(n,)]

time-domain signal reconstructed from *C*

See also

`cqt`
`icqt`
`griffinlim`
`filters.get_window`
`resample`

Examples

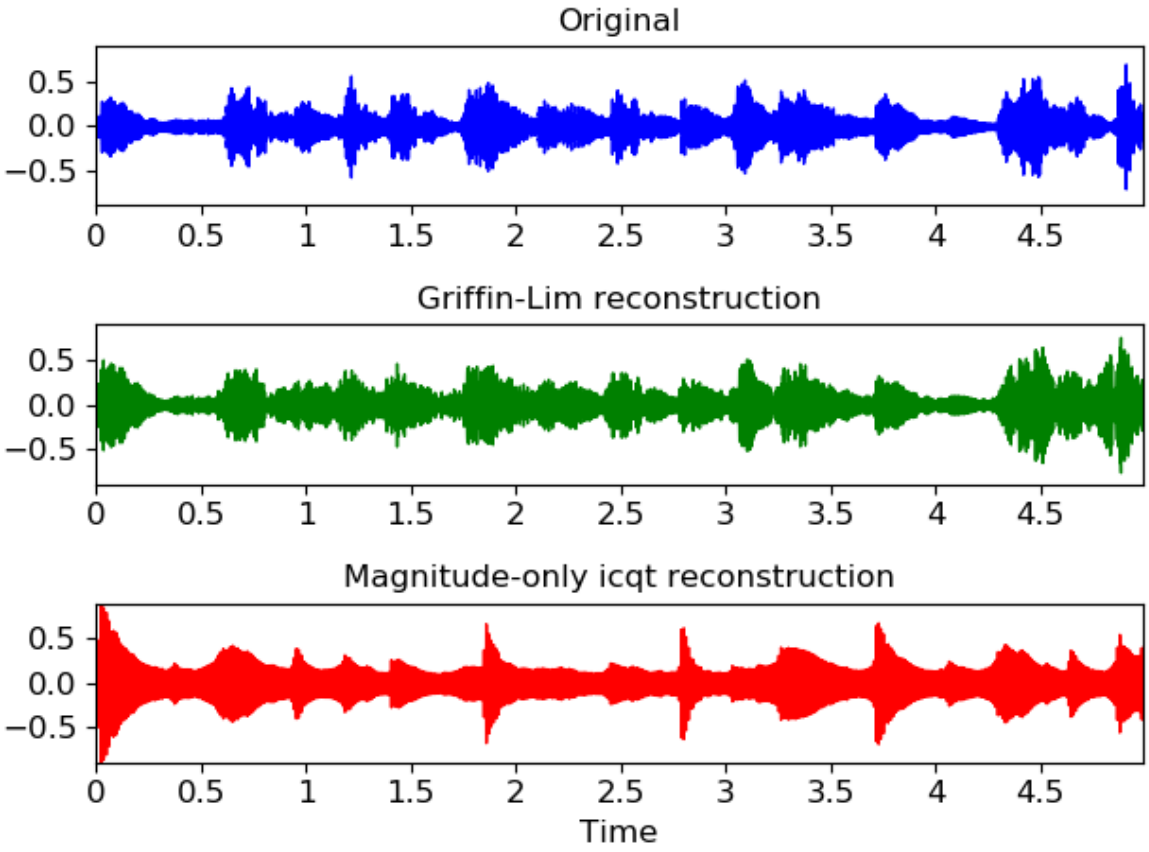
A basis CQT inverse example

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=5,
offset=30, sr=None)
>>> # Get the CQT magnitude, 7 octaves at 36 bins per octave
>>> C = np.abs(librosa.cqt(y=y, sr=sr, bins_per_octave=36, n_bins=7*36))
>>> # Invert using Griffin-Lim
>>> y_inv = librosa.griffinlim_cqt(C, sr=sr, bins_per_octave=36)
>>> # And invert without estimating phase
>>> y_icqt = librosa.icqt(C, sr=sr, bins_per_octave=36)
```

Wave-plot the results

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> ax = plt.subplot(3,1,1)
>>> librosa.display.waveplot(y, sr=sr, color='b')
>>> plt.title('Original')
>>> plt.xlabel('')
>>> plt.subplot(3,1,2, sharex=ax, sharey=ax)
>>> librosa.display.waveplot(y_inv, sr=sr, color='g')
>>> plt.title('Griffin-Lim reconstruction')
>>> plt.xlabel('')
>>> plt.subplot(3,1,3, sharex=ax, sharey=ax)
>>> librosa.display.waveplot(y_icqt, sr=sr, color='r')
>>> plt.title('Magnitude-only icqt reconstruction')
>>> plt.tight_layout()
>>> plt.show()
```

(Source code)



librosa.core.interp_harmonics

`librosa.core.interp_harmonics`(*x*, *freqs*, *h_range*, *kind='linear'*, *fill_value=0*, *axis=0*) [source]

Compute the energy at harmonics of time-frequency representation.

Given a frequency-based energy representation such as a spectrogram or tempogram, this function computes the energy at the chosen harmonics of the frequency axis. (See examples below.) The resulting harmonic array can then be used as input to a salience computation.

Parameters: **x** : np.ndarray

The input energy

freqs : np.ndarray, shape=(X.shape[axis])

The frequency values corresponding to X's elements along the chosen axis.

h_range : list-like, non-negative

Harmonics to compute. The first harmonic (1) corresponds to *x* itself. Values less than one (e.g., 1/2) correspond to sub-harmonics.

kind : str

Interpolation type. See `scipy.interpolate.interp1d`.

fill_value : float

The value to fill when extrapolating beyond the observed frequency range.

axis : int

The axis along which to compute harmonics

Returns: **x_harm** : np.ndarray, shape=(len(h_range), [x.shape])

x_harm[i] will have the same shape as *x*, and measure the energy at the *h_range[i]* harmonic of each frequency.

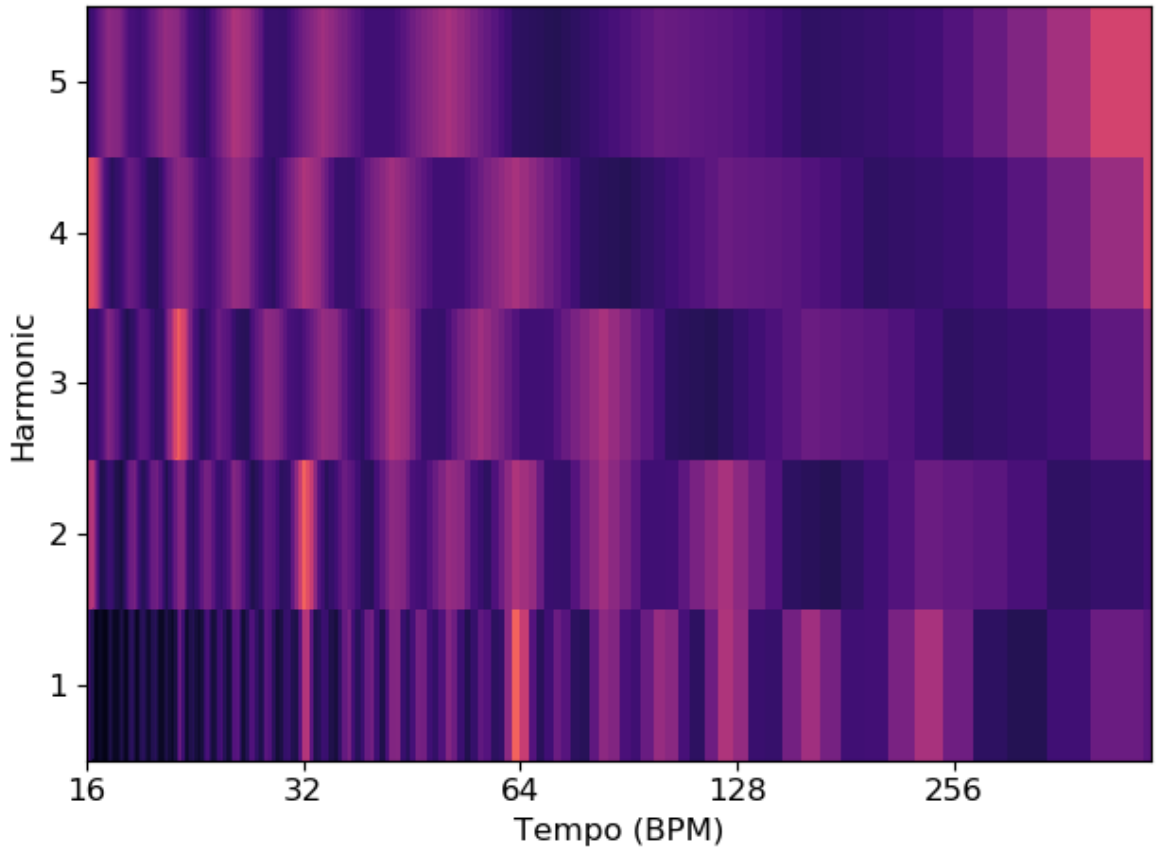
See also

`scipy.interpolate.interp1d`

Examples

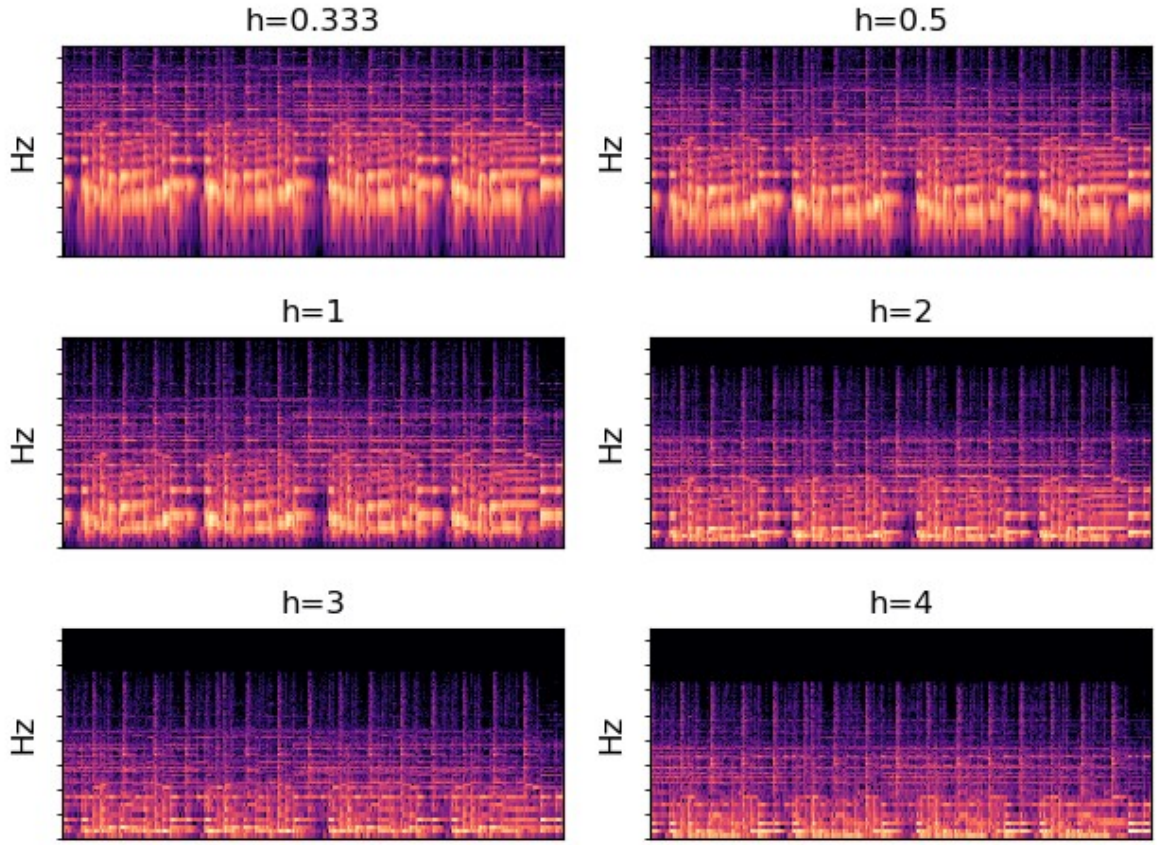
Estimate the harmonics of a time-averaged tempogram

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=15, offset=30)
>>> # Compute the time-varying tempogram and average over time
>>> tempi = np.mean(librosa.feature.tempogram(y=y, sr=sr), axis=1)
>>> # We'll measure the first five harmonics
>>> h_range = [1, 2, 3, 4, 5]
>>> f_tempo = librosa.tempo_frequencies(len(tempi), sr=sr)
>>> # Build the harmonic tensor
>>> t_harmonics = librosa.interp_harmonics(tempi, f_tempo, h_range)
>>> print(t_harmonics.shape)
(5, 384)

>>> # And plot the results
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> librosa.display.specshow(t_harmonics, x_axis='tempo', sr=sr)
>>> plt.yticks(0.5 + np.arange(len(h_range)),
...            ['{:.3g}'.format(_) for _ in h_range])
>>> plt.ylabel('Harmonic')
>>> plt.xlabel('Tempo (BPM)')
>>> plt.tight_layout()
>>> plt.show()
```



We can also compute frequency harmonics for spectrograms. To calculate sub-harmonic energy, use values < 1.

```
>>> h_range = [1./3, 1./2, 1, 2, 3, 4]
>>> S = np.abs(librosa.stft(y))
>>> fft_freqs = librosa.fft_frequencies(sr=sr)
>>> S_harm = librosa.interp_harmonics(S, fft_freqs, h_range, axis=0)
>>> print(S_harm.shape)
(6, 1025, 646)

>>> plt.figure()
>>> for i, _sh in enumerate(S_harm, 1):
...     plt.subplot(3, 2, i)
...     librosa.display.specshow(librosa.amplitude_to_db(_sh,
...                                                      ref=S.max()),
...                              sr=sr, y_axis='log')
...     plt.title('h={:.3g}'.format(h_range[i-1]))
...     plt.yticks([])
>>> plt.tight_layout()
```



librosa.core.salience

`librosa.core.salience`(*S, freqs, h_range, weights=None, aggregate=None, filter_peaks=True, fill_value=nan, kind='linear', axis=0*)[source]

Harmonic salience function.

Parameters **S** : np.ndarray [shape=(d, n)]

:

input time frequency magnitude representation (e.g. STFT or CQT magnitudes). Must be real-valued and non-negative.

freqs : np.ndarray, shape=(S.shape[axis])

The frequency values corresponding to S's elements along the chosen axis.

h_range : list-like, non-negative

Harmonics to include in salience computation. The first harmonic (1) corresponds to *S* itself. Values less than one (e.g., 1/2) correspond to sub-harmonics.

weights : list-like

The weight to apply to each harmonic in the summation. (default:

uniform weights). Must be the same length as *harmonics*.

aggregate : function

aggregation function (default: *np.average*) If *aggregate=np.average*, then a weighted average is computed per-harmonic according to the specified weights. For all other aggregation functions, all harmonics are treated equally.

filter_peaks : bool

If true, returns harmonic summation only on frequencies of peak magnitude. Otherwise returns harmonic summation over the full spectrum. Defaults to True.

fill_value : float

The value to fill non-peaks in the output representation. (default: np.nan) Only used if *filter_peaks == True*.

kind : str

Interpolation type for harmonic estimation. See `scipy.interpolate.interp1d`.

axis : int

The axis along which to compute harmonics

Returns: **S_sal** : np.ndarray, shape=(len(h_range), [x.shape])

S_sal will have the same shape as *S*, and measure the overal harmonic energy at each frequency.

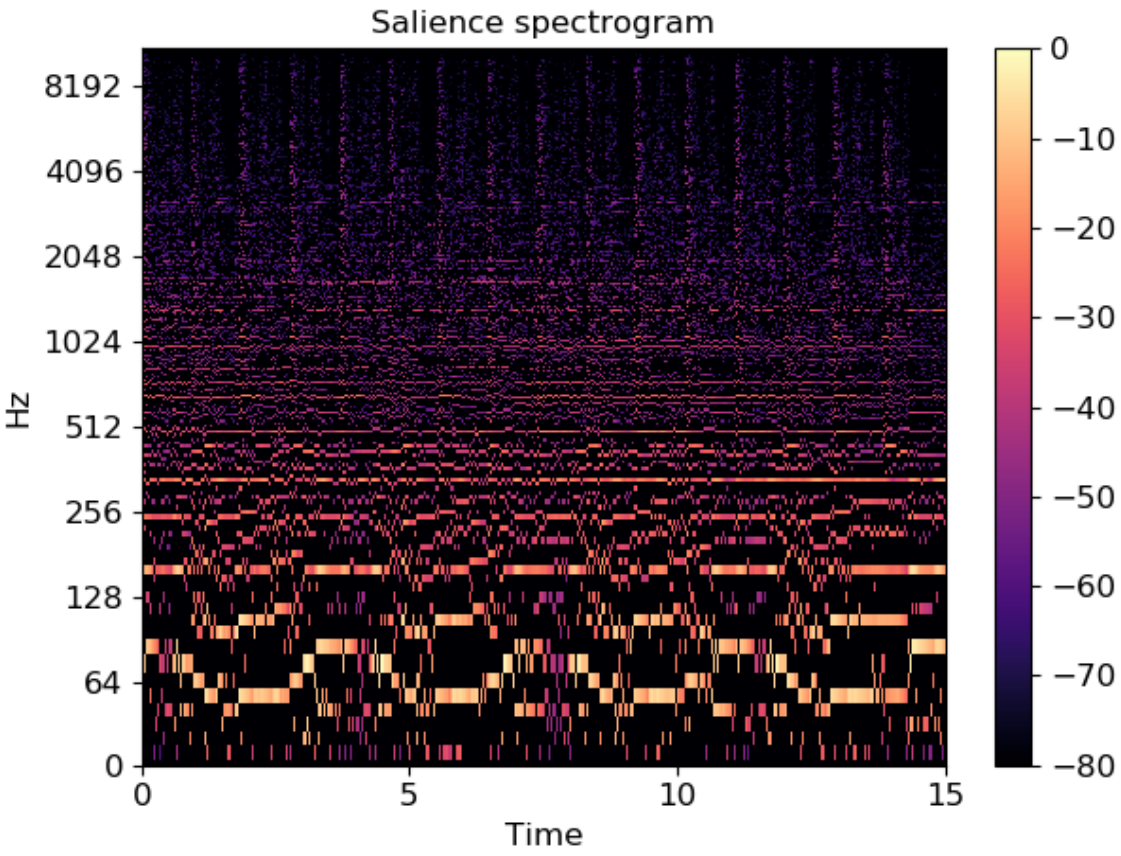
See also

`interp_harmonics`

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=15, offset=30)
>>> S = np.abs(librosa.stft(y))
>>> freqs = librosa.core.fft_frequencies(sr)
>>> harms = [1, 2, 3, 4]
>>> weights = [1.0, 0.5, 0.33, 0.25]
>>> S_sal = librosa.salience(S, freqs, harms, weights, fill_value=0)
>>> print(S_sal.shape)
(1025, 646)
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> librosa.display.specshow(librosa.amplitude_to_db(S_sal,
...                                                  ref=np.max),
...                          sr=sr, y_axis='log', x_axis='time')
>>> plt.colorbar()
```

```
>>> plt.title('Salience spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.phase_vocoder

`librosa.core.phase_vocoder`(*D*, *rate*, *hop_length=None*)[source]

Phase vocoder. Given an STFT matrix D, speed up by a factor of *rate*

Based on the implementation provided by [1].

Note

This is a simplified implementation, intended primarily for reference and pedagogical purposes. It makes no attempt to handle transients, and is likely to produce many audible artifacts. For a higher quality implementation, we recommend the RubberBand library [2] and its Python wrapper `pyrubberband`.

[1] Ellis, D. P. W. “A phase vocoder in Matlab.” Columbia University, 2002. http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/

[2] https://breakfastquay.com/rubberband/

Parameters: **D** : np.ndarray [shape=(d, t), dtype=complex]

STFT matrix

rate : float > 0 [scalar]

Speed-up factor: *rate > 1* is faster, *rate < 1* is slower.

hop_length : int > 0 [scalar] or None

The number of samples between successive columns of *D*.

If None, defaults to *n_fft/4 = (D.shape[0]-1)/2*

Returns: **D_stretched** : np.ndarray [shape=(d, t / rate), dtype=complex]

time-stretched STFT

See also

`pyrubberband`

Examples

```
>>> # Play at double speed
>>> y, sr   = librosa.load(librosa.util.example_audio_file())
>>> D       = librosa.stft(y, n_fft=2048, hop_length=512)
>>> D_fast  = librosa.phase_vocoder(D, 2.0, hop_length=512)
>>> y_fast  = librosa.istft(D_fast, hop_length=512)

>>> # Or play at 1/3 speed
>>> y, sr   = librosa.load(librosa.util.example_audio_file())
>>> D       = librosa.stft(y, n_fft=2048, hop_length=512)
>>> D_slow  = librosa.phase_vocoder(D, 1./3, hop_length=512)
>>> y_slow  = librosa.istft(D_slow, hop_length=512)
```

librosa.core.magphase

`librosa.core.magphase`(*D*, *power=1*)[source]

Separate a complex-valued spectrogram D into its magnitude (S) and phase (P) components, so that *D = S * P*.

Parameters: **D** : np.ndarray [shape=(d, t), dtype=complex]

complex-valued spectrogram

power : float > 0

Exponent for the magnitude spectrogram, e.g., 1 for energy, 2 for power, etc.

Returns: **D_mag** : np.ndarray [shape=(d, t), dtype=real]

magnitude of *D*, raised to *power*

D_phase : np.ndarray [shape=(d, t), dtype=complex]

*exp(1.j * phi)* where *phi* is the phase of *D*

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> D = librosa.stft(y)
>>> magnitude, phase = librosa.magphase(D)
>>> magnitude
array([[  2.524e-03,   4.329e-02, ...,   3.217e-04,   3.520e-05],
       [  2.645e-03,   5.152e-02, ...,   3.283e-04,   3.432e-04],
       ...,
       [  1.966e-05,   9.828e-06, ...,   3.164e-07,   9.370e-06],
       [  1.966e-05,   9.830e-06, ...,   3.161e-07,   9.366e-06]],
dtype=float32)
>>> phase
array([[  1.000e+00 +0.000e+00j,   1.000e+00 +0.000e+00j, ...,
         -1.000e+00 +8.742e-08j,  -1.000e+00 +8.742e-08j],
       [  1.000e+00 +1.615e-16j,   9.950e-01 -1.001e-01j, ...,
          9.794e-01 +2.017e-01j,   1.492e-02 -9.999e-01j],
       ...,
       [  1.000e+00 -5.609e-15j,  -5.081e-04 +1.000e+00j, ...,
         -9.549e-01 -2.970e-01j,   2.938e-01 -9.559e-01j],
       [ -1.000e+00 +8.742e-08j,  -1.000e+00 +8.742e-08j, ...,
         -1.000e+00 +8.742e-08j,  -1.000e+00 +8.742e-08j]],
dtype=complex64)
```

Or get the phase angle (in radians)

```
>>> np.angle(phase)
array([[  0.000e+00,   0.000e+00, ...,   3.142e+00,   3.142e+00],
       [  1.615e-16,  -1.003e-01, ...,   2.031e-01,  -1.556e+00],
       ...,
       [ -5.609e-15,   1.571e+00, ...,  -2.840e+00,  -1.273e+00],
       [  3.142e+00,   3.142e+00, ...,   3.142e+00,   3.142e+00]],
dtype=float32)
```

librosa.core.get_fftlib

`librosa.core.get_fftlib`()[source]

Get the FFT library currently used by librosa

Returns: **fft** : module

The FFT library currently used by librosa. Must API-compatible with `numpy.fft`.

librosa.core.set_fftlib

`librosa.core.set_fftlib`(*lib=None*)[source]

Set the FFT library used by librosa.

Parameters: **lib** : None or module

Must implement an interface compatible with `numpy.fft`. If *None*, reverts to `numpy.fft`.

Examples

Use `pyfftw`:

```
>>> import pyfftw
>>> librosa.set_fftlib(pyfftw.interfaces.numpy_fft)
```

Reset to default `numpy` implementation

```
>>> librosa.set_fftlib()librosa.core.amplitude_to_db
```

`librosa.core.amplitude_to_db`(*S*, *ref=1.0*, *amin=1e-05*, *top_db=80.0*)[source]

Convert an amplitude spectrogram to dB-scaled spectrogram.

This is equivalent to `power_to_db(S**2)`, but is provided for convenience.

Parameters: **S** : np.ndarray

input amplitude

ref : scalar or callable

If scalar, the amplitude *abs(S)* is scaled relative to *ref*: *20 * log10(S / ref)*. Zeros in the output correspond to positions where *S == ref*.

If callable, the reference value is computed as *ref(S)*.

amin : float > 0 [scalar]

minimum threshold for *S* and *ref*

top_db : float >= 0 [scalar]

threshold the output at *top_db* below the peak: `max(20 * log10(S)) - top_db`

Returns: **S_db** : np.ndarray

`S` measured in dB

See also

`power_to_db`, `db_to_amplitude`

Notes

This function caches at level 30.

librosa.core.db_to_amplitude

`librosa.core.db_to_amplitude`(*S_db*, *ref=1.0*)[source]

Convert a dB-scaled spectrogram to an amplitude spectrogram.

This effectively inverts `amplitude_to_db`:

*db_to_amplitude(S_db) ~= 10.0**(0.5 * (S_db + log10(ref)/10))*

Parameters: **S_db** : np.ndarray

dB-scaled spectrogram

ref: number > 0

Optional reference power.

Returns: **S** : np.ndarray

Linear magnitude spectrogram

Notes

This function caches at level 30.

librosa.core.power_to_db

`librosa.core.power_to_db`(*S*, *ref=1.0*, *amin=1e-10*, *top_db=80.0*)[source]

Convert a power spectrogram (amplitude squared) to decibel (dB) units

This computes the scaling `10 * log10(S / ref)` in a numerically stable way.

Parameters: **S** : np.ndarray

input power

ref : scalar or callable

If scalar, the amplitude *abs(S)* is scaled relative to *ref*: *10 * log10(S / ref)*.

Zeros in the output correspond to positions where *S == ref*.

If callable, the reference value is computed as *ref(S)*.

amin : float > 0 [scalar]

minimum threshold for *abs(S)* and *ref*

top_db : float >= 0 [scalar]

threshold the output at *top_db* below the peak: `max(10 * log10(S)) - top_db`

Returns: **S_db** : np.ndarray

`S_db ~= 10 * log10(S) - 10 * log10(ref)`

See also

`perceptual_weighting`
`db_to_power`
`amplitude_to_db`
`db_to_amplitude`

Notes

This function caches at level 30.

Examples

Get a power spectrogram from a waveform `y`

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
>>> librosa.power_to_db(S**2)
array([[-33.293, -27.32 , ..., -33.293, -33.293],
       [-33.293, -25.723, ..., -33.293, -33.293],
       ...,
       [-33.293, -33.293, ..., -33.293, -33.293],
       [-33.293, -33.293, ..., -33.293, -33.293]], dtype=float32)
```

Compute dB relative to peak power

```
>>> librosa.power_to_db(S**2, ref=np.max)
array([[-80.   , -74.027, ..., -80.   , -80.   ],
       [-80.   , -72.431, ..., -80.   , -80.   ],
       ...,
       [-80.   , -80.   , ..., -80.   , -80.   ],
       [-80.   , -80.   , ..., -80.   , -80.   ]], dtype=float32)
```

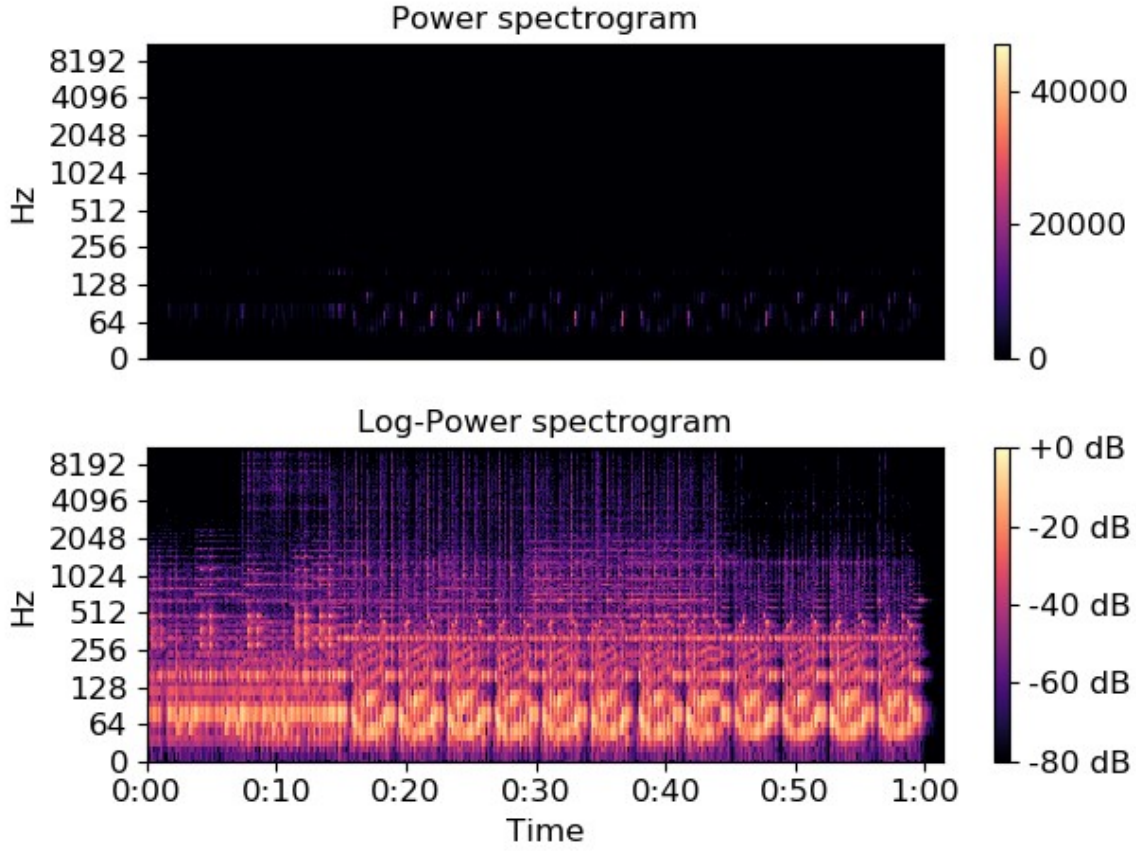
Or compare to median power

```
>>> librosa.power_to_db(S**2, ref=np.median)
array([[-0.189,  5.784, ..., -0.189, -0.189],
       [-0.189,  7.381, ..., -0.189, -0.189],
```

```
       ...,
       [-0.189, -0.189, ..., -0.189, -0.189],
       [-0.189, -0.189, ..., -0.189, -0.189]], dtype=float32)
```

And plot the results

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(S**2, sr=sr, y_axis='log')
>>> plt.colorbar()
>>> plt.title('Power spectrogram')
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.power_to_db(S**2, ref=np.max),
...                          sr=sr, y_axis='log', x_axis='time')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Log-Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.db_to_power

`librosa.core.db_to_power`(*S_db*, *ref=1.0*)[source]

Convert a dB-scale spectrogram to a power spectrogram.

This effectively inverts `power_to_db`:

*db_to_power(S_db) ~= ref * 10.0**(S_db / 10)*

Parameters: **S_db** : np.ndarray

dB-scaled spectrogram

ref : number > 0

Reference power: output will be scaled by this value

Returns: **S** : np.ndarray

Power spectrogram

Notes

This function caches at level 30.

librosa.core.perceptual_weighting

`librosa.core.perceptual_weighting`(*S*, *frequencies*, ***kwargs*)[source]

Perceptual weighting of a power spectrogram:

*S_p[f] = A_weighting(f) + 10*log(S[f] / ref)*

Parameters: **S** : np.ndarray [shape=(d, t)]

Power spectrogram

frequencies : np.ndarray [shape=(d,)]

Center frequency for each row of *S*

kwargs : additional keyword arguments

Additional keyword arguments to `power_to_db`.

Returns: **S_p** : np.ndarray [shape=(d, t)]

perceptually weighted version of *S*

See also

`power_to_db`

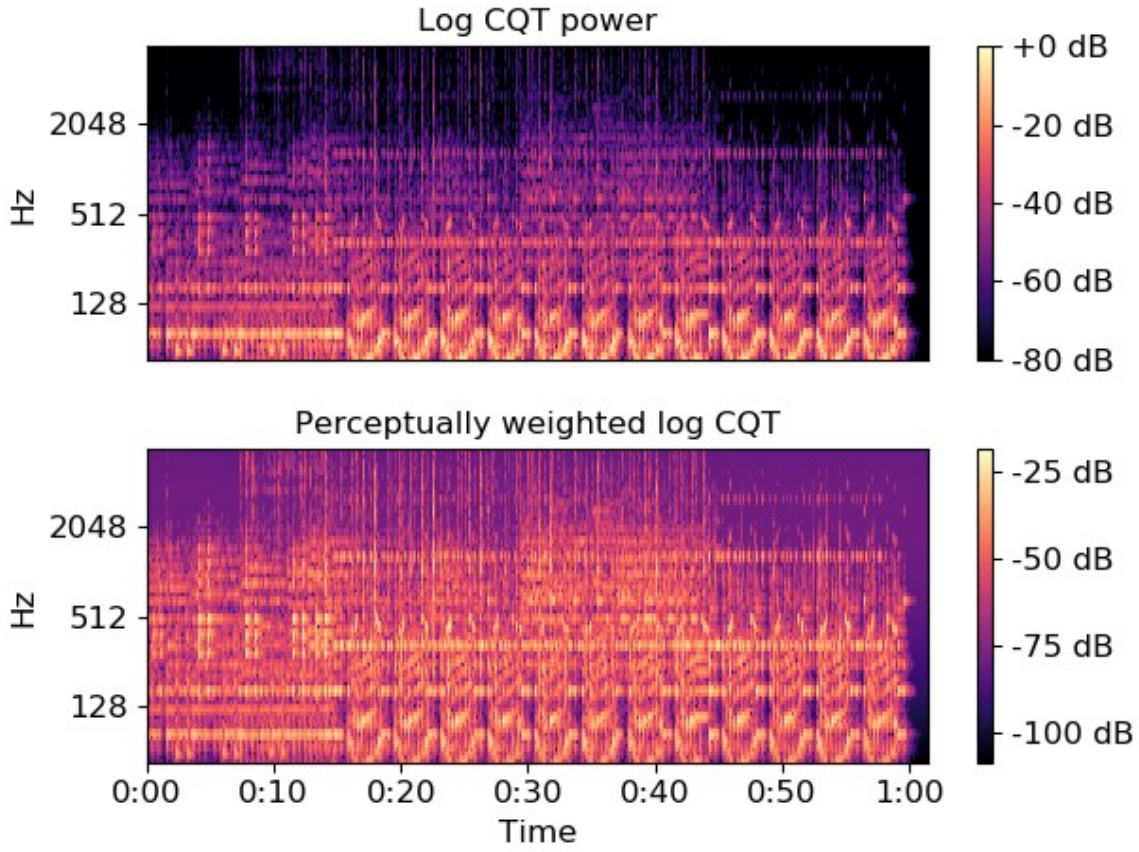
Notes

This function caches at level 30.

Examples

Re-weight a CQT power spectrum, using peak power as reference

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> C = np.abs(librosa.cqt(y, sr=sr, fmin=librosa.note_to_hz('A1')))
>>> freqs = librosa.cqt_frequencies(C.shape[0],
...                                 fmin=librosa.note_to_hz('A1'))
>>> perceptual_CQT = librosa.perceptual_weighting(C**2,
...                                               freqs,
...                                               ref=np.max)
>>> perceptual_CQT
array([[ -80.076,  -80.049, ..., -104.735, -104.735],
       [ -78.344,  -78.555, ..., -103.725, -103.725],
       ...,
       [ -76.272,  -76.272, ...,  -76.272,  -76.272],
       [ -76.485,  -76.485, ...,  -76.485,  -76.485]])

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(C,
...                                                  ref=np.max),
...                          fmin=librosa.note_to_hz('A1'),
...                          y_axis='cqt_hz')
>>> plt.title('Log CQT power')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(perceptual_CQT, y_axis='cqt_hz',
...                          fmin=librosa.note_to_hz('A1'),
...                          x_axis='time')
>>> plt.title('Perceptually weighted log CQT')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.A_weighting

`librosa.core.A_weighting`(*frequencies*, *min_db=-80.0*)[source]

Compute the A-weighting of a set of frequencies.

Parameters: **frequencies** : scalar or np.ndarray [shape=(n,)]

One or more frequencies (in Hz)

min_db : float [scalar] or None

Clip weights below this threshold. If *None*, no clipping is performed.

Returns: **A_weighting** : scalar or np.ndarray [shape=(n,)]

A_weighting[i] is the A-weighting of *frequencies[i]*

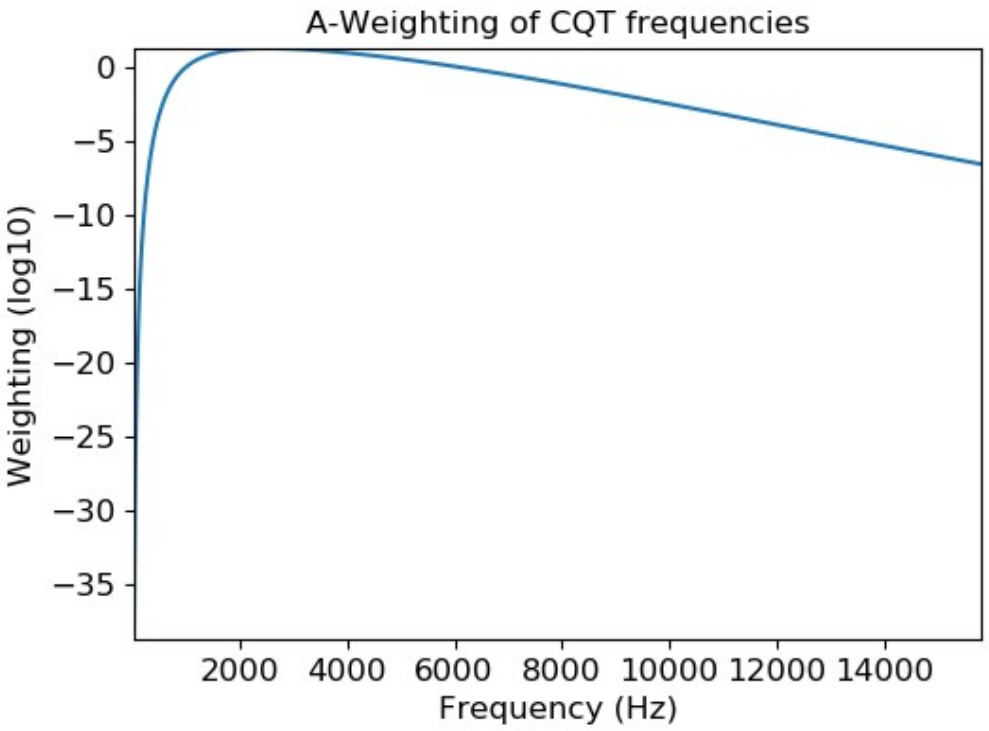
See also

`perceptual_weighting`

Examples

Get the A-weighting for CQT frequencies

```
>>> import matplotlib.pyplot as plt
>>> freqs = librosa.cqt_frequencies(108, librosa.note_to_hz('C1'))
>>> aw = librosa.A_weighting(freqs)
>>> plt.plot(freqs, aw)
>>> plt.xlabel('Frequency (Hz)')
>>> plt.ylabel('Weighting (log10)')
>>> plt.title('A-Weighting of CQT frequencies')
>>> plt.show()
```



librosa.core.pcen

`librosa.core.pcen`(*S, sr=22050, hop_length=512, gain=0.98, bias=2, power=0.5, time_constant=0.4, eps=1e-06, b=None, max_size=1, ref=None, axis=-1, max_axis=None, zi=None, return_zf=False*)[source]

Per-channel energy normalization (PCEN) [1]

This function normalizes a time-frequency representation *S* by performing automatic gain control, followed by nonlinear compression:

P[f, t] = (S / (eps + M[f, t])**gain + bias)**power - bias**power

IMPORTANT: the default values of eps, gain, bias, and power match the original publication [1], in which M is a 40-band mel-frequency spectrogram with 25 ms windowing, 10 ms frame shift, and raw audio values in the interval [-2**31; 2**31-1[. If you use these default values,

we recommend to make sure that the raw audio is properly scaled to this interval, and not to [-1, 1[as is most often the case.

The matrix *M* is the result of applying a low-pass, temporal IIR filter to *S*:

M[f, t] = (1 - b) * M[f, t - 1] + b * S[f, t]

If *b* is not provided, it is calculated as:

b = (sqrt(1 + 4* T**2) - 1) / (2 * T**2)

where *T = time_constant * sr / hop_length*, as in [2].

This normalization is designed to suppress background noise and emphasize foreground signals, and can be used as an alternative to decibel scaling (`amplitude_to_db`).

This implementation also supports smoothing across frequency bins by specifying *max_size > 1*. If this option is used, the filtered spectrogram *M* is computed as

M[f, t] = (1 - b) * M[f, t - 1] + b * R[f, t]

where *R* has been max-filtered along the frequency axis, similar to the SuperFlux algorithm implemented in *onset.onset_strength*:

R[f, t] = max(S[f - max_size//2: f + max_size//2, t])

This can be used to perform automatic gain control on signals that cross or span multiple frequency bans, which may be desirable for spectrograms with high frequency resolution.

[1] *(1, 2)* Wang, Y., Getreuer, P., Hughes, T., Lyon, R. F., & Saurous, R. A. (2017, March). Trainable frontend for robust and far-field keyword spotting. In Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on (pp. 5670-5674). IEEE.

[2] Lostanlen, V., Salamon, J., McFee, B., Cartwright, M., Farnsworth, A., Kelling, S., and Bello, J. P. Per-Channel Energy Normalization: Why and How. IEEE Signal Processing Letters, 26(1), 39-43.

Parameters: S : np.ndarray (non-negative)

The input (magnitude) spectrogram

sr : number > 0 [scalar]

The audio sampling rate

hop_length : int > 0 [scalar]

The hop length of *S*, expressed in samples

gain : number >= 0 [scalar]

The gain factor. Typical values should be slightly less than 1.

bias : number >= 0 [scalar]

The bias point of the nonlinear compression (default: 2)

power : number >= 0 [scalar]

The compression exponent. Typical values should be between 0 and 0.5. Smaller values of *power* result in stronger compression. At the limit *power=0*, polynomial compression becomes logarithmic.

time_constant : number > 0 [scalar]

The time constant for IIR filtering, measured in seconds.

eps : number > 0 [scalar]

A small constant used to ensure numerical stability of the filter.

b : number in [0, 1] [scalar]

The filter coefficient for the low-pass filter. If not provided, it will be inferred from *time_constant*.

max_size : int > 0 [scalar]

The width of the max filter applied to the frequency axis. If left as *1*, no filtering is performed.

ref : None or np.ndarray (shape=S.shape)

An optional pre-computed reference spectrum (*R* in the above). If not provided it will be computed from *S*.

axis : int [scalar]

The (time) axis of the input spectrogram.

max_axis : None or int [scalar]

The frequency axis of the input spectrogram. If *None*, and *S* is two-dimensional, it will be inferred as the opposite from *axis*. If *S* is not two-dimensional, and *max_size > 1*, an error will be raised.

zi : np.ndarray

The initial filter delay values.

This may be the *zf* (final delay values) of a previous call to `pcen`, or

computed by scipy.signal.lfilter_zi.

return_zf : bool

If *True*, return the final filter delay values along with the PCEN output *P*. This is primarily useful in streaming contexts, where the final state of one block of processing should be used to initialize the next block.

If *False* (default) only the PCEN values *P* are returned.

Returns: **P** : np.ndarray, non-negative [shape=(n, m)]

The per-channel energy normalized version of *S*.

zf : np.ndarray (optional)

The final filter delay values. Only returned if *return_zf=True*.

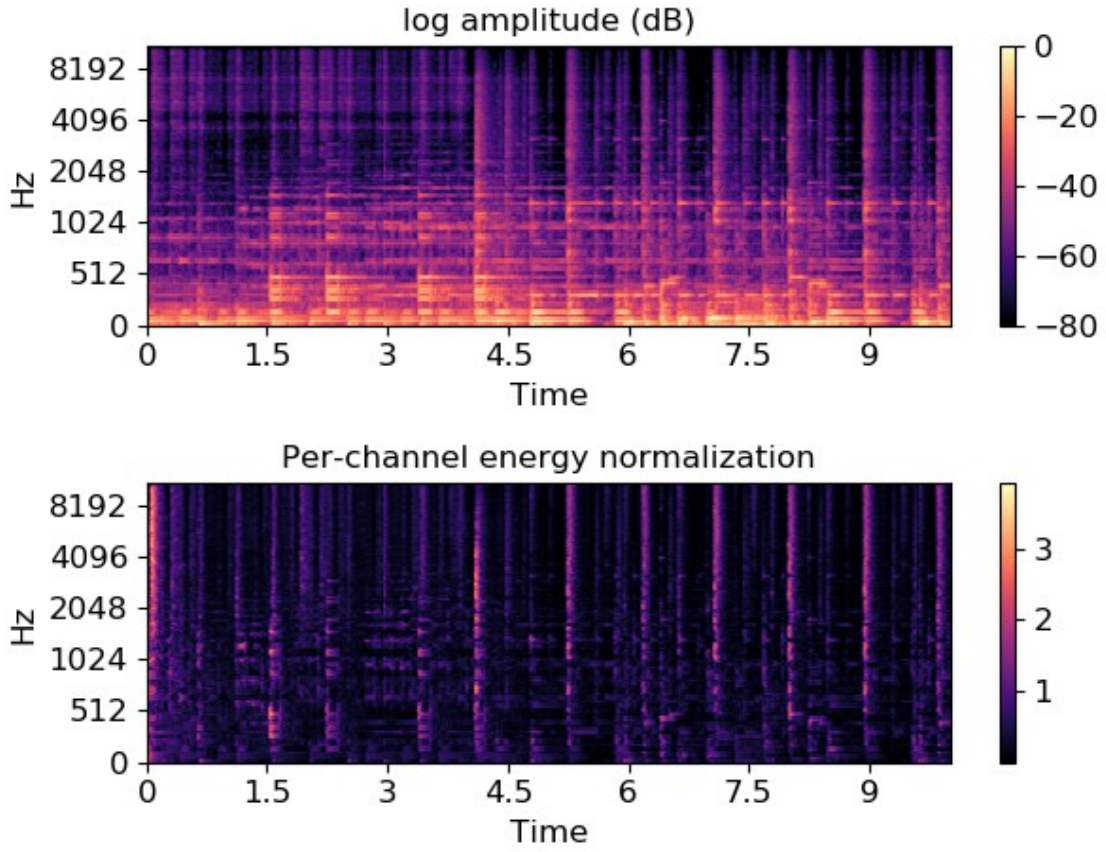
See also

amplitude_to_db
librosa.onset.onset_strength

Examples

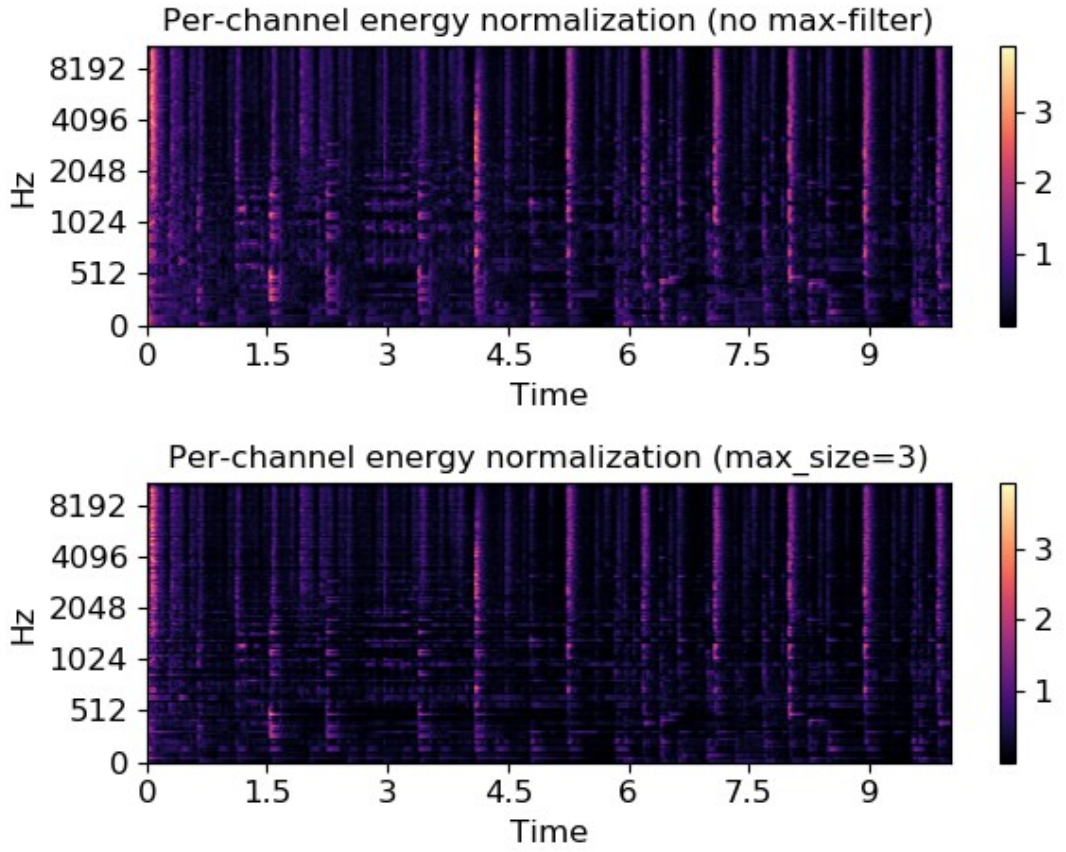
Compare PCEN to log amplitude (dB) scaling on Mel spectra

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10, duration=10)

>>> # We recommend scaling y to the range [-2**31, 2**31[ before applying
>>> # PCEN's default parameters. Furthermore, we use power=1 to get a
>>> # magnitude spectrum instead of a power spectrum.
>>> S = librosa.feature.melspectrogram(y, sr=sr, power=1)
>>> log_S = librosa.amplitude_to_db(S, ref=np.max)
>>> pcen_S = librosa.pcen(S * (2**31))
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(log_S, x_axis='time', y_axis='mel')
>>> plt.title('log amplitude (dB)')
>>> plt.colorbar()
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(pcen_S, x_axis='time', y_axis='mel')
>>> plt.title('Per-channel energy normalization')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



Compare PCEN with and without max-filtering

```
>>> pcen_max = librosa.pcen(S * (2**31), max_size=3)
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(pcen_S, x_axis='time', y_axis='mel')
>>> plt.title('Per-channel energy normalization (no max-filter)')
>>> plt.colorbar()
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(pcen_max, x_axis='time', y_axis='mel')
>>> plt.title('Per-channel energy normalization (max_size=3)')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.core.frames_to_samples

`librosa.core.frames_to_samples`(*frames, hop_length=512, n_fft=None*)[source]

Converts frame indices to audio sample indices.

Parameters: **frames** : number or np.ndarray [shape=(n,)]

frame index or vector of frame indices

hop_length : int > 0 [scalar]

number of samples between successive frames

n_fft : None or int > 0 [scalar]

Optional: length of the FFT window. If given, time conversion will include an offset of *n_fft / 2* to counteract windowing effects when using a non-centered STFT.

Returns: **times** : number or np.ndarray

time (in samples) of each given frame number: *times[i] = frames[i] * hop_length*

See also

`frames_to_time`
: convert frame indices to time values

`samples_to_frames`
: convert sample indices to frame indices

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> tempo, beats = librosa.beat.beat_track(y, sr=sr)
>>> beat_samples = librosa.frames_to_samples(beats)
```

librosa.core.frames_to_time

`librosa.core.frames_to_time`(*frames, sr=22050, hop_length=512, n_fft=None*)[source]

Converts frame counts to time (seconds).

Parameters: **frames** : np.ndarray [shape=(n,)]

frame index or vector of frame indices

sr : number > 0 [scalar]

audio sampling rate

hop_length : int > 0 [scalar]

number of samples between successive frames

n_fft : None or int > 0 [scalar]

Optional: length of the FFT window. If given, time conversion will include an offset of *n_fft / 2* to counteract windowing effects when using a non-centered STFT.

Returns: **times** : np.ndarray [shape=(n,)]

time (in seconds) of each given frame number: *times[i] = frames[i] * hop_length / sr*

See also

`time_to_frames`
: convert time values to frame indices

`frames_to_samples`
: convert frame indices to sample indices

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> tempo, beats = librosa.beat.beat_track(y, sr=sr)
>>> beat_times = librosa.frames_to_time(beats, sr=sr)
```

librosa.core.samples_to_frames

`librosa.core.samples_to_frames`(*samples*, *hop_length=512*, *n_fft=None*)[source]

Converts sample indices into STFT frames.

Parameters: **samples** : int or np.ndarray [shape=(n,)]

sample index or vector of sample indices

hop_length : int > 0 [scalar]

number of samples between successive frames

n_fft : None or int > 0 [scalar]

Optional: length of the FFT window. If given, time conversion will include an offset of - *n_fft / 2* to counteract windowing effects in STFT.

Note

This may result in negative frame indices.

Returns: **frames** : int or np.ndarray [shape=(n,), dtype=int]

Frame numbers corresponding to the given times: *frames[i] = floor(samples[i] / hop_length)*

See also

`samples_to_time`
convert sample indices to time values

`frames_to_samples`
convert frame indices to sample indices

Examples

```
>>> # Get the frame numbers for every 256 samples
>>> librosa.samples_to_frames(np.arange(0, 22050, 256))
array([ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5,  6,  6,
        7,  7,  8,  8,  9,  9, 10, 10, 11, 11, 12, 12, 13, 13,
       14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20,
       21, 21, 22, 22, 23, 23, 24, 24, 25, 25, 26, 26, 27, 27,
       28, 28, 29, 29, 30, 30, 31, 31, 32, 32, 33, 33, 34, 34,
       35, 35, 36, 36, 37, 37, 38, 38, 39, 39, 40, 40, 41, 41,
       42, 42, 43])
```

librosa.core.blocks_to_frames

`librosa.core.blocks_to_frames`(*blocks*, *block_length*)[source]

Convert block indices to frame indices

Parameters: **blocks** : np.ndarray

Block index or array of block indices

block_length : int > 0

The number of frames per block

Returns: **frames** : np.ndarray [shape=samples.shape, dtype=int]

The index or indices of frames corresponding to the beginning of each provided block.

See also

`blocks_to_samples`
`blocks_to_time`

Examples

Get frame indices for each block in a stream

```
>>> filename = librosa.util.example_audio_file()
>>> sr = librosa.get_samplerate(filename)
>>> stream = librosa.stream(filename, block_length=16,
...                         frame_length=2048, hop_length=512)
>>> for n, y in enumerate(stream):
...     n_frame = librosa.blocks_to_frames(n, block_length=16)
```

librosa.core.blocks_to_samples

`librosa.core.blocks_to_samples`(*blocks*, *block_length*, *hop_length*)[source]

Convert block indices to sample indices

Parameters: **blocks** : np.ndarray

Block index or array of block indices

block_length : int > 0

The number of frames per block

hop_length : int > 0

The number of samples to advance between frames

Returns: **samples** : np.ndarray [shape=samples.shape, dtype=int]

The index or indices of samples corresponding to the beginning of each

provided block.

Note that these correspond to the *first* sample index in each block, and are not frame-centered.

See also

`blocks_to_frames`
`blocks_to_time`

Examples

Get sample indices for each block in a stream

```
>>> filename = librosa.util.example_audio_file()
>>> sr = librosa.get_samplerate(filename)
>>> stream = librosa.stream(filename, block_length=16,
...                         frame_length=2048, hop_length=512)
>>> for n, y in enumerate(stream):
...     n_sample = librosa.blocks_to_samples(n, block_length=16,
...                                          hop_length=512)
```

librosa.core.blocks_to_time

`librosa.core.blocks_to_time`(*blocks*, *block_length*, *hop_length*, *sr*)[source]

Convert block indices to time (in seconds)

Parameters: **blocks** : np.ndarray

Block index or array of block indices

block_length : int > 0

The number of frames per block

hop_length : int > 0

The number of samples to advance between frames

sr : int > 0

The sampling rate (samples per second)

Returns: **times** : np.ndarray [shape=samples.shape]

The time index or indices (in seconds) corresponding to the beginning of each provided block.

Note that these correspond to the time of the *first* sample in each block, and are not frame-centered.

See also

`blocks_to_frames`
`blocks_to_samples`

Examples

Get time indices for each block in a stream

```
>>> filename = librosa.util.example_audio_file()
>>> sr = librosa.get_samplerate(filename)
>>> stream = librosa.stream(filename, block_length=16,
...                         frame_length=2048, hop_length=512)
>>> for n, y in enumerate(stream):
...     n_time = librosa.blocks_to_time(n, block_length=16,
...                                     hop_length=512,
sr=sr)librosa.core.hz_to_note
```

`librosa.core.hz_to_note`(*frequencies*, ***kwargs*)[source]

Convert one or more frequencies (in Hz) to the nearest note names.

Parameters: **frequencies** : float or iterable of float

Input frequencies, specified in Hz

kwargs : additional keyword arguments

Arguments passed through to `midi_to_note`

Returns: **notes** : list of str

notes[i] is the closest note name to *frequency[i]* (or *frequency* if the input is scalar)

See also

`hz_to_midi`
`midi_to_note`
`note_to_hz`

Examples

Get a single note name for a frequency

```
>>> librosa.hz_to_note(440.0)
['A5']
```

Get multiple notes with cent deviation

```
>>> librosa.hz_to_note([32, 64], cents=True)
['C1-38', 'C2-38']
```

Get multiple notes, but suppress octave labels

```
>>> librosa.hz_to_note(440.0 * (2.0 ** np.linspace(0, 1, 12)),
...                    octave=False)
['A', 'A#', 'B', 'C', 'C#', 'D', 'E', 'F', 'F#', 'G', 'G#', 'A']
```

librosa.core.hz_to_midi

`librosa.core.hz_to_midi`(*frequencies*)[source]

Get MIDI note number(s) for given frequencies

Parameters: **frequencies** : float or np.ndarray [shape=(n,), dtype=float]

frequencies to convert

Returns: **note_nums** : number or np.ndarray [shape=(n,), dtype=float]

MIDI notes to *frequencies*

See also

`midi_to_hz`
`note_to_midi`
`hz_to_note`

Examples

```
>>> librosa.hz_to_midi(60)
34.506
>>> librosa.hz_to_midi([110, 220, 440])
array([ 45.,  57.,  69.])
```

librosa.core.midi_to_hz

`librosa.core.midi_to_hz`(*notes*)[source]

Get the frequency (Hz) of MIDI note(s)

Parameters: **notes** : int or np.ndarray [shape=(n,), dtype=int]

midi number(s) of the note(s)

Returns: **frequency** : number or np.ndarray [shape=(n,), dtype=float]

frequency (frequencies) of *notes* in Hz

See also

`hz_to_midi`
`note_to_hz`

Examples

```
>>> librosa.midi_to_hz(36)
65.406

>>> librosa.midi_to_hz(np.arange(36, 48))
array([  65.406,   69.296,   73.416,   77.782,   82.407,
         87.307,   92.499,   97.999,  103.826,  110.   ,
        116.541,  123.471])
```

librosa.core.midi_to_note

`librosa.core.midi_to_note`(*midi*, *octave=True*, *cents=False*)[source]

Convert one or more MIDI numbers to note strings.

MIDI numbers will be rounded to the nearest integer.

Notes will be of the format ‘C0’, ‘C#0’, ‘D0’, …

Parameters: **midi** : int or iterable of int

Midi numbers to convert.

octave: bool

If True, include the octave number

cents: bool

If true, cent markers will be appended for fractional notes. Eg, *midi_to_note(69.3, cents=True) == A4+03*

Returns: **notes** : str or iterable of str

Strings describing each midi note.

Raises: ParameterError

if *cents* is True and *octave* is False

See also

`midi_to_hz`
`note_to_midi`
`hz_to_note`

Examples

```
>>> librosa.midi_to_note(0)
'C-1'
>>> librosa.midi_to_note(37)
'C#2'
>>> librosa.midi_to_note(-2)
'A#-2'
```

```
>>> librosa.midi_to_note(104.7)
'A7'
>>> librosa.midi_to_note(104.7, cents=True)
'A7-30'
>>> librosa.midi_to_note(list(range(12, 24)))
['C0', 'C#0', 'D0', 'D#0', 'E0', 'F0', 'F#0', 'G0', 'G#0', 'A0', 'A#0',
'B0']
```

librosa.core.note_to_hz

`librosa.core.note_to_hz`(*note*, ***kwargs*)[source]

Convert one or more note names to frequency (Hz)

Parameters: **note** : str or iterable of str

One or more note names to convert

kwargs : additional keyword arguments

Additional parameters to `note_to_midi`

Returns: **frequencies** : number or np.ndarray [shape=(len(note),)]

Array of frequencies (in Hz) corresponding to *note*

See also

`midi_to_hz`
`note_to_midi`
`hz_to_note`

Examples

```
>>> # Get the frequency of a note
>>> librosa.note_to_hz('C')
array([ 16.352])
>>> # Or multiple notes
>>> librosa.note_to_hz(['A3', 'A4', 'A5'])
array([ 220.,  440.,  880.])
>>> # Or notes with tuning deviations
>>> librosa.note_to_hz('C2-32', round_midi=False)
array([ 64.209])
```

librosa.core.note_to_midi

`librosa.core.note_to_midi`(*note*, *round_midi=True*)[source]

Convert one or more spelled notes to MIDI number(s).

Notes may be spelled out with optional accidentals or octave numbers.

The leading note name is case-insensitive.

Sharps are indicated with #, flats may be indicated with `!` or `b`.

Parameters: **note** : str or iterable of str

One or more note names.

round_midi : bool

- If *True*, allow for fractional midi notes
- Otherwise, round cent deviations to the nearest note

Returns: **midi** : float or np.array

Midi note numbers corresponding to inputs.

Raises: ParameterError

If the input is not in valid note format

See also

`midi_to_note`
`note_to_hz`

Examples

```
>>> librosa.note_to_midi('C')
12
>>> librosa.note_to_midi('C#3')
49
>>> librosa.note_to_midi('f4')
65
>>> librosa.note_to_midi('Bb-1')
10
>>> librosa.note_to_midi('A!8')
116
>>> # Lists of notes also work
>>> librosa.note_to_midi(['C', 'E', 'G'])
array([12, 16, 19])
```

librosa.core.hz_to_mel

`librosa.core.hz_to_mel`(*frequencies*, *htk=False*)[source]

Convert Hz to Mels

Parameters: **frequencies** : number or np.ndarray [shape=(n,)] , float

scalar or array of frequencies

htk : bool

use HTK formula instead of Slaney

Returns: **mels** : number or np.ndarray [shape=(n,)]

input frequencies in Mels

See also

`mel_to_hz`

Examples

```
>>> librosa.hz_to_mel(60)
0.9
>>> librosa.hz_to_mel([110, 220, 440])
array([ 1.65,  3.3 ,  6.6 ])
```

librosa.core.hz_to_octs

`librosa.core.hz_to_octs`(*frequencies*, *tuning=0.0*, *bins_per_octave=12*, *A440=<DEPRECATED parameter>*)[source]

Convert frequencies (Hz) to (fractional) octave numbers.

Parameters: **frequencies** : number >0 or np.ndarray [shape=(n,)] or float

scalar or vector of frequencies

tuning : float

Tuning deviation from A440 in (fractional) bins per octave.

bins_per_octave : int > 0

Number of bins per octave.

A440 : float <DEPRECATED>

frequency of A440 (in Hz)

Note

This parameter is deprecated in 0.7.1 and will be removed in version 0.8.0. Use *tuning=* instead.

Returns: **octaves** : number or np.ndarray [shape=(n,)]

octave number for each frequency

See also

octs_to_hz

Examples

```
>>> librosa.hz_to_octs(440.0)
4.
>>> librosa.hz_to_octs([32, 64, 128, 256])
array([ 0.219,  1.219,  2.219,  3.219])
```

librosa.core.mel_to_hz

librosa.core.mel_to_hz(*mels*, *htk=False*)[source]

Convert mel bin numbers to frequencies

Parameters: **mels** : np.ndarray [shape=(n,)], float

mel bins to convert

htk : bool

use HTK formula instead of Slaney

Returns: **frequencies** : np.ndarray [shape=(n,)]

input mels in Hz

See also

hz_to_mel

Examples

```
>>> librosa.mel_to_hz(3)
200.

>>> librosa.mel_to_hz([1,2,3,4,5])
array([  66.667,  133.333,  200.   ,  266.667,  333.333])
```

librosa.core.octs_to_hz

librosa.core.octs_to_hz(*octs*, *tuning=0.0*, *bins_per_octave=12*, *A440=<DEPRECATED parameter>*)[source]

Convert octaves numbers to frequencies.

Octaves are counted relative to A.

Parameters: **octaves** : np.ndarray [shape=(n,)] or float

octave number for each frequency

tuning : float

Tuning deviation from A440 in (fractional) bins per octave.

bins_per_octave : int > 0

Number of bins per octave.

A440 : float <DEPRECATED>

frequency of A440

Returns: **frequencies** : number or np.ndarray [shape=(n,)]

scalar or vector of frequencies

See also

`hz_to_octs`

Examples

```
>>> librosa.octs_to_hz(1)
55.
>>> librosa.octs_to_hz([-2, -1, 0, 1, 2])
array([   6.875,   13.75 ,   27.5  ,   55.   ,  110.   ])
```

librosa.core.fft_frequencies

`librosa.core.fft_frequencies`(*sr=22050, n_fft=2048*)[source]

Alternative implementation of *np.fft.fftfreq*

Parameters: **sr** : number > 0 [scalar]

Audio sampling rate

n_fft : int > 0 [scalar]

FFT window size

Returns: **freqs** : np.ndarray [shape=(1 + n_fft/2,)]

Frequencies *(0, sr/n_fft, 2*sr/n_fft, …, sr/2)*

Examples

```
>>> librosa.fft_frequencies(sr=22050, n_fft=16)
```

```
array([     0.   ,   1378.125,   2756.25 ,   4134.375,
        5512.5  ,   6890.625,   8268.75 ,   9646.875,  11025.   ])
```

librosa.core.cqt_frequencies

`librosa.core.cqt_frequencies`(*n_bins, fmin, bins_per_octave=12, tuning=0.0*)[source]

Compute the center frequencies of Constant-Q bins.

Parameters: **n_bins** : int > 0 [scalar]

Number of constant-Q bins

fmin : float > 0 [scalar]

Minimum frequency

bins_per_octave : int > 0 [scalar]

Number of bins per octave

tuning : float

Deviation from A440 tuning in fractional bins

Returns: **frequencies** : np.ndarray [shape=(n_bins,)]

Center frequency for each CQT bin

Examples

```
>>> # Get the CQT frequencies for 24 notes, starting at C2
>>> librosa.cqt_frequencies(24, fmin=librosa.note_to_hz('C2'))
array([  65.406,   69.296,   73.416,   77.782,   82.407,   87.307,
         92.499,   97.999,  103.826,  110.   ,  116.541,  123.471,
        130.813,  138.591,  146.832,  155.563,  164.814,  174.614,
        184.997,  195.998,  207.652,  220.   ,  233.082,  246.942])
```

librosa.core.mel_frequencies

`librosa.core.mel_frequencies`(*n_mels=128, fmin=0.0, fmax=11025.0, htk=False*)
[source]

Compute an array of acoustic frequencies tuned to the mel scale.

The mel scale is a quasi-logarithmic function of acoustic frequency designed such that perceptually similar pitch intervals (e.g. octaves) appear equal in width over the full hearing range.

Because the definition of the mel scale is conditioned by a finite number of subjective psychoaoustical experiments, several implementations coexist in the audio signal processing literature [1]. By default, librosa replicates the behavior of the well-established MATLAB Auditory Toolbox of Slaney [2]. According to this default implementation, the conversion from Hertz to mel is linear below 1 kHz and logarithmic above 1 kHz. Another available implementation replicates the Hidden Markov Toolkit [3] (HTK) according to the following formula:

*mel = 2595.0 * np.log10(1.0 + f / 700.0).*

The choice of implementation is determined by the *htk* keyword argument: setting *htk=False* leads to the Auditory toolbox implementation, whereas setting it *htk=True* leads to the HTK implementation.

[1] Umesh, S., Cohen, L., & Nelson, D. Fitting the mel scale. In Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 1, pp. 217-220, 1998.

[2] Slaney, M. Auditory Toolbox: A MATLAB Toolbox for Auditory Modeling Work. Technical Report, version 2, Interval Research Corporation, 1998.

[3] Young, S., Evermann, G., Gales, M., Hain, T., Kershaw, D., Liu, X., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V., & Woodland, P. The HTK book, version 3.4. Cambridge University, March 2009.

Parameters: **n_mels** : int > 0 [scalar]

Number of mel bins.

fmin : float >= 0 [scalar]

Minimum frequency (Hz).

fmax : float >= 0 [scalar]

Maximum frequency (Hz).

htk : bool

If True, use HTK formula to convert Hz to mel. Otherwise (False), use Slaney's Auditory Toolbox.

Returns: **bin_frequencies** : ndarray [shape=(n_mels,)]

Vector of n_mels frequencies in Hz which are uniformly spaced on the Mel axis.

See also

`hz_to_mel`
`mel_to_hz`
`librosa.feature.melspectrogram`
`librosa.feature.mfcc`

Examples

```
>>> librosa.mel_frequencies(n_mels=40)
array([     0.   ,     85.317,    170.635,    255.952,
          341.269,    426.586,    511.904,    597.221,
          682.538,    767.855,    853.173,    938.49 ,
         1024.856,   1119.114,   1222.042,   1334.436,
         1457.167,   1591.187,   1737.532,   1897.337,
         2071.84 ,   2262.393,   2470.47 ,   2697.686,
         2945.799,   3216.731,   3512.582,   3835.643,
         4188.417,   4573.636,   4994.285,   5453.621,
         5955.205,   6502.92 ,   7101.009,   7754.107,
         8467.272,   9246.028,  10096.408,  11025.   ])
```

librosa.core.tempo_frequencies

`librosa.core.tempo_frequencies`(*n_bins*, *hop_length=512*, *sr=22050*)[source]

Compute the frequencies (in beats per minute) corresponding to an onset auto-correlation or tempogram matrix.

Parameters:

n_bins : int > 0

The number of lag bins

hop_length : int > 0

The number of samples between each bin

sr : number > 0

The audio sampling rate

Returns:

bin_frequencies : ndarray [shape=(n_bins,)]

vector of bin frequencies measured in BPM.

Note

bin_frequencies[0] = +np.inf corresponds to 0-lag

Examples

Get the tempo frequencies corresponding to a 384-bin (8-second) tempogram

```
>>> librosa.tempo_frequencies(384)
array([      inf,  2583.984,  1291.992, ...,     6.782,
           6.764,     6.747])
```

librosa.core.fourier_tempo_frequencies

`librosa.core.fourier_tempo_frequencies`(*sr=22050*, *win_length=384*, *hop_length=512*)[source]

Compute the frequencies (in beats per minute) corresponding to a Fourier tempogram matrix.

Parameters: **sr** : number > 0

The audio sampling rate

win_length : int > 0

The number of frames per analysis window

hop_length : int > 0

The number of samples between each bin

Returns: **bin_frequencies** : ndarray [shape=(win_length // 2 + 1 ,)]

vector of bin frequencies measured in BPM.

Examples

Get the tempo frequencies corresponding to a 384-bin (8-second) tempogram

```
>>> librosa.fourier_tempo_frequencies(384)
array([ 0.   ,  0.117,  0.234, ..., 22.266, 22.383, 22.5  ])
```

librosa.core.samples_like

`librosa.core.samples_like`(*X*, *hop_length=512*, *n_fft=None*, *axis=-1*)[source]

Return an array of sample indices to match the time axis from a feature matrix.

Parameters: **X** : np.ndarray or scalar

- If ndarray, X is a feature matrix, e.g. STFT, chromagram, or mel spectrogram.
- If scalar, X represents the number of frames.

hop_length : int > 0 [scalar]

number of samples between successive frames

n_fft : None or int > 0 [scalar]

Optional: length of the FFT window. If given, time conversion will include an offset of *n_fft / 2* to counteract windowing effects when using a non-centered STFT.

axis : int [scalar]

The axis representing the time axis of X. By default, the last axis (-1) is taken.

Returns: **samples** : np.ndarray [shape=(n,)]

ndarray of sample indices corresponding to each frame of X.

See also

`times_like`
Return an array of time values to match the time axis from a feature matrix.

Examples

Provide a feature matrix input:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> X = librosa.stft(y)
>>> samples = librosa.samples_like(X)
>>> samples
array([      0,     512,    1024, ..., 1353728, 1354240, 1354752])
```

Provide a scalar input:

```
>>> n_frames = 2647
>>> samples = librosa.samples_like(n_frames)
>>> samples
array([      0,     512,    1024, ..., 1353728, 1354240, 1354752])
```

librosa.core.times_like

`librosa.core.times_like`(*X*, *sr=22050*, *hop_length=512*, *n_fft=None*, *axis=-1*)[source]

Return an array of time values to match the time axis from a feature matrix.

Parameters: **X** : np.ndarray or scalar

- If ndarray, X is a feature matrix, e.g. STFT, chromagram, or mel spectrogram.
- If scalar, X represents the number of frames.

sr : number > 0 [scalar]

audio sampling rate

hop_length : int > 0 [scalar]

number of samples between successive frames

n_fft : None or int > 0 [scalar]

Optional: length of the FFT window. If given, time conversion will include an offset of *n_fft / 2* to counteract windowing effects when using

a non-centered STFT.

axis : int [scalar]

The axis representing the time axis of X. By default, the last axis (-1) is taken.

Returns: **times** : np.ndarray [shape=(n,)]

ndarray of times (in seconds) corresponding to each frame of X.

See also

`samples_like`
Return an array of sample indices to match the time axis from a feature matrix.

Examples

Provide a feature matrix input:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> X = librosa.stft(y)
>>> times = librosa.times_like(X)
>>> times
array([  0.00000000e+00,   2.32199546e-02,   4.64399093e-02, ...,
         6.13935601e+01,   6.14167800e+01,   6.14400000e+01])
```

Provide a scalar input:

```
>>> n_frames = 2647
>>> times = librosa.times_like(n_frames)
>>> times
array([  0.00000000e+00,   2.32199546e-02,   4.64399093e-02, ...,
         6.13935601e+01,   6.14167800e+01,   6.14400000e+01])
```

librosa.core.estimate_tuning

`librosa.core.estimate_tuning`(*y=None, sr=22050, S=None, n_fft=2048, resolution=0.01, bins_per_octave=12, **kwargs*)[source]

Estimate the tuning of an audio time series or spectrogram input.

Parameters: **y: np.ndarray [shape=(n,)] or None**

audio signal

sr : number > 0 [scalar]

audio sampling rate of *y*

S: np.ndarray [shape=(d, t)] or None

magnitude or power spectrogram

n_fft : int > 0 [scalar] or None

number of FFT bins to use, if *y* is provided.

resolution : float in *(0, 1)*

Resolution of the tuning as a fraction of a bin. 0.01 corresponds to measurements in cents.

bins_per_octave : int > 0 [scalar]

How many frequency bins per octave

kwargs : additional keyword arguments

Additional arguments passed to `piptrack`

Returns: tuning: float in *[-0.5, 0.5)*

estimated tuning deviation (fractions of a bin)

See also

`piptrack`
Pitch tracking by parabolic interpolation

Examples

```
>>> # With time-series input
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.estimate_tuning(y=y, sr=sr)
0.089999999999999969

>>> # In tenths of a cent
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.estimate_tuning(y=y, sr=sr, resolution=1e-3)
0.093999999999999972

>>> # Using spectrogram input
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
>>> librosa.estimate_tuning(S=S, sr=sr)
0.089999999999999969

>>> # Using pass-through arguments to `librosa.piptrack`
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.estimate_tuning(y=y, sr=sr, n_fft=8192,
...                         fmax=librosa.note_to_hz('G#9'))
0.070000000000000062
```

librosa.core.pitch_tuning

librosa.core.pitch_tuning(*frequencies, resolution=0.01, bins_per_octave=12*)[source]

Given a collection of pitches, estimate its tuning offset (in fractions of a bin) relative to A440=440.0Hz.

Parameters: **frequencies** : array-like, float

A collection of frequencies detected in the signal. See piptrack

resolution : float in *(0, 1)*

Resolution of the tuning as a fraction of a bin. 0.01 corresponds to cents.

bins_per_octave : int > 0 [scalar]

How many frequency bins per octave

Returns: tuning: float in *[-0.5, 0.5)*

estimated tuning deviation (fractions of a bin)

See also

estimate_tuning

Estimating tuning from time-series or spectrogram input

Examples

```
>>> # Generate notes at +25 cents
>>> freqs = librosa.cqt_frequencies(24, 55, tuning=0.25)
>>> librosa.pitch_tuning(freqs)
0.25

>>> # Track frequencies from a real spectrogram
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> pitches, magnitudes, stft = librosa.ifptrack(y, sr)
>>> # Select out pitches with high energy
>>> pitches = pitches[magnitudes > np.median(magnitudes)]
>>> librosa.pitch_tuning(pitches)
0.089999999999999969
```

librosa.core.piptrack

librosa.core.piptrack(*y=None, sr=22050, S=None, n_fft=2048, hop_length=None, fmin=150.0, fmax=4000.0, threshold=0.1, win_length=None, window='hann', center=True, pad_mode='reflect', ref=None*)[source]

Pitch tracking on thresholded parabolically-interpolated STFT.

This implementation uses the parabolic interpolation method described by [1].

[1] https://ccrma.stanford.edu/~jos/sasp/Sinusoidal_Peak_Interpolation.html

Parameters: y: np.ndarray [shape=(n,)] or None

audio signal

sr : number > 0 [scalar]

audio sampling rate of *y*

S: np.ndarray [shape=(d, t)] or None

magnitude or power spectrogram

n_fft : int > 0 [scalar] or None

number of FFT bins to use, if *y* is provided.

hop_length : int > 0 [scalar] or None

number of samples to hop

threshold : float in *(0, 1)*

A bin in spectrum *S* is considered a pitch when it is greater than *threshold*ref(S)*.

By default, *ref(S)* is taken to be *max(S, axis=0)* (the maximum value in each column).

fmin : float > 0 [scalar]

lower frequency cutoff.

fmax : float > 0 [scalar]

upper frequency cutoff.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t **

hop_length].

- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

ref : scalar or callable [default=np.max]

If scalar, the reference value against which *S* is compared for determining pitches.

If callable, the reference value is computed as *ref(S, axis=0)*.

.. note::

One of *S* or *y* must be provided.

If *S* is not given, it is computed from *y* using the default parameters of `librosa.core.stft`.

Returns: **pitches** : np.ndarray [shape=(d, t)]
magnitudes : np.ndarray [shape=(d,t)]

Where *d* is the subset of FFT bins within *fmin* and *fmax*.

pitches[f, t] contains instantaneous frequency at bin *f*, time *t*

magnitudes[f, t] contains the corresponding magnitudes.

Both *pitches* and *magnitudes* take value 0 at bins of non-maximal magnitude.

Notes

This function caches at level 30.

Examples

Computing pitches from a waveform input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> pitches, magnitudes = librosa.piptrack(y=y, sr=sr)
```

Or from a spectrogram input

```
>>> S = np.abs(librosa.stft(y))
>>> pitches, magnitudes = librosa.piptrack(S=S, sr=sr)
```

Or with an alternate reference value for pitch detection, where values above the mean spectral energy in each frame are counted as pitches

```
>>> pitches, magnitudes = librosa.piptrack(S=S, sr=sr, threshold=1,
```

```
...                                                    ref=np.mean)
```

librosa.core.ifgram

`librosa.core.ifgram`(*y*, *sr=22050*, *n_fft=2048*, *hop_length=None*, *win_length=None*, *window='hann'*, *norm=False*, *center=True*, *ref_power=1e-06*, *clip=True*, *dtype=<class 'numpy.complex64'>*, *pad_mode='reflect'*)[source]

Compute the instantaneous frequency (as a proportion of the sampling rate) obtained as the time-derivative of the phase of the complex spectrum as described by [1].

Calculates regular STFT as a side effect.

[1] Abe, Toshihiko, Takao Kobayashi, and Satoshi Imai. “Harmonics tracking and pitch extraction based on instantaneous frequency.” International Conference on Acoustics, Speech, and Signal Processing, ICASSP-95., Vol. 1. IEEE, 1995.

Warning

This function is deprecated in version 0.7.1, and will be removed in version 0.8.0. The function `reassigned_spectrogram` provides comparable functionality, and should be used instead of `ifgram`.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length, number samples between subsequent frames. If not supplied, defaults to *win_length / 4*.

win_length : int > 0, <= n_fft

Window length. Defaults to *n_fft*. See `stft` for details.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a user-specified window vector of length *n_fft*

See `stft` for details.

norm : bool

Normalize the STFT.

center : boolean

- If *True*, the signal *y* is padded so that frame *D[:, t]* (and *if_gram*) is centered at *y[t * hop_length]*.
- If *False*, then *D[:, t]* at *y[t * hop_length]*

ref_power : float >= 0 or callable

Minimum power threshold for estimating instantaneous frequency. Any bin with *np.abs(D[f, t])**2 < ref_power* will receive the default frequency estimate.

If callable, the threshold is set to *ref_power(np.abs(D)**2)*.

clip : boolean

- If *True*, clip estimated frequencies to the range *[0, 0.5 * sr]*.
- If *False*, estimated frequencies can be negative or exceed *0.5 * sr*.

dtype : numeric type

Complex numeric type for *D*. Default is 64-bit complex.

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

Returns: **if_gram** : np.ndarray [shape=(1 + n_fft/2, t), dtype=real]

Instantaneous frequency spectrogram: *if_gram[f, t]* is the frequency at bin *f*, time *t*

D : np.ndarray [shape=(1 + n_fft/2, t), dtype=complex]

Short-time Fourier transform

See also

`stft`
Short-time Fourier Transform

`reassigned_spectrogram`
Time-frequency reassigned spectrogram

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> frequencies, D = librosa.ifgram(y, sr=sr)
>>> frequencies
array([[  0.000e+00,   0.000e+00, ...,   0.000e+00,   0.000e+00],
       [  3.150e+01,   3.070e+01, ...,   1.077e+01,   1.077e+01],
```

```
...,
[  1.101e+04,   1.101e+04, ...,   1.101e+04,   1.101e+04],
[  1.102e+04,   1.102e+04, ...,   1.102e+04,   1.102e+04]])
```

Display

specshow(data[, x_coords, y_coords, x_axis, …])	Display a spectrogram/chromagram/cqt/etc.
waveplot(y[, sr, max_points, x_axis, …])	Plot the amplitude envelope of a waveform.
cmap(data[, robust, cmap_seq, cmap_bool, …])	Get a default colormap from the given data.
TimeFormatter([lag, unit])	A tick formatter for time axes.
NoteFormatter([octave, major])	Ticker formatter for Notes
LogHzFormatter([major])	Ticker formatter for logarithmic frequency
ChromaFormatter	A formatter for chroma axes
TonnetzFormatter	A formatter for tonnetz axes

librosa.display.specshow

librosa.display.specshow(*data*, *x_coords=None*, *y_coords=None*, *x_axis=None*, *y_axis=None*, *sr=22050*, *hop_length=512*, *fmin=None*, *fmax=None*, *tuning=0.0*, *bins_per_octave=12*, *ax=None*, ***kwargs*)[source]

Display a spectrogram/chromagram/cqt/etc.

Parameters:

data : np.ndarray [shape=(d, n)]

Matrix to display (e.g., spectrogram)

sr : number > 0 [scalar]

Sample rate used to determine time scale in x-axis.

hop_length : int > 0 [scalar]

Hop length, also used to determine time scale in x-axis

x_axis : None or str
y_axis : None or str

Range for the x- and y-axes.

Valid types are:

- None, ‘none’, or ‘off’ : no axis decoration is displayed.

Frequency types:

- ‘linear’, ‘fft’, ‘hz’ : frequency range is determined by the FFT window and sampling rate.
- ‘log’ : the spectrum is displayed on a log scale.
- ‘mel’ : frequencies are determined by the mel scale.
- ‘cqt_hz’ : frequencies are determined by the CQT scale.
- ‘cqt_note’ : pitches are determined by the CQT scale.

All frequency types are plotted in units of Hz.

Categorical types:

- ‘chroma’ : pitches are determined by the chroma filters. Pitch classes are arranged at integer locations (0-11).
- ‘tonnetz’ : axes are labeled by Tonnetz dimensions (0-5)
- ‘frames’ : markers are shown as frame counts.

Time types:

- ‘time’ : markers are shown as milliseconds, seconds, minutes, or hours.
  Values are plotted in units of seconds.
- ‘s’ : markers are shown as seconds.
- ‘ms’ : markers are shown as milliseconds.
- ‘lag’ : like time, but past the halfway point counts as negative values.
- ‘lag_s’ : same as lag, but in seconds.
- ‘lag_ms’ : same as lag, but in milliseconds.

Rhythm:

- ‘tempo’ : markers are shown as beats-per-minute (BPM)

  using a logarithmic scale. This is useful for visualizing the outputs of *feature.tempogram*.

- ‘fourier_tempo’ : same as *‘tempo’*, but used when

  tempograms are calculated in the Frequency domain using *feature.fourier_tempogram*.

x_coords : np.ndarray [shape=data.shape[1]+1]
y_coords : np.ndarray [shape=data.shape[0]+1]

Optional positioning coordinates of the input data. These can be use to explicitly set the location of each element *data[i, j]*, e.g., for displaying

beat-synchronous features in natural time coordinates.

If not provided, they are inferred from *x_axis* and *y_axis*.

fmin : float > 0 [scalar] or None

Frequency of the lowest spectrogram bin. Used for Mel and CQT scales.

If *y_axis* is *cqt_hz* or *cqt_note* and *fmin* is not given, it is set by default to *note_to_hz('C1')*.

fmax : float > 0 [scalar] or None

Used for setting the Mel frequency scales

tuning : float

Tuning deviation from A440, in fractions of a bin.

This is used for CQT frequency scales, so that *fmin* is adjusted to *fmin * 2**(tuning / bins_per_octave)*.

bins_per_octave : int > 0 [scalar]

Number of bins per octave. Used for CQT frequency scale.

ax : matplotlib.axes.Axes or None

Axes to plot on instead of the default *plt.gca()*.

kwargs : additional keyword arguments

Arguments passed through to `matplotlib.pyplot.pcolormesh`.

By default, the following options are set:

- *rasterized=True*
- *shading='flat'*
- *edgecolors='None'*

Returns: axes

The axis handle for the figure.

See also

`cmap`
Automatic colormap detection
`matplotlib.pyplot.pcolormesh`

Examples

Visualize an STFT power spectrum

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> plt.figure(figsize=(12, 8))

>>> D = librosa.amplitude_to_db(np.abs(librosa.stft(y)), ref=np.max)
>>> plt.subplot(4, 2, 1)
>>> librosa.display.specshow(D, y_axis='linear')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Linear-frequency power spectrogram')
```

Or on a logarithmic scale

```
>>> plt.subplot(4, 2, 2)
>>> librosa.display.specshow(D, y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Log-frequency power spectrogram')
```

Or use a CQT scale

```
>>> CQT = librosa.amplitude_to_db(np.abs(librosa.cqt(y, sr=sr)),
ref=np.max)
>>> plt.subplot(4, 2, 3)
>>> librosa.display.specshow(CQT, y_axis='cqt_note')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Constant-Q power spectrogram (note)')

>>> plt.subplot(4, 2, 4)
>>> librosa.display.specshow(CQT, y_axis='cqt_hz')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Constant-Q power spectrogram (Hz)')
```

Draw a chromagram with pitch classes

```
>>> C = librosa.feature.chroma_cqt(y=y, sr=sr)
>>> plt.subplot(4, 2, 5)
>>> librosa.display.specshow(C, y_axis='chroma')
>>> plt.colorbar()
>>> plt.title('Chromagram')
```

Force a grayscale colormap (white -> black)

```
>>> plt.subplot(4, 2, 6)
>>> librosa.display.specshow(D, cmap='gray_r', y_axis='linear')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Linear power spectrogram (grayscale)')
```

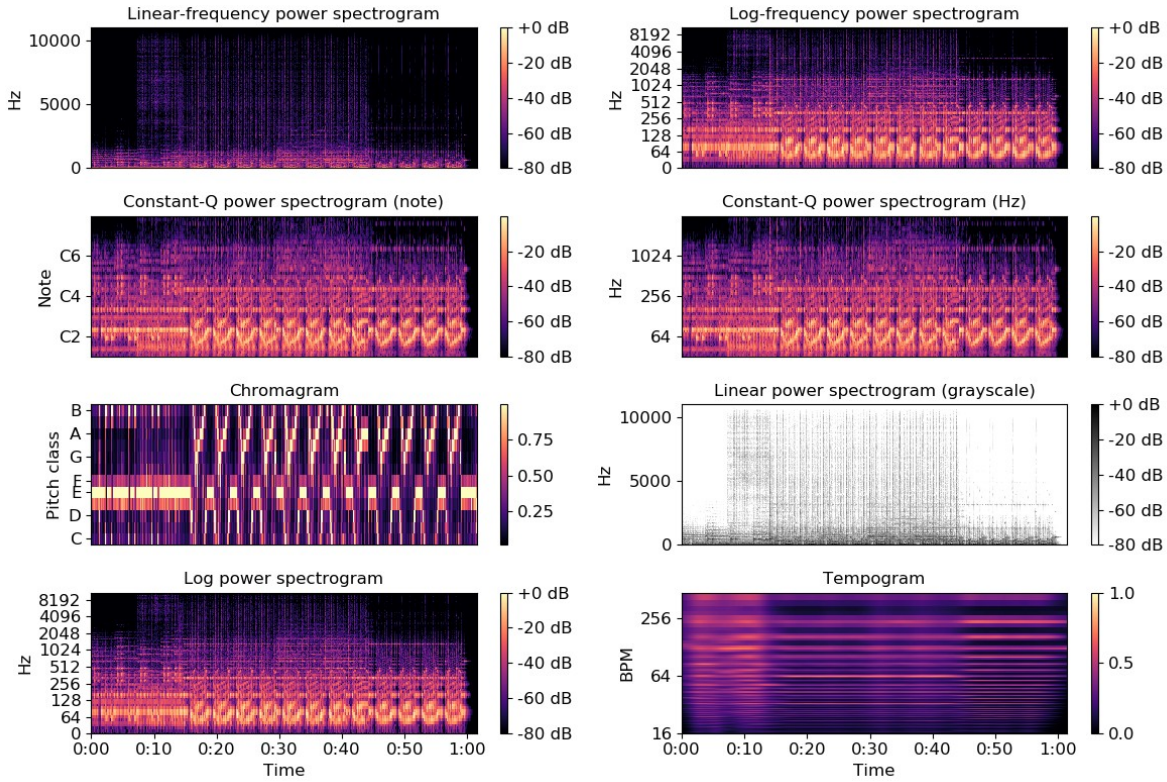
Draw time markers automatically

```
>>> plt.subplot(4, 2, 7)
>>> librosa.display.specshow(D, x_axis='time', y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Log power spectrogram')
```

Draw a tempogram with BPM markers

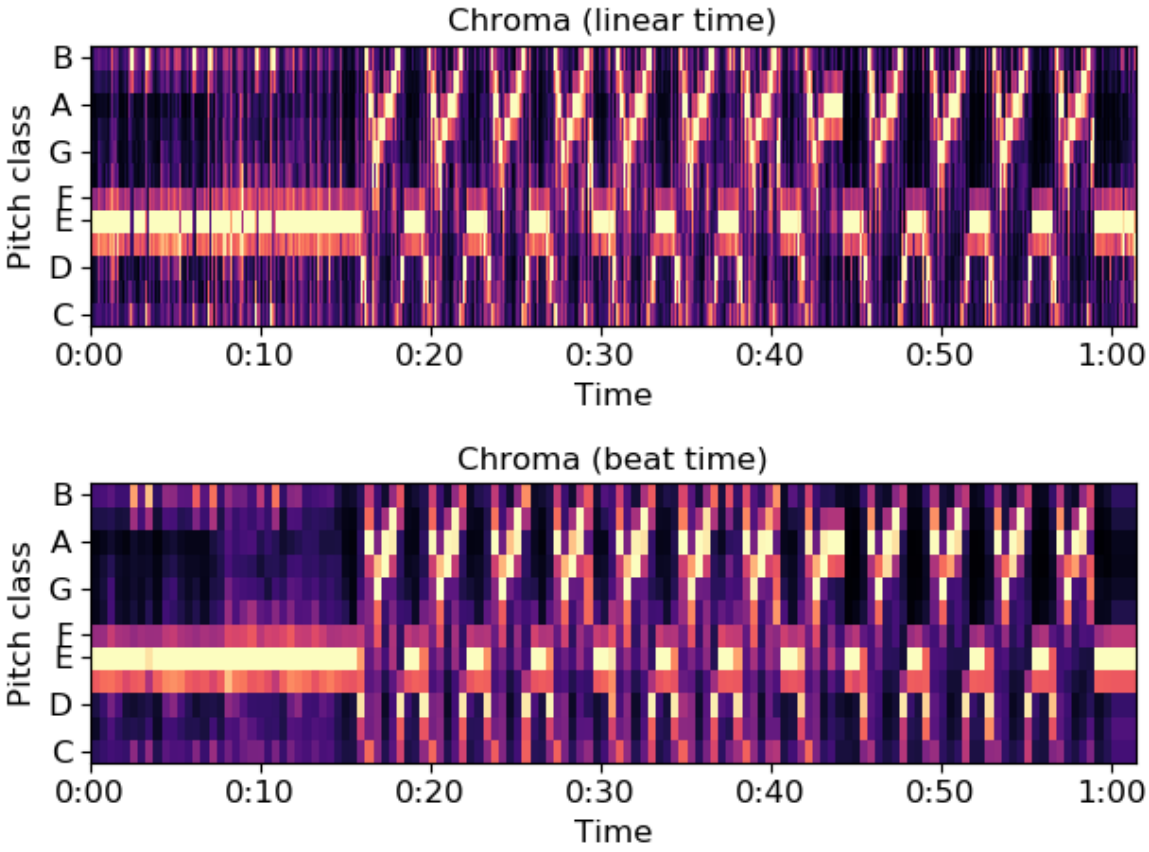
```
>>> plt.subplot(4, 2, 8)
```

```
>>> Tgram = librosa.feature.tempogram(y=y, sr=sr)
>>> librosa.display.specshow(Tgram, x_axis='time', y_axis='tempo')
>>> plt.colorbar()
>>> plt.title('Tempogram')
>>> plt.tight_layout()
>>> plt.show()
```



Draw beat-synchronous chroma in natural time

```
>>> plt.figure()
>>> tempo, beat_f = librosa.beat.beat_track(y=y, sr=sr, trim=False)
>>> beat_f = librosa.util.fix_frames(beat_f, x_max=C.shape[1])
>>> Csync = librosa.util.sync(C, beat_f, aggregate=np.median)
>>> beat_t = librosa.frames_to_time(beat_f, sr=sr)
>>> ax1 = plt.subplot(2,1,1)
>>> librosa.display.specshow(C, y_axis='chroma', x_axis='time')
>>> plt.title('Chroma (linear time)')
>>> ax2 = plt.subplot(2,1,2, sharex=ax1)
>>> librosa.display.specshow(Csync, y_axis='chroma', x_axis='time',
...                          x_coords=beat_t)
>>> plt.title('Chroma (beat time)')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.display.waveplot

`librosa.display.waveplot`(*y, sr=22050, max_points=50000.0, x_axis='time', offset=0.0, max_sr=1000, ax=None, **kwargs*)[source]

Plot the amplitude envelope of a waveform.

If *y* is monophonic, a filled curve is drawn between *[-abs(y), abs(y)]*.

If *y* is stereo, the curve is drawn between *[-abs(y[1]), abs(y[0])]*, so that the left and right channels are drawn above and below the axis, respectively.

Long signals (*duration >= max_points*) are down-sampled to at most *max_sr* before plotting.

Parameters: **y** : np.ndarray [shape=(n,) or (2,n)]

audio time series (mono or stereo)

sr : number > 0 [scalar]

sampling rate of *y*

max_points : postive number or None

Maximum number of time-points to plot: if *max_points* exceeds the

duration of *y*, then *y* is downsampled.

If *None*, no downsampling is performed.

x_axis : str or None

Display of the x-axis ticks and tick markers. Accepted values are:

- ‘time’ : markers are shown as milliseconds, seconds, minutes, or hours.
  Values are plotted in units of seconds.
- ‘s’ : markers are shown as seconds.
- ‘ms’ : markers are shown as milliseconds.
- ‘lag’ : like time, but past the halfway point counts as negative values.
- ‘lag_s’ : same as lag, but in seconds.
- ‘lag_ms’ : same as lag, but in milliseconds.
- *None*, ‘none’, or ‘off’: ticks and tick markers are hidden.

ax : matplotlib.axes.Axes or None

Axes to plot on instead of the default *plt.gca()*.

offset : float

Horizontal offset (in seconds) to start the waveform plot

max_sr : number > 0 [scalar]

Maximum sampling rate for the visualization

kwargs

Additional keyword arguments to matplotlib.pyplot.fill_between

Returns: **pc** : matplotlib.collections.PolyCollection

The PolyCollection created by *fill_between*.

See also

librosa.core.resample
matplotlib.pyplot.fill_between

Examples

Plot a monophonic waveform

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=10)
>>> plt.figure()
>>> plt.subplot(3, 1, 1)
```

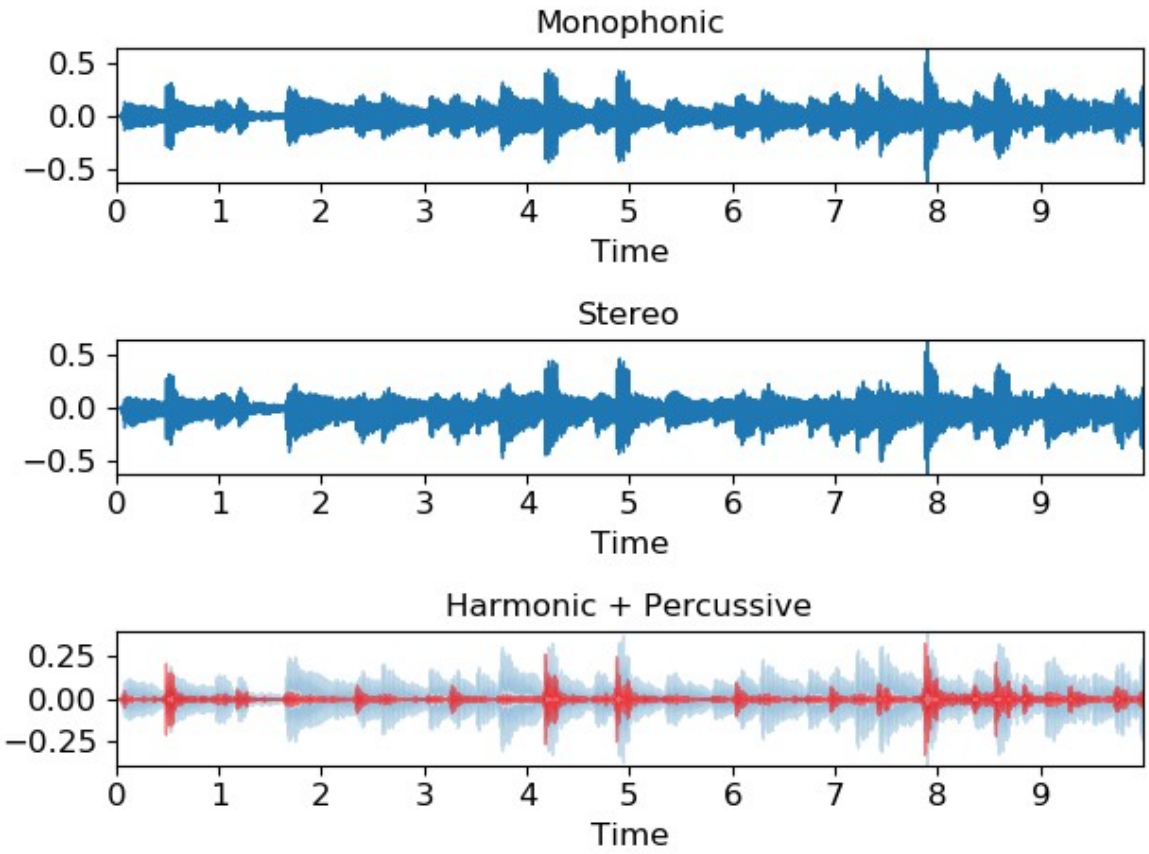
```
>>> librosa.display.waveplot(y, sr=sr)
>>> plt.title('Monophonic')
```

Or a stereo waveform

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      mono=False, duration=10)
>>> plt.subplot(3, 1, 2)
>>> librosa.display.waveplot(y, sr=sr)
>>> plt.title('Stereo')
```

Or harmonic and percussive components with transparency

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=10)
>>> y_harm, y_perc = librosa.effects.hpss(y)
>>> plt.subplot(3, 1, 3)
>>> librosa.display.waveplot(y_harm, sr=sr, alpha=0.25)
>>> librosa.display.waveplot(y_perc, sr=sr, color='r', alpha=0.5)
>>> plt.title('Harmonic + Percussive')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.display.cmap

`librosa.display.cmap`(*data, robust=True, cmap_seq='magma', cmap_bool='gray_r', cmap_div='coolwarm'*)[source]

Get a default colormap from the given data.

If the data is boolean, use a black and white colormap.

If the data has both positive and negative values, use a diverging colormap.

Otherwise, use a sequential colormap.

Parameters:

data : np.ndarray

Input data

robust : bool

If True, discard the top and bottom 2% of data when calculating range.

cmap_seq : str

The sequential colormap name

cmap_bool : str

The boolean colormap name

cmap_div : str

The diverging colormap name

Returns:

cmap : matplotlib.colors.Colormap

The colormap to use for *data*

See also

`matplotlib.pyplot.colormaps`

librosa.display.TimeFormatter

class `librosa.display.TimeFormatter`(*lag=False*, *unit=None*)[source]

A tick formatter for time axes.

Automatically switches between seconds, minutes:seconds, or hours:minutes:seconds.

Parameters:

lag : bool

If *True*, then the time axis is interpreted in lag coordinates. Anything past the midpoint will be converted to negative time.

unit : str or None

Abbreviation of the physical unit for axis labels and ticks. Either equal to *s* (seconds) or *ms* (milliseconds) or None (default). If set to None, the resulting TimeFormatter object adapts its string representation to the duration of the underlying time range: *hh:mm:ss* above 3600 seconds; *mm:ss* between 60 and 3600 seconds; and *ss* below 60 seconds.

See also

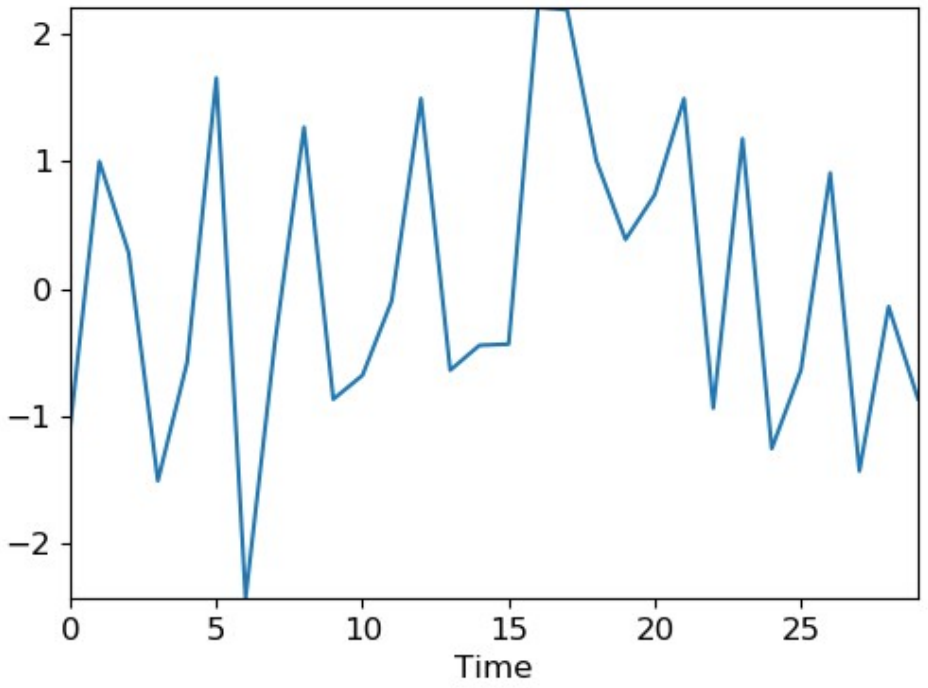
`matplotlib.ticker.Formatter`

Examples

For normal time

```
>>> import matplotlib.pyplot as plt
>>> times = np.arange(30)
>>> values = np.random.randn(len(times))
>>> plt.figure()
>>> ax = plt.gca()
>>> ax.plot(times, values)
>>> ax.xaxis.set_major_formatter(librosa.display.TimeFormatter())
>>> ax.set_xlabel('Time')
>>> plt.show()
```

(Source code)



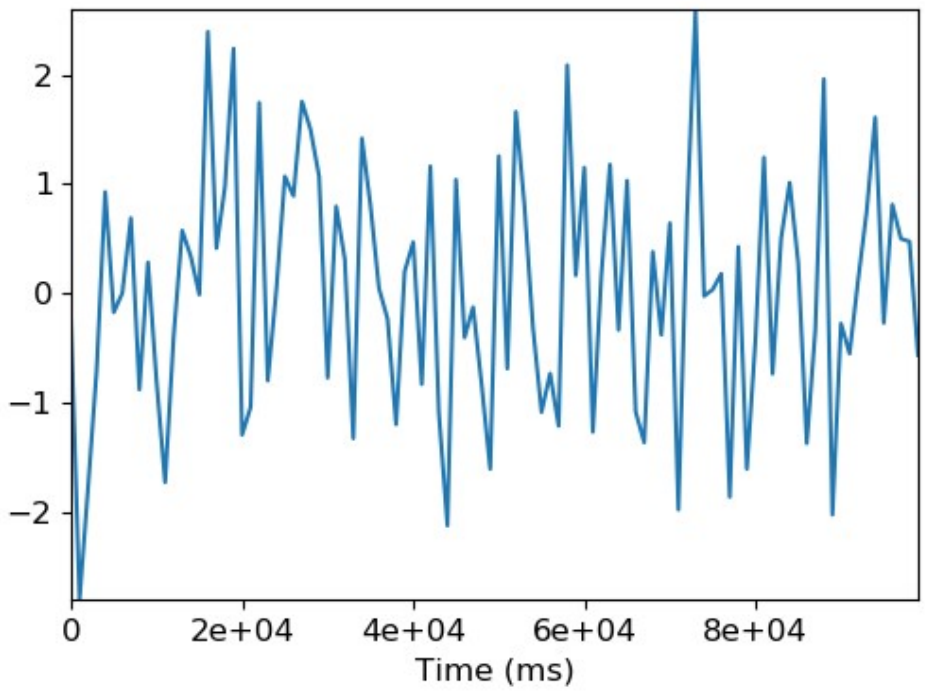
Manually set the physical time unit of the x-axis to milliseconds

```
>>> times = np.arange(100)
>>> values = np.random.randn(len(times))
```

```
>>> plt.figure()
>>> ax = plt.gca()
>>> ax.plot(times, values)
>>> ax.xaxis.set_major_formatter(librosa.display.TimeFormatter(unit='ms'))
>>> ax.set_xlabel('Time (ms)')
```

For lag plots

```
>>> times = np.arange(60)
>>> values = np.random.randn(len(times))
>>> plt.figure()
>>> ax = plt.gca()
>>> ax.plot(times, values)
>>> ax.xaxis.set_major_formatter(librosa.display.TimeFormatter(lag=True))
>>> ax.set_xlabel('Lag')
```



__init__(*self*, *lag=False*, *unit=None*)[source]

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, lag, unit])	Initialize self.
create_dummy_axis(self, **kwargs)	
fix_minus(self, s)	Some classes may want to

	replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness.
`format_data`(self, value)	Returns the full string representation of the value with the position unspecified.
`format_data_short`(self, value)	Return a short string version of the tick value.
`format_ticks`(self, values)	Return the tick labels for all the ticks at once.
`get_offset`(self)	
`set_axis`(self, axis)	
`set_bounds`(self, vmin, vmax)	
`set_data_interval`(self, vmin, vmax)	
`set_locs`(self, locs)	
`set_view_interval`(self, vmin, vmax)	

Attributes

`axis`	
`locs`	

librosa.display.NoteFormatter

class `librosa.display.NoteFormatter`(*octave=True, major=True*)[source]

Ticker formatter for Notes

Parameters: **octave** : bool

If *True*, display the octave number along with the note name.

Otherwise, only show the note name (and cent deviation)

major : bool

If *True*, ticks are always labeled.

If *False*, ticks are only labeled if the span is less than 2 octaves

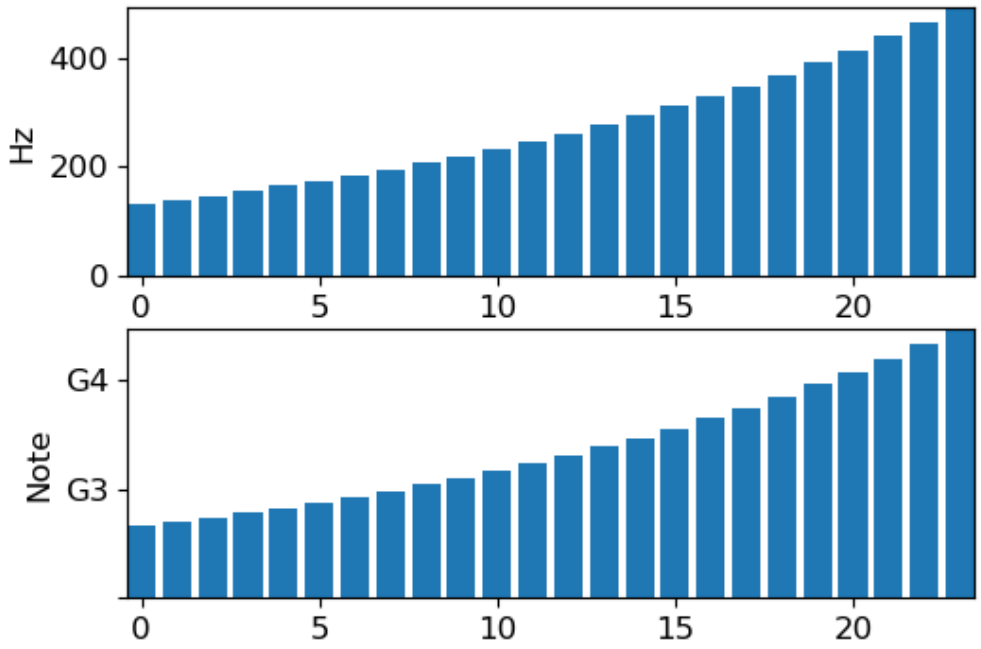
See also

`LogHzFormatter`
`matplotlib.ticker.Formatter`

Examples

```
>>> import matplotlib.pyplot as plt
>>> values = librosa.midi_to_hz(np.arange(48, 72))
>>> plt.figure()
>>> ax1 = plt.subplot(2,1,1)
>>> ax1.bar(np.arange(len(values)), values)
>>> ax1.set_ylabel('Hz')
>>> ax2 = plt.subplot(2,1,2)
>>> ax2.bar(np.arange(len(values)), values)
>>> ax2.yaxis.set_major_formatter(librosa.display.NoteFormatter())
>>> ax2.set_ylabel('Note')
>>> plt.show()
```

(Source code)



__init__(*self*, *octave=True*, *major=True*)[source]

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, octave, major])	Initialize self.
create_dummy_axis(self, **kwargs)	
fix_minus(self, s)	Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for

	typographical correctness.
`format_data`(self, value)	Returns the full string representation of the value with the position unspecified.
`format_data_short`(self, value)	Return a short string version of the tick value.
`format_ticks`(self, values)	Return the tick labels for all the ticks at once.
`get_offset`(self)	
`set_axis`(self, axis)	
`set_bounds`(self, vmin, vmax)	
`set_data_interval`(self, . vmin, vmax)	
`set_locs`(self, locs)	
`set_view_interval`(self, vmin, vmax)	

Attributes

`axis`	
`locs`	

librosa.display.LogHzFormatter

class `librosa.display.LogHzFormatter`(*major=True*)[source]

Ticker formatter for logarithmic frequency

Parameters: **major** : bool

If *True*, ticks are always labeled.

If *False*, ticks are only labeled if the span is less than 2 octaves

See also

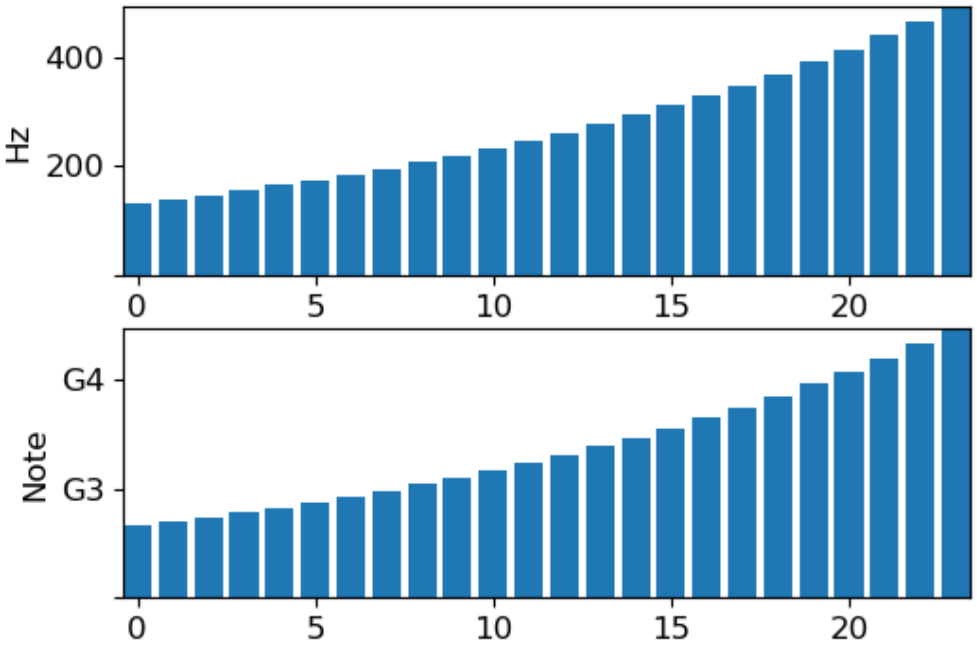
`NoteFormatter`
`matplotlib.ticker.Formatter`

Examples

```
>>> import matplotlib.pyplot as plt
>>> values = librosa.midi_to_hz(np.arange(48, 72))
>>> plt.figure()
>>> ax1 = plt.subplot(2,1,1)
```

```
>>> ax1.bar(np.arange(len(values)), values)
>>> ax1.yaxis.set_major_formatter(librosa.display.LogHzFormatter())
>>> ax1.set_ylabel('Hz')
>>> ax2 = plt.subplot(2,1,2)
>>> ax2.bar(np.arange(len(values)), values)
>>> ax2.yaxis.set_major_formatter(librosa.display.NoteFormatter())
>>> ax2.set_ylabel('Note')
>>> plt.show()
```

(Source code)



__init__(*self*, *major=True*)[source]

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, major])	Initialize self.
create_dummy_axis(self, **kwargs)	
fix_minus(self, s)	Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness.
format_data(self, value)	Returns the full string representation of the value with the position unspecified.
format_data_short(self, value)	Return a short string version of the tick value.
format_ticks(self, values)	Return the tick labels for all the ticks at once.

`get_offset`(self)	
`set_axis`(self, axis)	
`set_bounds`(self, vmin, vmax)	
`set_data_interval`(self, vmin, vmax)	
`set_locs`(self, locs)	
`set_view_interval`(self, vmin, vmax)	

Attributes

`axis`	
`locs`	

librosa.display.ChromaFormatter

class `librosa.display.ChromaFormatter`[source]

A formatter for chroma axes

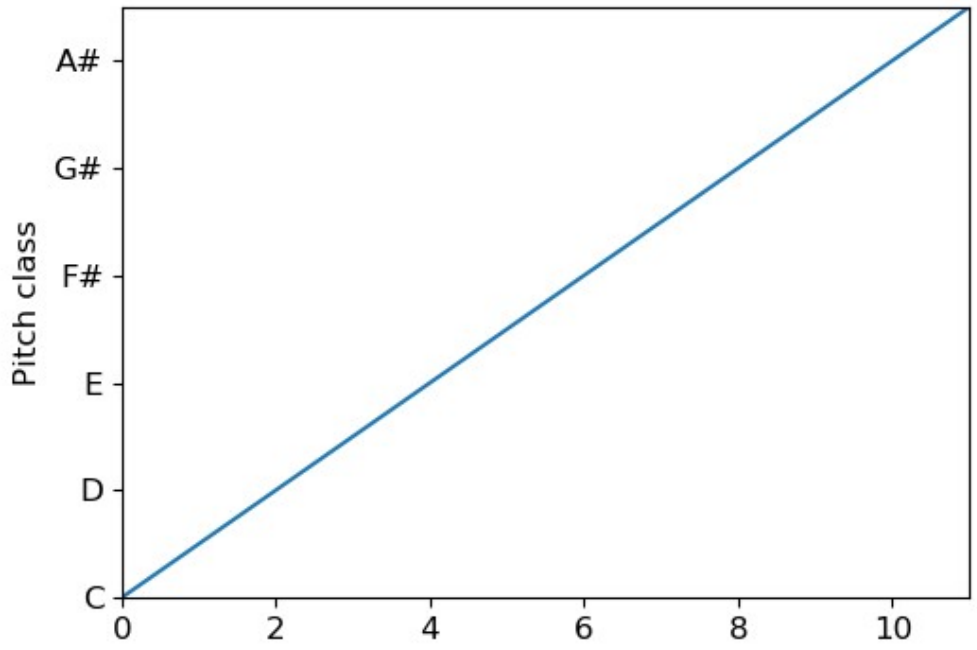
See also

`matplotlib.ticker.Formatter`

Examples

```
>>> import matplotlib.pyplot as plt
>>> values = np.arange(12)
>>> plt.figure()
>>> ax = plt.gca()
>>> ax.plot(values)
>>> ax.yaxis.set_major_formatter(librosa.display.ChromaFormatter())
>>> ax.set_ylabel('Pitch class')
>>> plt.show()
```

(Source code)



__init__(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

create_dummy_axis(self, **kwargs)	
fix_minus(self, s)	Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness.
format_data(self, value)	Returns the full string representation of the value with the position unspecified.
format_data_short(self, value)	Return a short string version of the tick value.
format_ticks(self, values)	Return the tick labels for all the ticks at once.
get_offset(self)	
set_axis(self, axis)	
set_bounds(self, vmin, vmax)	
set_data_interval(self, vmin, vmax)	

`set_locs`(self, locs)	
`set_view_interval`(self, v min, vmax)	

Attributes

`axis`	
`locs`	

librosa.display.TonnetzFormatter

class `librosa.display.TonnetzFormatter`[source]

A formatter for tonnetz axes

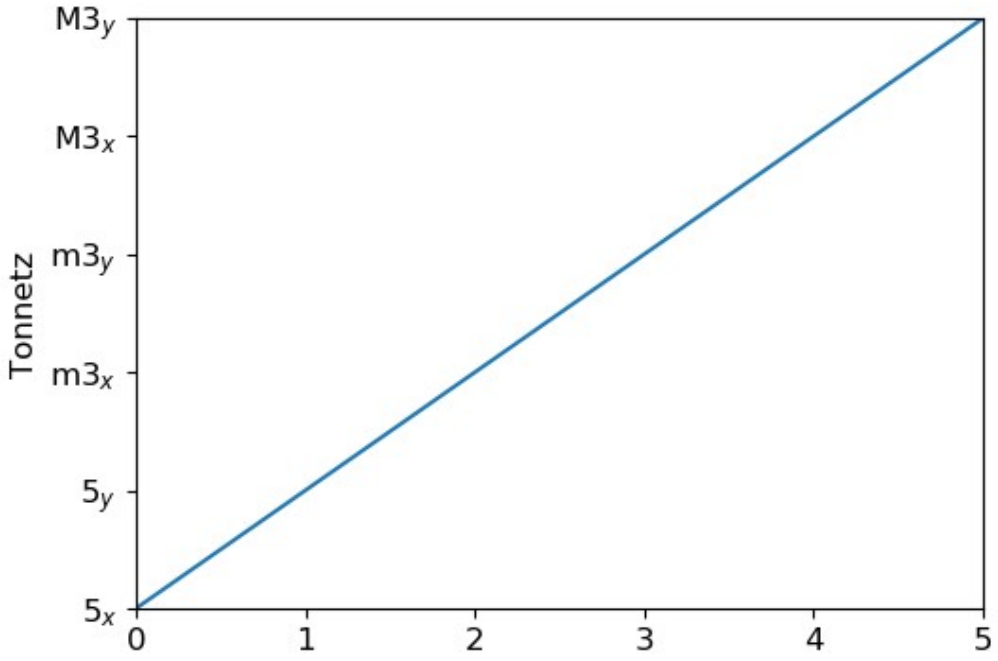
See also

`matplotlib.ticker.Formatter`

Examples

```
>>> import matplotlib.pyplot as plt
>>> values = np.arange(6)
>>> plt.figure()
>>> ax = plt.gca()
>>> ax.plot(values)
>>> ax.yaxis.set_major_formatter(librosa.display.TonnetzFormatter())
>>> ax.set_ylabel('Tonnetz')
>>> plt.show()
```

(Source code)



__init__(*self, /, *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

create_dummy_axis(self, **kwargs)	
fix_minus(self, s)	Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness.
format_data(self, value)	Returns the full string representation of the value with the position unspecified.
format_data_short(self, value)	Return a short string version of the tick value.
format_ticks(self, values)	Return the tick labels for all the ticks at once.
get_offset(self)	
set_axis(self, axis)	
set_bounds(self, vmin, vmax)	
set_data_interval(self, vmin, vmax)	

`set_locs`(self, locs)	
`set_view_interval`(self, vmin, vmax)	

Attributes

`axis`	
`locs`	

Feature extraction

Spectral features

`chroma_stft`([y, sr, S, norm, n_fft, …])	Compute a chromagram from a waveform or power spectrogram.
`chroma_cqt`([y, sr, C, hop_length, fmin, …])	Constant-Q chromagram
`chroma_cens`([y, sr, C, hop_length, fmin, …])	Computes the chroma variant “Chroma Energy Normalized” (CENS), following [R674badebce0d-1].
`melspectrogram`([y, sr, S, n_fft, …])	Compute a mel-scaled spectrogram.
`mfcc`([y, sr, S, n_mfcc, dct_type, norm, lifter])	Mel-frequency cepstral coefficients (MFCCs)
`rms`([y, S, frame_length, hop_length, …])	Compute root-mean-square (RMS) value for each frame, either from the audio samples *y* or from a spectrogram *S*.
`spectral_centroid`([y, sr, S, n_fft, …])	Compute the spectral centroid.
`spectral_bandwidth`([y, sr, S, n_fft, …])	Compute p’th-order spectral bandwidth.
`spectral_contrast`([y, sr, S, n_fft, …])	Compute spectral contrast [R6ffcc01153df-1]
`spectral_flatness`([y, S, n_fft, hop_length, …])	Compute spectral flatness
`spectral_rolloff`([y, sr, S, n_fft, …])	Compute roll-off frequency.
`poly_features`([y, sr, S, n_fft, hop_length, …])	Get coefficients of fitting an nth-order polynomial to the columns of a spectrogram.
`tonnetz`([y, sr, chroma])	Computes the tonal centroid features (tonnetz), following the method of [Recf246e5a035-1].
`zero_crossing_rate`(y[, frame_length, …])	Compute the zero-crossing rate of an audio time series.

Rhythm features

`tempogram`([y, sr, onset_envelope, …])	Compute the tempogram: local autocorrelation of the onset strength envelope.
`fourier_tempogram`([y, sr, onset_envelope, …])	Compute the Fourier tempogram: the short-time Fourier transform of the onset strength

	envelope.

Feature manipulation

`delta`(data[, width, order, axis, mode])	Compute delta features: local estimate of the derivative of the input data along the selected axis.
`stack_memory`(data[, n_steps, delay])	Short-term history embedding: vertically concatenate a data vector or matrix with delayed copies of itself.

Feature inversion

`inverse.mel_to_stft`(M[, sr, n_fft, power])	Approximate STFT magnitude from a Mel power spectrogram.
`inverse.mel_to_audio`(M[, sr, n_fft, …])	Invert a mel power spectrogram to audio using Griffin-Lim.
`inverse.mfcc_to_mel`(mfcc[, n_mels, …])	Invert Mel-frequency cepstral coefficients to approximate a Mel power spectrogram.
`inverse.mfcc_to_audio`(mfcc[, n_mels, …])	Convert Mel-frequency cepstral coefficients to a time-domain audio signal

librosa.feature.chroma_stft

`librosa.feature.chroma_stft`(*y=None, sr=22050, S=None, norm=inf, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', tuning=None, n_chroma=12, **kwargs*)[source]

Compute a chromagram from a waveform or power spectrogram.

This implementation is derived from *chromagram_E* [1]

[1] Ellis, Daniel P.W. “Chroma feature analysis and synthesis” 2007/04/21 http://labrosa.ee.columbia.edu/matlab/chroma-ansyn/

Parameters: y : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

power spectrogram

norm : float or None

Column-wise normalization. See `librosa.util.normalize` for details.

If *None*, no normalization is performed.

n_fft : int > 0 [scalar]

FFT window size if provided *y, sr* instead of *S*

hop_length : int > 0 [scalar]

hop length if provided *y, sr* instead of *S*

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

tuning : float [scalar] or None.

Deviation from A440 tuning in fractional chroma bins. If *None*, it is automatically estimated.

n_chroma : int > 0 [scalar]

Number of chroma bins to produce (12 by default).

kwargs : additional keyword arguments

Arguments to parameterize chroma filters. See

librosa.filters.chroma for details.

Returns: **chromagram** : np.ndarray [shape=(n_chroma, t)]

Normalized energy for each chroma bin at each frame.

See also

librosa.filters.chroma
: Chroma filter bank construction

librosa.util.normalize
: Vector normalization

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.feature.chroma_stft(y=y, sr=sr)
array([[ 0.974,  0.881, ...,  0.925,  1.   ],
       [ 1.   ,  0.841, ...,  0.882,  0.878],
       ...,
       [ 0.658,  0.985, ...,  0.878,  0.764],
       [ 0.969,  0.92 , ...,  0.974,  0.915]])
```

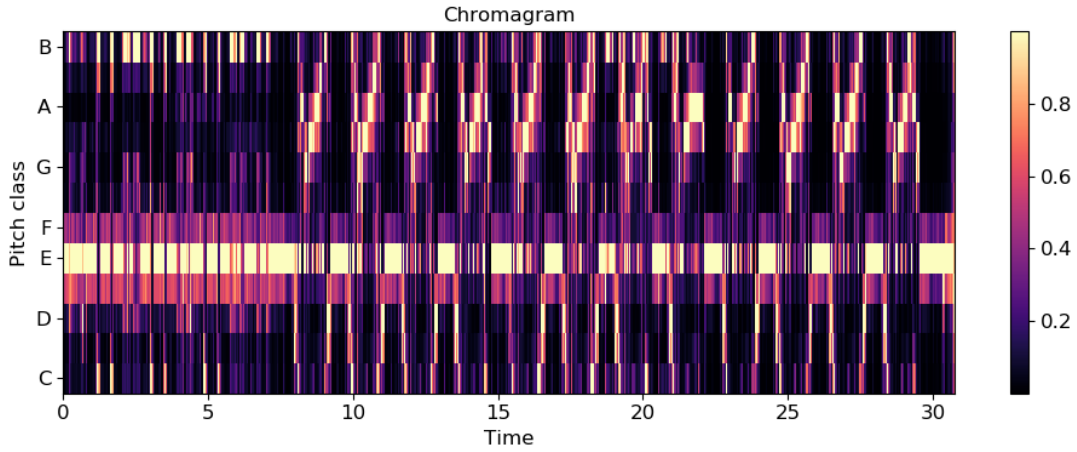
Use an energy (magnitude) spectrum instead of power spectrogram

```
>>> S = np.abs(librosa.stft(y))
>>> chroma = librosa.feature.chroma_stft(S=S, sr=sr)
>>> chroma
array([[ 0.884,  0.91 , ...,  0.861,  0.858],
       [ 0.963,  0.785, ...,  0.968,  0.896],
       ...,
       [ 0.871,  1.   , ...,  0.928,  0.829],
       [ 1.   ,  0.982, ...,  0.93 ,  0.878]])
```

Use a pre-computed power spectrogram with a larger frame

```
>>> S = np.abs(librosa.stft(y, n_fft=4096))**2
>>> chroma = librosa.feature.chroma_stft(S=S, sr=sr)
>>> chroma
array([[ 0.685,  0.477, ...,  0.961,  0.986],
       [ 0.674,  0.452, ...,  0.952,  0.926],
       ...,
       [ 0.844,  0.575, ...,  0.934,  0.869],
       [ 0.793,  0.663, ...,  0.964,  0.972]])
```

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10, 4))
>>> librosa.display.specshow(chroma, y_axis='chroma', x_axis='time')
>>> plt.colorbar()
>>> plt.title('Chromagram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.chroma_cqt

`librosa.feature.chroma_cqt`(*y=None, sr=22050, C=None, hop_length=512, fmin=None, norm=inf, threshold=0.0, tuning=None, n_chroma=12, n_octaves=7, window=None, bins_per_octave=None, cqt_mode='full'*)[source]

Constant-Q chromagram

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0

sampling rate of *y*

C : np.ndarray [shape=(d, t)] [Optional]

a pre-computed constant-Q spectrogram

hop_length : int > 0

number of samples between successive chroma frames

fmin : float > 0

minimum frequency to analyze in the CQT. Default: ‘C1’ ~= 32.7 Hz

norm : int > 0, +-np.inf, or None

Column-wise normalization of the chromagram.

threshold : float

Pre-normalization energy threshold. Values below the threshold are discarded, resulting in a sparse chromagram.

tuning : float

Deviation (in fractions of a CQT bin) from A440 tuning

n_chroma : int > 0

Number of chroma bins to produce

n_octaves : int > 0

Number of octaves to analyze above *fmin*

window : None or np.ndarray

Optional window parameter to *filters.cq_to_chroma*

bins_per_octave : int > 0

Number of bins per octave in the CQT. Default: matches *n_chroma*

cqt_mode : [‘full’, ‘hybrid’]

Constant-Q transform mode

Returns: **chromagram** : np.ndarray [shape=(n_chroma, t)]

The output chromagram

See also

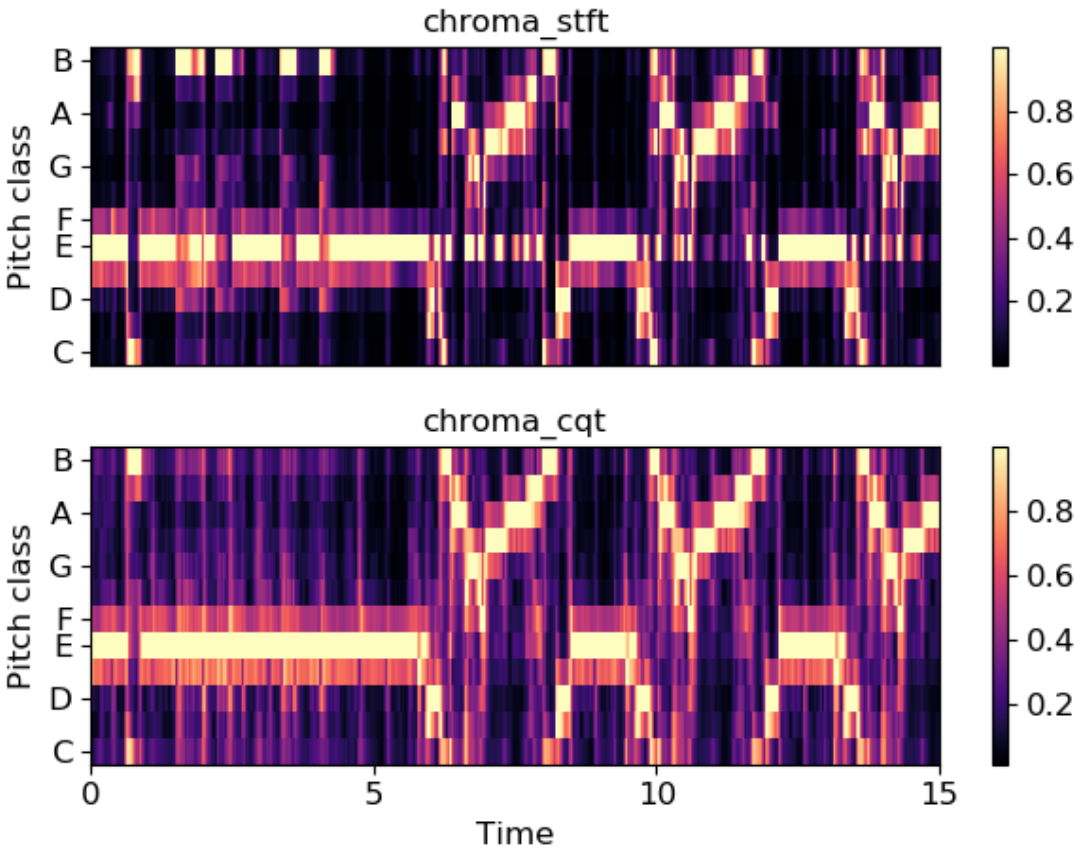
librosa.util.normalize
librosa.core.cqt
librosa.core.hybrid_cqt
chroma_stft

Examples

Compare a long-window STFT chromagram to the CQT chromagram

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10, duration=15)
>>> chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr,
...                                           n_chroma=12, n_fft=4096)
>>> chroma_cq = librosa.feature.chroma_cqt(y=y, sr=sr)

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(chroma_stft, y_axis='chroma')
>>> plt.title('chroma_stft')
>>> plt.colorbar()
```

```
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(chroma_cq, y_axis='chroma', x_axis='time')
>>> plt.title('chroma_cqt')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.chroma_cens

`librosa.feature.chroma_cens`(*y=None, sr=22050, C=None, hop_length=512, fmin=None, tuning=None, n_chroma=12, n_octaves=7, bins_per_octave=None, cqt_mode='full', window=None, norm=2, win_len_smooth=41, smoothing_window='hann'*)[source]

Computes the chroma variant “Chroma Energy Normalized” (CENS), following [1].

To compute CENS features, following steps are taken after obtaining chroma vectors using `chroma_cqt`: 1. L-1 normalization of each chroma vector 2. Quantization of amplitude based on “log-like” amplitude thresholds 3. (optional) Smoothing with sliding window. Default window length = 41 frames 4. (not implemented) Downsampling

CENS features are robust to dynamics, timbre and articulation, thus these are commonly used in audio matching and retrieval applications.

[1] Meinard Müller and Sebastian Ewert “Chroma Toolbox: MATLAB implementations for extracting variants of chroma-based audio features” In Proceedings of the International Conference on Music Information Retrieval (ISMIR), 2011.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0

sampling rate of *y*

C : np.ndarray [shape=(d, t)] [Optional]

a pre-computed constant-Q spectrogram

hop_length : int > 0

number of samples between successive chroma frames

fmin : float > 0

minimum frequency to analyze in the CQT. Default: ‘C1’ ~= 32.7 Hz

norm : int > 0, +-np.inf, or None

Column-wise normalization of the chromagram.

tuning : float

Deviation (in fractions of a CQT bin) from A440 tuning

n_chroma : int > 0

Number of chroma bins to produce

n_octaves : int > 0

Number of octaves to analyze above *fmin*

window : None or np.ndarray

Optional window parameter to *filters.cq_to_chroma*

bins_per_octave : int > 0

Number of bins per octave in the CQT. Default: matches *n_chroma*

cqt_mode : [‘full’, ‘hybrid’]

Constant-Q transform mode

win_len_smooth : int > 0 or None

Length of temporal smoothing window. *None* disables temporal

smoothing. Default: 41

smoothing_window : str, float or tuple

Type of window function for temporal smoothing. See *filters.get_window* for possible inputs. Default: ‘hann’

Returns: **chroma_cens** : np.ndarray [shape=(n_chroma, t)]

The output cens-chromagram

See also

`chroma_cqt`
Compute a chromagram from a constant-Q transform.

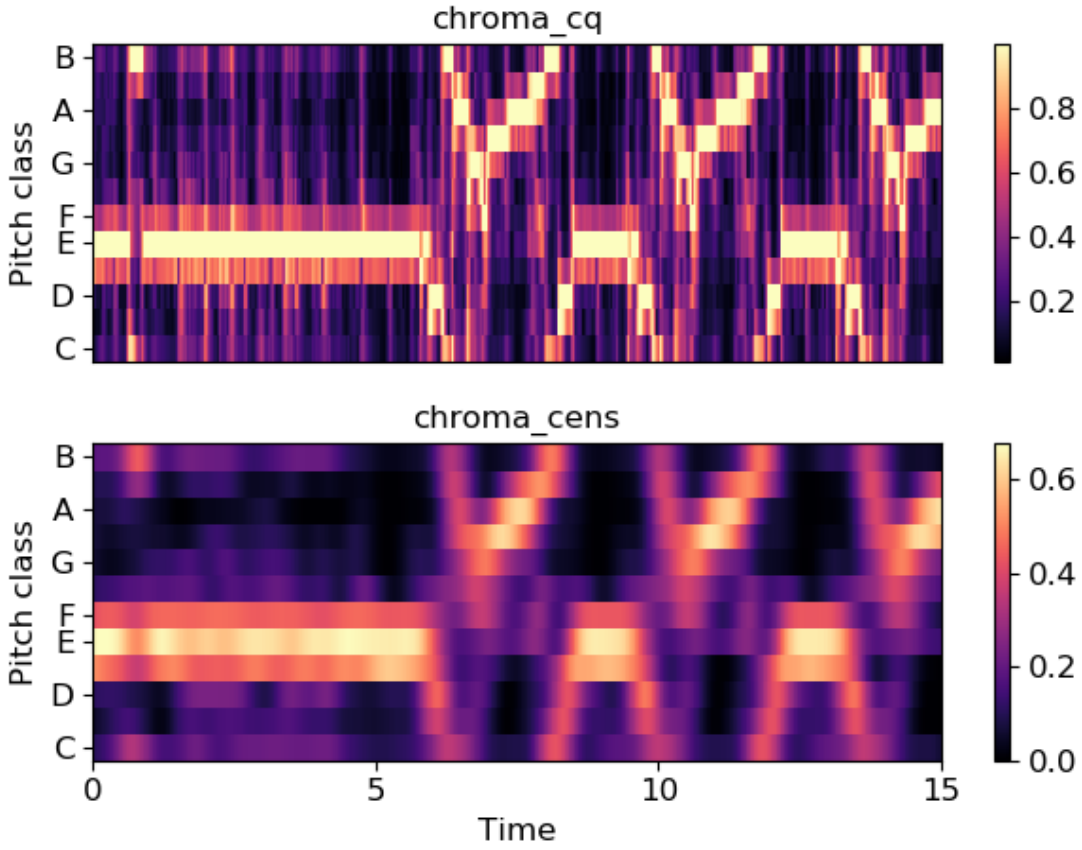
`chroma_stft`
Compute a chromagram from an STFT spectrogram or waveform.

`filters.get_window`
Compute a window function.

Examples

Compare standard cqt chroma to CENS.

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10, duration=15)
>>> chroma_cens = librosa.feature.chroma_cens(y=y, sr=sr)
>>> chroma_cq = librosa.feature.chroma_cqt(y=y, sr=sr)

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(chroma_cq, y_axis='chroma')
>>> plt.title('chroma_cq')
>>> plt.colorbar()
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(chroma_cens, y_axis='chroma', x_axis='time')
>>> plt.title('chroma_cens')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.melspectrogram

`librosa.feature.melspectrogram`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', power=2.0, **kwargs*)[source]

Compute a mel-scaled spectrogram.

If a spectrogram input *S* is provided, then it is mapped directly onto the mel basis *mel_f* by *mel_f.dot(S)*.

If a time-series input *y, sr* is provided, then its magnitude spectrogram *S* is first computed, and then mapped onto the mel scale by *mel_f.dot(S**power)*. By default, *power=2* operates on a power spectrum.

Parameters: y : np.ndarray [shape=(n,)] or None

audio time-series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, t)]

spectrogram

n_fft : int > 0 [scalar]

length of the FFT window

hop_length : int > 0 [scalar]

number of samples between successive frames. See `librosa.core.stft`

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

power : float > 0 [scalar]

Exponent for the magnitude melspectrogram. e.g., 1 for energy, 2 for power, etc.

kwargs : additional keyword arguments

Mel filter bank parameters. See `librosa.filters.mel` for details.

Returns: **S** : np.ndarray [shape=(n_mels, t)]

Mel spectrogram

See also

`librosa.filters.mel`
Mel filter bank construction

`librosa.core.stft`
Short-time Fourier Transform

Examples

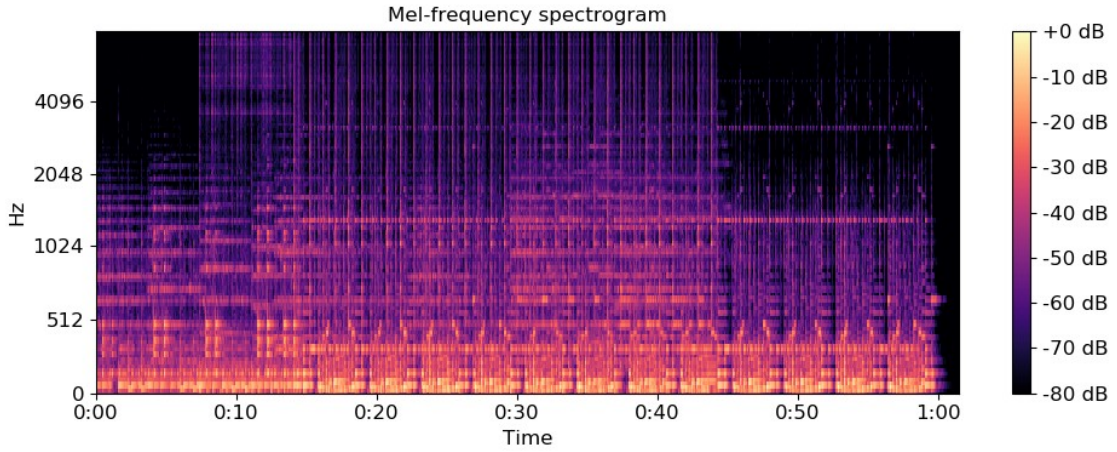
```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.feature.melspectrogram(y=y, sr=sr)
array([[  2.891e-07,   2.548e-03, ...,   8.116e-09,   5.633e-09],
       [  1.986e-07,   1.162e-02, ...,   9.332e-08,   6.716e-09],
       ...,
       [  3.668e-09,   2.029e-08, ...,   3.208e-09,   2.864e-09],
       [  2.561e-10,   2.096e-09, ...,   7.543e-10,   6.101e-10]])
```

Using a pre-computed power spectrogram would give the same result:

```
>>> D = np.abs(librosa.stft(y))**2
>>> S = librosa.feature.melspectrogram(S=D, sr=sr)
```

Display of mel-frequency spectrogram coefficients, with custom arguments for mel filterbank construction (default is fmax=sr/2):

```
>>> # Passing through arguments to the Mel filters
>>> S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128,
...                                    fmax=8000)

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10, 4))
>>> S_dB = librosa.power_to_db(S, ref=np.max)
>>> librosa.display.specshow(S_dB, x_axis='time',
...                          y_axis='mel', sr=sr,
...                          fmax=8000)
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Mel-frequency spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.mfcc

`librosa.feature.mfcc`(*y=None, sr=22050, S=None, n_mfcc=20, dct_type=2, norm='ortho', lifter=0, **kwargs*)[source]

Mel-frequency cepstral coefficients (MFCCs)

Parameters: y : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

log-power Mel spectrogram

n_mfcc: int > 0 [scalar]

number of MFCCs to return

dct_type : None, or {1, 2, 3}

Discrete cosine transform (DCT) type. By default, DCT type-2 is used.

norm : None or ‘ortho’

If *dct_type* is *2 or 3*, setting *norm=’ortho’* uses an ortho-normal DCT basis.

Normalization is not supported for *dct_type=1*.

lifter : number >= 0

If *lifter>0*, apply *liftering* (cepstral filtering) to the MFCCs:

*M[n, :] <- M[n, :] * (1 + sin(pi * (n + 1) / lifter)) * lifter / 2*

Setting *lifter >= 2 * n_mfcc* emphasizes the higher-order coefficients. As *lifter* increases, the coefficient weighting becomes approximately linear.

kwargs : additional keyword arguments

Arguments to `melspectrogram`, if operating on time series input

Returns: **M** : np.ndarray [shape=(n_mfcc, t)]

MFCC sequence

See also

`melspectrogram`
`scipy.fftpack.dct`

Examples

Generate mfccs from a time series

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=30,
duration=5)
>>> librosa.feature.mfcc(y=y, sr=sr)
array([[ -5.229e+02,  -4.944e+02, ...,  -5.229e+02,  -5.229e+02],
       [  7.105e-15,   3.787e+01, ...,  -7.105e-15,  -7.105e-15],
       ...,
       [  1.066e-14,  -7.500e+00, ...,   1.421e-14,   1.421e-14],
       [  3.109e-14,  -5.058e+00, ...,   2.931e-14,   2.931e-14]])
```

Using a different hop length and HTK-style Mel frequencies

```
>>> librosa.feature.mfcc(y=y, sr=sr, hop_length=1024, htk=True)
array([[-1.628e+02, -8.903e+01, -1.409e+02, ..., -1.078e+02,
    -2.504e+02, -2.393e+02],
   [ 1.275e+02,  9.532e+01,  1.019e+02, ...,  1.152e+02,
     2.224e+02,  1.750e+02],
   [ 1.139e+01,  6.155e+00,  1.266e+01, ...,  4.557e+01,
     4.585e+01,  3.985e+01],
   ...,
   [ 3.462e+00,  4.032e+00, -5.694e-01, ..., -6.677e+00,
    -1.183e-01,  1.485e+00],
   [ 9.569e-01,  1.069e+00, -6.865e+00, ..., -9.598e+00,
    -1.611e+00, -6.716e+00],
   [ 8.457e+00,  3.582e+00, -1.156e-01, ..., -3.018e+00,
    -1.456e+01, -6.991e+00]], dtype=float32)
```

Use a pre-computed log-power Mel spectrogram

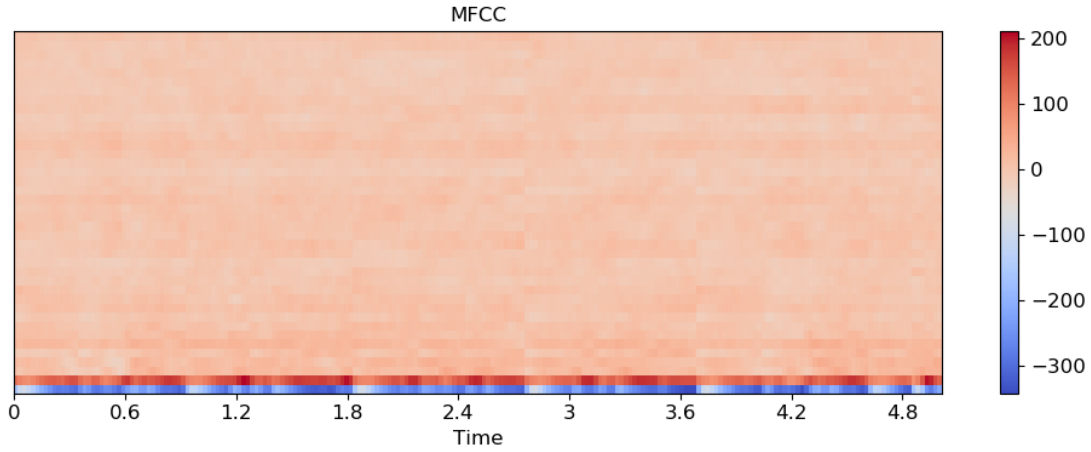
```
>>> S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128,
...                                    fmax=8000)
>>> librosa.feature.mfcc(S=librosa.power_to_db(S))
array([[ -5.207e+02,  -4.898e+02, ...,  -5.207e+02,  -5.207e+02],
       [ -2.576e-14,   4.054e+01, ...,  -3.997e-14,  -3.997e-14],
       ...,
       [  7.105e-15,  -3.534e+00, ...,   0.000e+00,   0.000e+00],
       [  3.020e-14,  -2.613e+00, ...,   3.553e-14,   3.553e-14]])
```

Get more components

```
>>> mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
```

Visualize the MFCC series

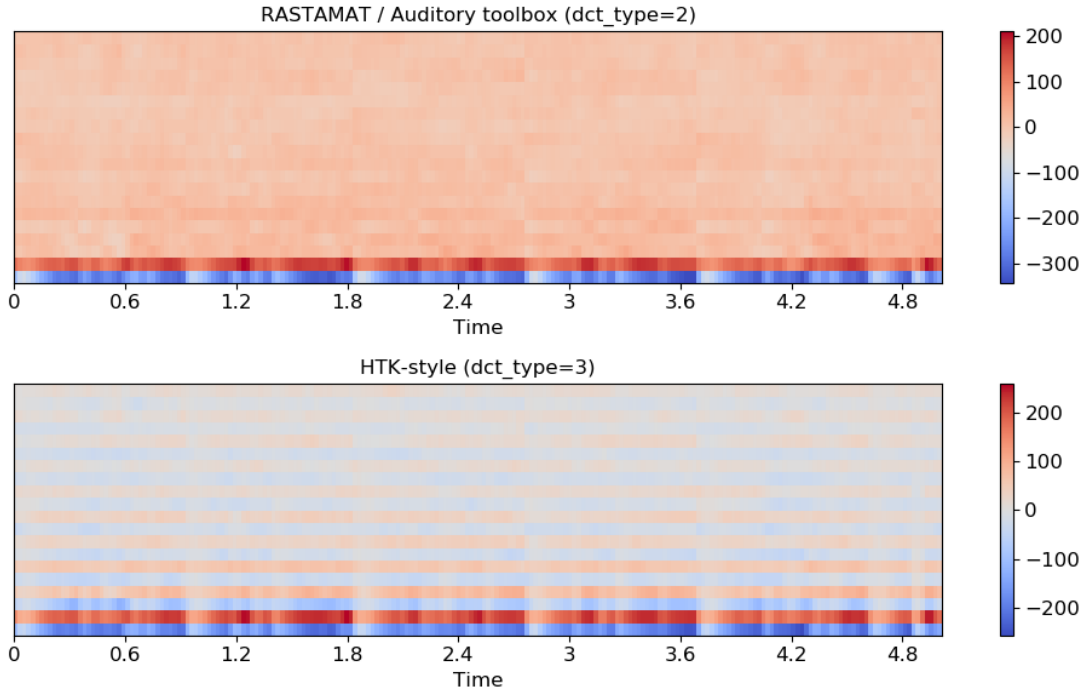
```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10, 4))
>>> librosa.display.specshow(mfccs, x_axis='time')
>>> plt.colorbar()
>>> plt.title('MFCC')
>>> plt.tight_layout()
```




```
>>> plt.show()
```

Compare different DCT bases

```
>>> m_slaney = librosa.feature.mfcc(y=y, sr=sr, dct_type=2)
>>> m_htk = librosa.feature.mfcc(y=y, sr=sr, dct_type=3)
>>> plt.figure(figsize=(10, 6))
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(m_slaney, x_axis='time')
>>> plt.title('RASTAMAT / Auditory toolbox (dct_type=2)')
>>> plt.colorbar()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(m_htk, x_axis='time')
>>> plt.title('HTK-style (dct_type=3)')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.rms

`librosa.feature.rms`(*y=None*, *S=None*, *frame_length=2048*, *hop_length=512*, *center=True*, *pad_mode='reflect'*)[source]

Compute root-mean-square (RMS) value for each frame, either from the audio samples *y* or from a spectrogram *S*.

Computing the RMS value from audio samples is faster as it doesn't require a STFT calculation. However, using a spectrogram will give a more accurate representation of energy over time because its frames can be windowed, thus prefer using *S* if it's already available.

Parameters: **y** : np.ndarray [shape=(n,)] or None

(optional) audio time series. Required if *S* is not input.

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude. Required if *y* is not input.

frame_length : int > 0 [scalar]

length of analysis frame (in samples) for energy calculation

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

center : bool

If *True* and operating on time-domain input (*y*), pad the signal by

frame_length//2 on either side.

If operating on spectrogram input, this has no effect.

pad_mode : str

Padding mode for centered analysis. See *np.pad* for valid values.

Returns: **rms** : np.ndarray [shape=(1, t)]

RMS value for each frame

Examples

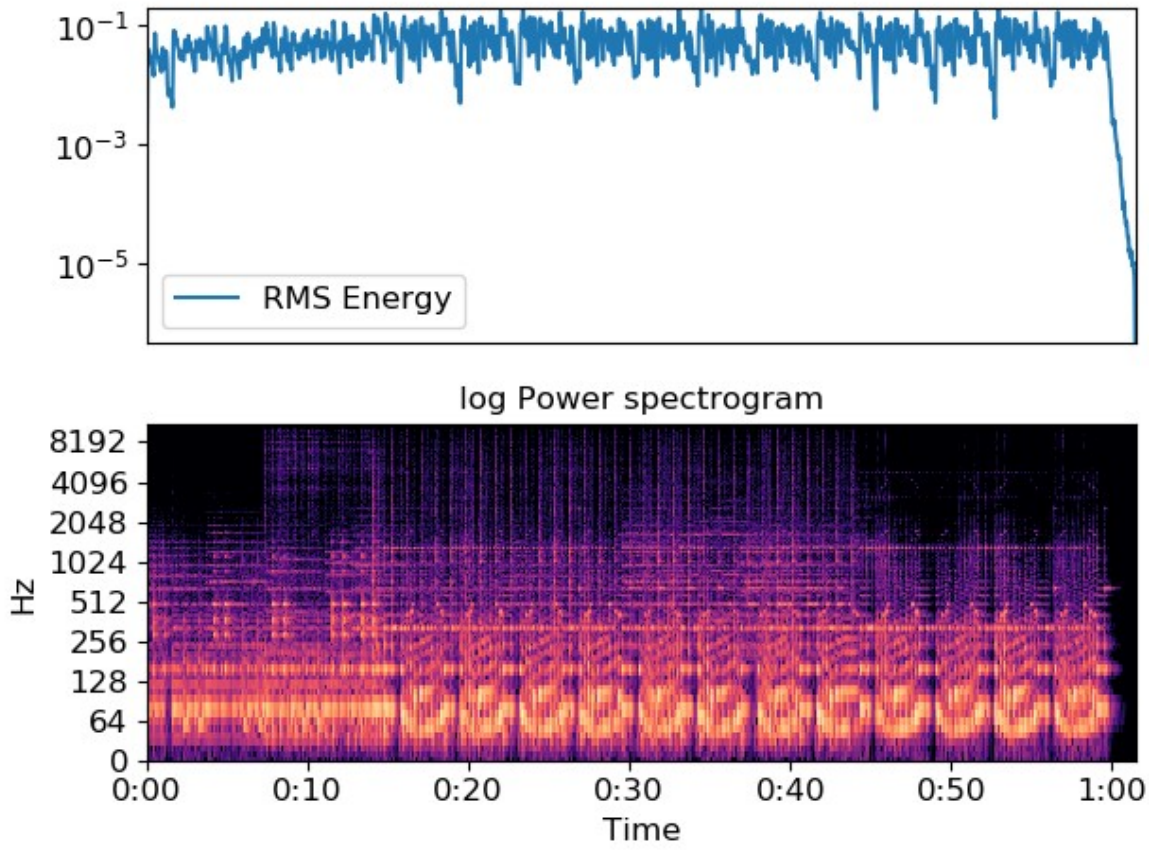
```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.feature.rms(y=y)
array([[ 0.   ,  0.056, ...,  0.   ,  0.   ]], dtype=float32)
```

Or from spectrogram input

```
>>> S, phase = librosa.magphase(librosa.stft(y))
>>> rms = librosa.feature.rms(S=S)

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(rms.T, label='RMS Energy')
>>> plt.xticks([])
>>> plt.xlim([0, rms.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
```

Use a STFT window of constant ones and no frame centering to get consistent results with the RMS computed from the audio samples *y*

```
>>> S = librosa.magphase(librosa.stft(y, window=np.ones, center=False))[0]
>>> librosa.feature.rms(S=S)
>>> plt.show()
```



librosa.feature.spectral_centroid

`librosa.feature.spectral_centroid`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, freq=None, win_length=None, window='hann', center=True, pad_mode='reflect'*) [source]

Compute the spectral centroid.

Each frame of a magnitude spectrogram is normalized and treated as a distribution over frequency bins, from which the mean (centroid) is extracted per frame.

More precisely, the centroid at frame *t* is defined as [1]:

```
centroid[t] = sum_k S[k, t] * freq[k] / (sum_j S[j, t])
```

where *S* is a magnitude spectrogram, and *freq* is the array of frequencies (e.g., FFT frequencies in Hz) of the rows of *S*.

[1] Klapuri, A., & Davy, M. (Eds.). (2007). Signal processing methods for music transcription, chapter 5. Springer Science & Business Media.

Parameters **y** : np.ndarray [shape=(n,)] or None
:
audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies, or a matrix of center frequencies as constructed by `librosa.core.ifgram`

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

Returns: **centroid** : np.ndarray [shape=(1, t)]

centroid frequencies

See also

`librosa.core.stft`
Short-time Fourier Transform
`librosa.core.ifgram`

Instantaneous-frequency spectrogram

Examples

From time-series input:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> cent = librosa.feature.spectral_centroid(y=y, sr=sr)
>>> cent
array([[ 4382.894,   626.588, ...,  5037.07 ,  5413.398]])
```

From spectrogram input:

```
>>> S, phase = librosa.magphase(librosa.stft(y=y))
>>> librosa.feature.spectral_centroid(S=S)
array([[ 4382.894,   626.588, ...,  5037.07 ,  5413.398]])
```

Using variable bin center frequencies:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> if_gram, D = librosa.ifgram(y)
>>> librosa.feature.spectral_centroid(S=np.abs(D), freq=if_gram)
array([[ 4420.719,   625.769, ...,  5011.86 ,  5221.492]])
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(cent.T, label='Spectral centroid')
>>> plt.ylabel('Hz')
>>> plt.xticks([])
>>> plt.xlim([0, cent.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```

librosa.feature.spectral_centroid

`librosa.feature.spectral_centroid`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, freq=None, win_length=None, window='hann', center=True, pad_mode='reflect'*) [source]

Compute the spectral centroid.

Each frame of a magnitude spectrogram is normalized and treated as a distribution over frequency bins, from which the mean (centroid) is extracted per frame.

More precisely, the centroid at frame *t* is defined as [1]:

```
centroid[t] = sum_k S[k, t] * freq[k] / (sum_j S[j, t])
```

where *S* is a magnitude spectrogram, and *freq* is the array of frequencies (e.g., FFT frequencies in Hz) of the rows of *S*.

[1] Klapuri, A., & Davy, M. (Eds.). (2007). Signal processing methods for music transcription, chapter 5. Springer Science & Business Media.

Parameters :

y : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies, or a matrix of center frequencies as constructed by `librosa.core.ifgram`

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

Returns: **centroid** : np.ndarray [shape=(1, t)]

centroid frequencies

See also

`librosa.core.stft`
Short-time Fourier Transform

`librosa.core.ifgram`
Instantaneous-frequency spectrogram

Examples

From time-series input:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> cent = librosa.feature.spectral_centroid(y=y, sr=sr)
>>> cent
array([[ 4382.894,   626.588, ...,  5037.07 ,  5413.398]])
```

From spectrogram input:

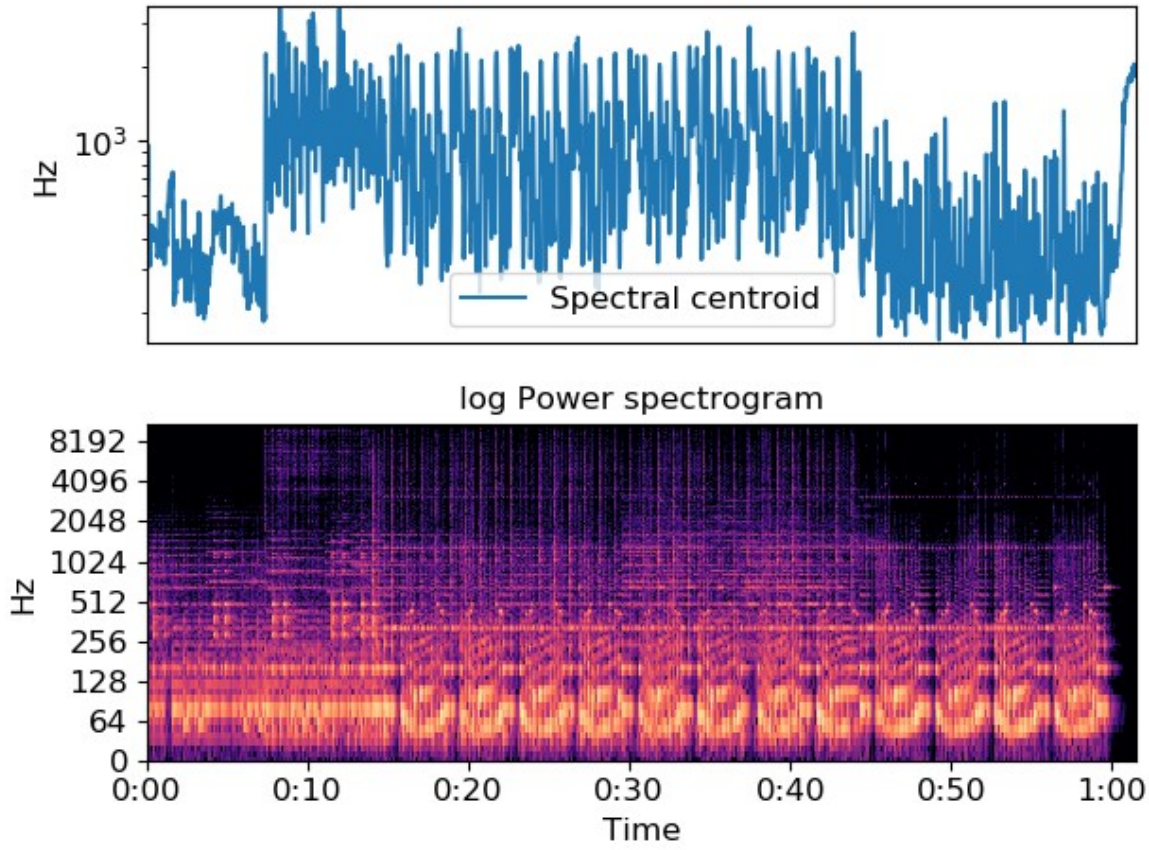
```
>>> S, phase = librosa.magphase(librosa.stft(y=y))
>>> librosa.feature.spectral_centroid(S=S)
array([[ 4382.894,   626.588, ...,  5037.07 ,  5413.398]])
```

Using variable bin center frequencies:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> if_gram, D = librosa.ifgram(y)
>>> librosa.feature.spectral_centroid(S=np.abs(D), freq=if_gram)
array([[ 4420.719,   625.769, ...,  5011.86 ,  5221.492]])
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(cent.T, label='Spectral centroid')
>>> plt.ylabel('Hz')
>>> plt.xticks([])
>>> plt.xlim([0, cent.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.spectral_bandwidth

`librosa.feature.spectral_bandwidth`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', freq=None, centroid=None, norm=True, p=2*)[source]

Compute p’th-order spectral bandwidth.

The spectral bandwidth [1] at frame *t* is computed by

```
(sum_k S[k, t] * (freq[k, t] - centroid[t])**p)**(1/p)
```

[1] Klapuri, A., & Davy, M. (Eds.). (2007). Signal processing methods for music transcription, chapter 5. Springer Science & Business Media.

Parameters **y** : np.ndarray [shape=(n,)] or None
:

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies, or a matrix of center frequencies as constructed by `librosa.core.ifgram`

centroid : None or np.ndarray [shape=(1, t)]

pre-computed centroid frequencies

norm : bool

Normalize per-frame spectral energy (sum to one)

p : float > 0

Power to raise deviation from spectral centroid.

Returns: **bandwidth** : np.ndarray [shape=(1, t)]

frequency bandwidth for each frame

Examples

From time-series input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> spec_bw = librosa.feature.spectral_bandwidth(y=y, sr=sr)
>>> spec_bw
array([[ 3379.878,  1429.486, ...,  3235.214,  3080.148]])
```

From spectrogram input

```
>>> S, phase = librosa.magphase(librosa.stft(y=y))
>>> librosa.feature.spectral_bandwidth(S=S)
array([[ 3379.878,  1429.486, ...,  3235.214,  3080.148]])
```

Using variable bin center frequencies

```
>>> if_gram, D = librosa.ifgram(y)
>>> librosa.feature.spectral_bandwidth(S=np.abs(D), freq=if_gram)
array([[ 3380.011,  1429.11 , ...,  3235.22 ,  3080.148]])
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(spec_bw.T, label='Spectral bandwidth')
>>> plt.ylabel('Hz')
>>> plt.xticks([])
>>> plt.xlim([0, spec_bw.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```

librosa.feature.spectral_bandwidth

`librosa.feature.spectral_bandwidth`(*y=None*, *sr=22050*, *S=None*, *n_fft=2048*, *hop_length=512*, *win_length=None*, *window='hann'*, *center=True*, *pad_mode='reflect'*, *freq=None*, *centroid=None*, *norm=True*, *p=2*)[source]

Compute p’th-order spectral bandwidth.

The spectral bandwidth [1] at frame *t* is computed by

(sum_k S[k, t] * (freq[k, t] - centroid[t])**p)**(1/p)

[1] Klapuri, A., & Davy, M. (Eds.). (2007). Signal processing methods for music transcription, chapter 5. Springer Science & Business Media.

Parameters **y** : np.ndarray [shape=(n,)] or None

:

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies, or a matrix of center frequencies as constructed by `librosa.core.ifgram`

centroid : None or np.ndarray [shape=(1, t)]

pre-computed centroid frequencies

norm : bool

Normalize per-frame spectral energy (sum to one)

p : float > 0

Power to raise deviation from spectral centroid.

Returns: **bandwidth** : np.ndarray [shape=(1, t)]

frequency bandwidth for each frame

Examples

From time-series input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> spec_bw = librosa.feature.spectral_bandwidth(y=y, sr=sr)
>>> spec_bw
array([[ 3379.878,  1429.486, ...,  3235.214,  3080.148]])
```

From spectrogram input

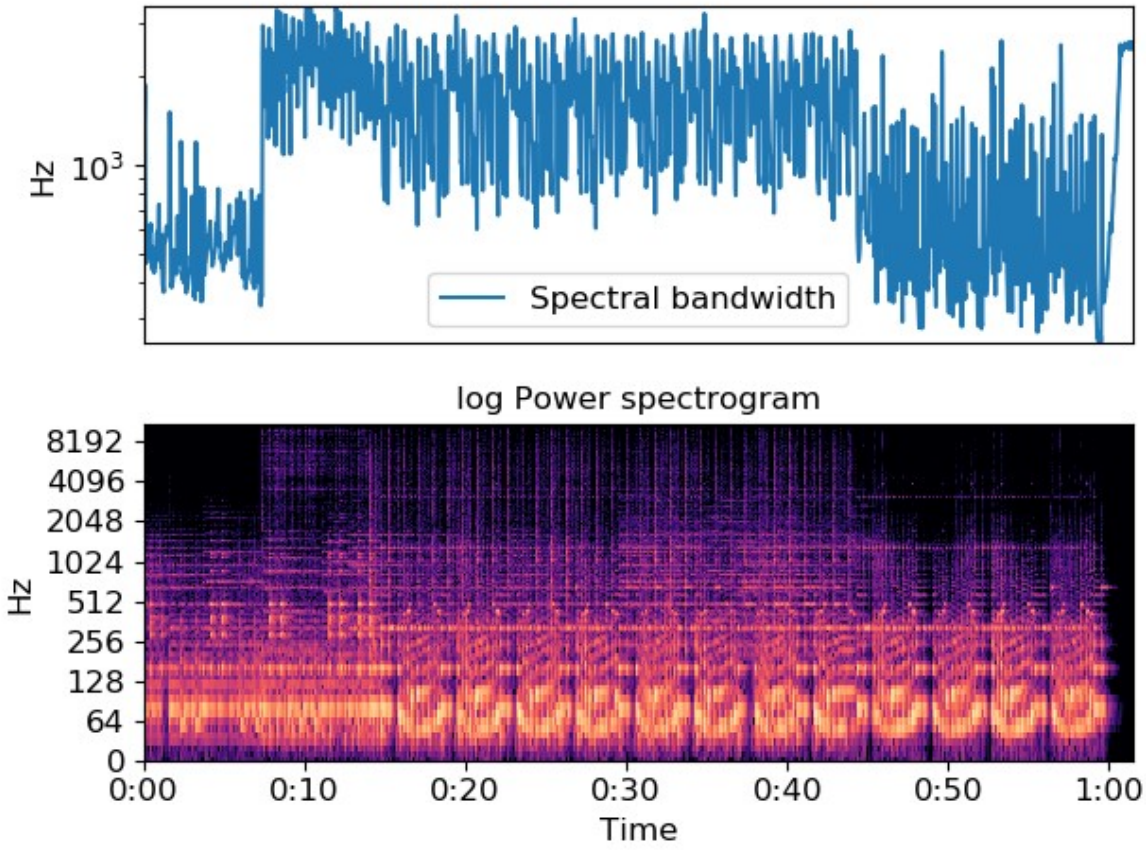
```
>>> S, phase = librosa.magphase(librosa.stft(y=y))
>>> librosa.feature.spectral_bandwidth(S=S)
array([[ 3379.878,  1429.486, ...,  3235.214,  3080.148]])
```

Using variable bin center frequencies

```
>>> if_gram, D = librosa.ifgram(y)
>>> librosa.feature.spectral_bandwidth(S=np.abs(D), freq=if_gram)
array([[ 3380.011,  1429.11 , ...,  3235.22 ,  3080.148]])
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(spec_bw.T, label='Spectral bandwidth')
>>> plt.ylabel('Hz')
>>> plt.xticks([])
>>> plt.xlim([0, spec_bw.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.spectral_contrast

`librosa.feature.spectral_contrast`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', freq=None, fmin=200.0, n_bands=6, quantile=0.02, linear=False*)[source]

Compute spectral contrast [1]

Each frame of a spectrogram *S* is divided into sub-bands. For each sub-band, the energy contrast is estimated by comparing the mean energy in the top quantile (peak energy) to that of the bottom quantile (valley energy). High contrast values generally correspond to clear, narrow-band signals, while low contrast values correspond to broad-band noise.

[1] Jiang, Dan-Ning, Lie Lu, Hong-Jiang Zhang, Jian-Hua Tao, and Lian-Hong Cai. “Music type classification by spectral contrast feature.” In Multimedia and Expo, 2002. ICME‘02. Proceedings. 2002 IEEE International Conference on, vol. 1, pp. 113-116. IEEE, 2002.

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

freq : None or np.ndarray [shape=(d,)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies.

fmin : float > 0

Frequency cutoff for the first bin *[0, fmin]* Subsequent bins will cover *[fmin, 2*fmin]*, *[2*fmin, 4*fmin]*, etc.

n_bands : int > 1

number of frequency bands

quantile : float in (0, 1)

quantile for determining peaks and valleys

linear : bool

If *True*, return the linear difference of magnitudes: *peaks* - *valleys*.

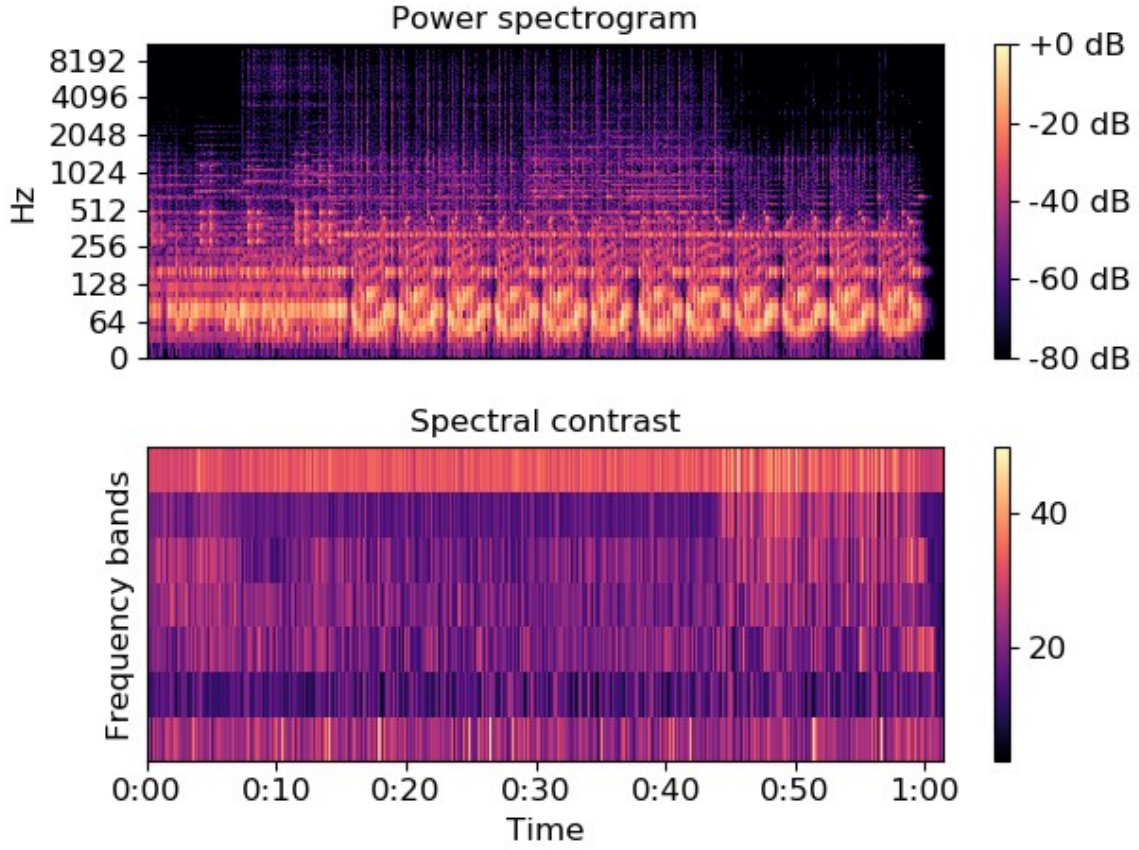
If *False*, return the logarithmic difference: *log(peaks)* - *log(valleys)*.

Returns: **contrast** : np.ndarray [shape=(n_bands + 1, t)]

each row of spectral contrast values corresponds to a given octave-based frequency

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
>>> contrast = librosa.feature.spectral_contrast(S=S, sr=sr)

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(S,
...                                                  ref=np.max),
...                           y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Power spectrogram')
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(contrast, x_axis='time')
>>> plt.colorbar()
>>> plt.ylabel('Frequency bands')
>>> plt.title('Spectral contrast')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.spectral_flatness

`librosa.feature.spectral_flatness`(*y=None, S=None, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', amin=1e-10, power=2.0*) [source]

Compute spectral flatness

Spectral flatness (or tonality coefficient) is a measure to quantify how much noise-like a sound is, as opposed to being tone-like [1]. A high spectral flatness (closer to 1.0) indicates the spectrum is similar to white noise. It is often converted to decibel.

[1] Dubnov, Shlomo “Generalization of spectral flatness measure for non-gaussian linear processes” IEEE Signal Processing Letters, 2004, Vol. 11.

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

S : np.ndarray [shape=(d, t)] or None

(optional) pre-computed spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

amin : float > 0 [scalar]

minimum threshold for *S* (=added noise floor for numerical stability)

power : float > 0 [scalar]

Exponent for the magnitude spectrogram. e.g., 1 for energy, 2 for power, etc. Power spectrogram is usually used for computing spectral flatness.

Returns: **flatness** : np.ndarray [shape=(1, t)]

spectral flatness for each frame. The returned value is in [0, 1] and often converted to dB scale.

Examples

From time-series input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> flatness = librosa.feature.spectral_flatness(y=y)
>>> flatness
array([[  1.00000e+00,   5.82299e-03,   5.64624e-04, ...,   9.99063e-01,
      1.00000e+00,   1.00000e+00]], dtype=float32)
```

From spectrogram input

```
>>> S, phase = librosa.magphase(librosa.stft(y))
```

```
>>> librosa.feature.spectral_flatness(S=S)
array([[  1.00000e+00,   5.82299e-03,   5.64624e-04, ...,   9.99063e-01,
      1.00000e+00,   1.00000e+00]], dtype=float32)
```

From power spectrogram input

```
>>> S, phase = librosa.magphase(librosa.stft(y))
>>> S_power = S ** 2
>>> librosa.feature.spectral_flatness(S=S_power, power=1.0)
array([[  1.00000e+00,   5.82299e-03,   5.64624e-04, ...,   9.99063e-01,
      1.00000e+00,   1.00000e+00]], dtype=float32)
```

librosa.feature.spectral_rolloff

`librosa.feature.spectral_rolloff`(*y=None*, *sr=22050*, *S=None*, *n_fft=2048*, *hop_length=512*, *win_length=None*, *window='hann'*, *center=True*, *pad_mode='reflect'*, *freq=None*, *roll_percent=0.85*)[source]

Compute roll-off frequency.

The roll-off frequency is defined for each frame as the center frequency for a spectrogram bin such that at least roll_percent (0.85 by default) of the energy of the spectrum in this frame is contained in this bin and the bins below. This can be used to, e.g., approximate the maximum (or minimum) frequency by setting roll_percent to a value close to 1 (or 0).

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies,

Note

freq is assumed to be sorted in increasing order

roll_percent : float [0 < roll_percent < 1]

Roll-off percentage.

Returns: **rolloff** : np.ndarray [shape=(1, t)]

roll-off frequency for each frame

Examples

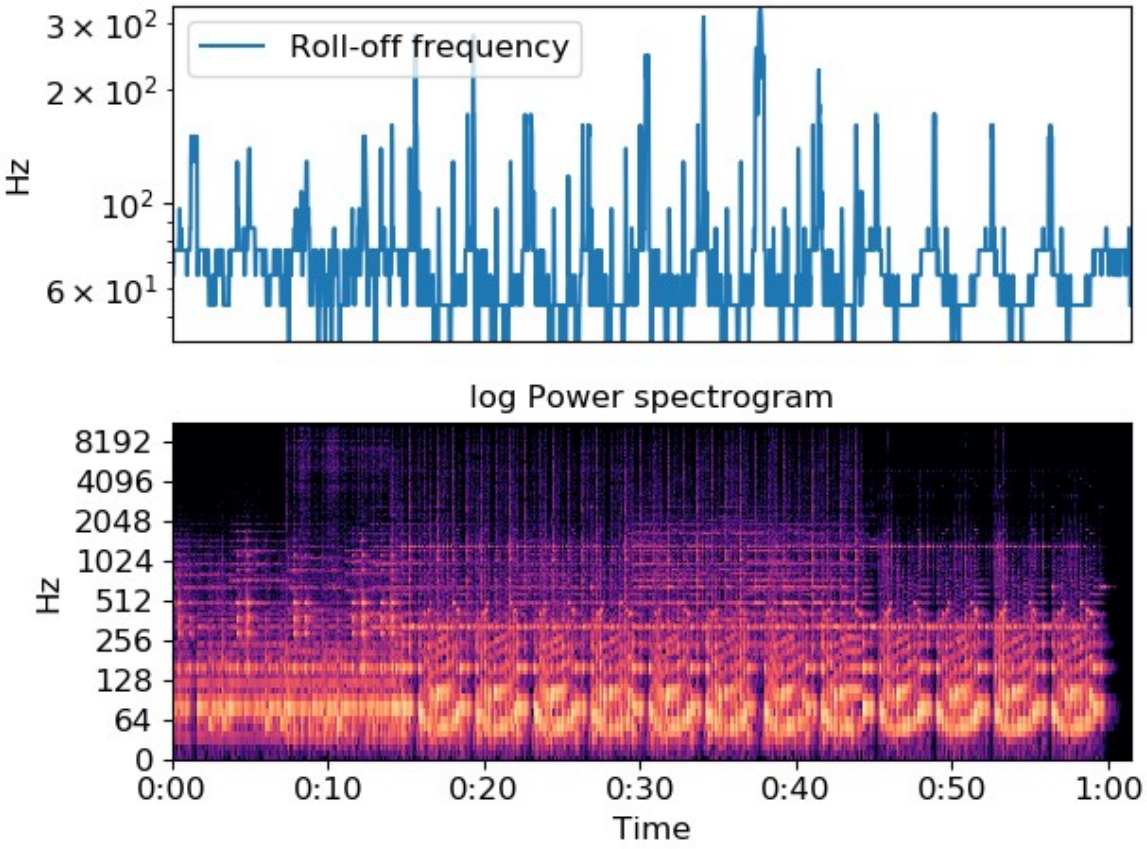
From time-series input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> # Approximate maximum frequencies with roll_percent=0.85 (default)
>>> rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
>>> rolloff
array([[ 8376.416,   968.994, ...,  8925.513,  9108.545]])
>>> # Approximate minimum frequencies with roll_percent=0.1
>>> rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr,
roll_percent=0.1)
>>> rolloff
array([[ 75.36621094,  64.59960938,  64.59960938, ...,  75.36621094,
     75.36621094,  64.59960938]])
```

From spectrogram input

```
>>> S, phase = librosa.magphase(librosa.stft(y))
>>> librosa.feature.spectral_rolloff(S=S, sr=sr)
```

```
array([[ 8376.416,   968.994, ...,  8925.513,  9108.545]])

>>> # With a higher roll percentage:
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.feature.spectral_rolloff(y=y, sr=sr, roll_percent=0.95)
array([[ 10012.939,   3003.882, ...,  10034.473,  10077.539]])

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> plt.semilogy(rolloff.T, label='Roll-off frequency')
>>> plt.ylabel('Hz')
>>> plt.xticks([])
>>> plt.xlim([0, rolloff.shape[-1]])
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('log Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.poly_features

`librosa.feature.poly_features`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', order=1, freq=None*)[source]

Get coefficients of fitting an nth-order polynomial to the columns of a spectrogram.

Parameters :

y : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

S : np.ndarray [shape=(d, t)] or None

(optional) spectrogram magnitude

n_fft : int > 0 [scalar]

FFT window size

hop_length : int > 0 [scalar]

hop length for STFT. See `librosa.core.stft` for details.

win_length : int <= n_fft [scalar]

Each frame of audio is windowed by *window()*. The window will be of length *win_length* and then padded with zeros to match *n_fft*.

If unspecified, defaults to `win_length = n_fft`.

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

- a window specification (string, tuple, or number); see `scipy.signal.get_window`
- a window function, such as `scipy.signal.hanning`
- a vector or array of length *n_fft*

center : boolean

- If *True*, the signal *y* is padded so that frame *t* is centered at *y[t * hop_length]*.
- If *False*, then frame *t* begins at *y[t * hop_length]*

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

order : int > 0

order of the polynomial to fit

freq : None or np.ndarray [shape=(d,) or shape=(d, t)]

Center frequencies for spectrogram bins. If *None*, then FFT bin center frequencies are used. Otherwise, it can be a single array of *d* center frequencies, or a matrix of center frequencies as constructed by `librosa.core.ifgram`

Returns: **coefficients** : np.ndarray [shape=(order+1, t)]

polynomial coefficients for each frame.

coeffecients[0] corresponds to the highest degree (*order*),

coefficients[1] corresponds to the next highest degree (*order-1*),

down to the constant term *coefficients[order]*.

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
```

Fit a degree-0 polynomial (constant) to each frame

```
>>> p0 = librosa.feature.poly_features(S=S, order=0)
```

Fit a linear polynomial to each frame

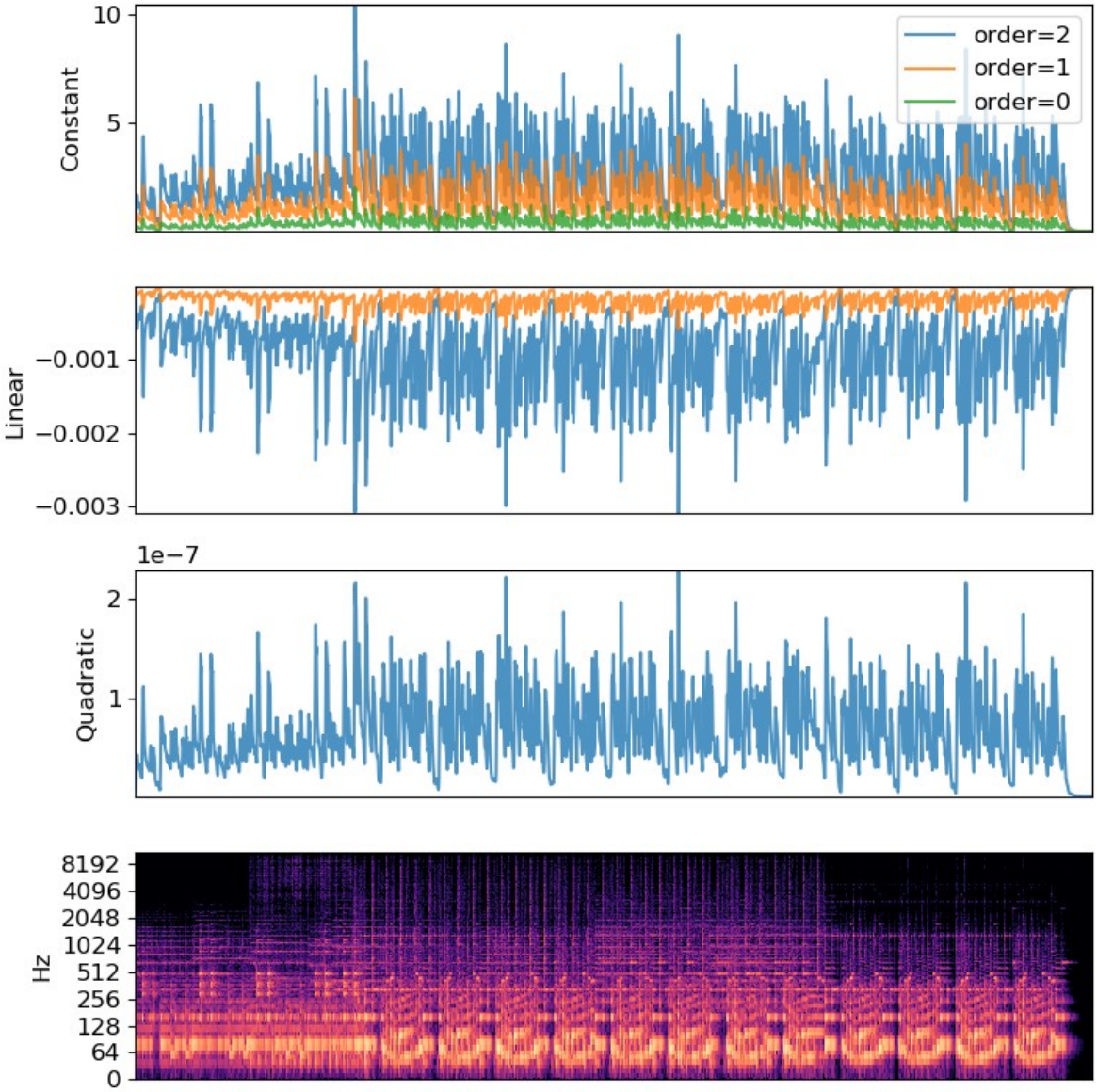
```
>>> p1 = librosa.feature.poly_features(S=S, order=1)
```

Fit a quadratic to each frame

```
>>> p2 = librosa.feature.poly_features(S=S, order=2)
```

Plot the results for comparison

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 8))
>>> ax = plt.subplot(4,1,1)
>>> plt.plot(p2[2], label='order=2', alpha=0.8)
>>> plt.plot(p1[1], label='order=1', alpha=0.8)
>>> plt.plot(p0[0], label='order=0', alpha=0.8)
>>> plt.xticks([])
>>> plt.ylabel('Constant')
>>> plt.legend()
>>> plt.subplot(4,1,2, sharex=ax)
>>> plt.plot(p2[1], label='order=2', alpha=0.8)
>>> plt.plot(p1[0], label='order=1', alpha=0.8)
>>> plt.xticks([])
>>> plt.ylabel('Linear')
>>> plt.subplot(4,1,3, sharex=ax)
>>> plt.plot(p2[0], label='order=2', alpha=0.8)
>>> plt.xticks([])
>>> plt.ylabel('Quadratic')
>>> plt.subplot(4,1,4, sharex=ax)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max),
...                          y_axis='log')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.tonnetz

`librosa.feature.tonnetz`(*y=None*, *sr=22050*, *chroma=None*)[source]

Computes the tonal centroid features (tonnetz), following the method of [1].

[1] Harte, C., Sandler, M., & Gasser, M. (2006). “Detecting Harmonic Change in Musical Audio.” In Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia (pp. 21-26). Santa Barbara, CA, USA: ACM Press. doi:10.1145/1178723.1178727.

Parameters: **y** : np.ndarray [shape=(n,)] or None

Audio time series.

sr : number > 0 [scalar]

sampling rate of *y*

chroma : np.ndarray [shape=(n_chroma, t)] or None

Normalized energy for each chroma bin at each frame.

If *None*, a cqt chromagram is performed.

Returns: **tonnetz** : np.ndarray [shape(6, t)]

Tonal centroid features for each frame.

Tonnetz dimensions:

- 0: Fifth x-axis
- 1: Fifth y-axis
- 2: Minor x-axis
- 3: Minor y-axis
- 4: Major x-axis
- 5: Major y-axis

See also

`chroma_cqt`
Compute a chromagram from a constant-Q transform.

`chroma_stft`
Compute a chromagram from an STFT spectrogram or waveform.

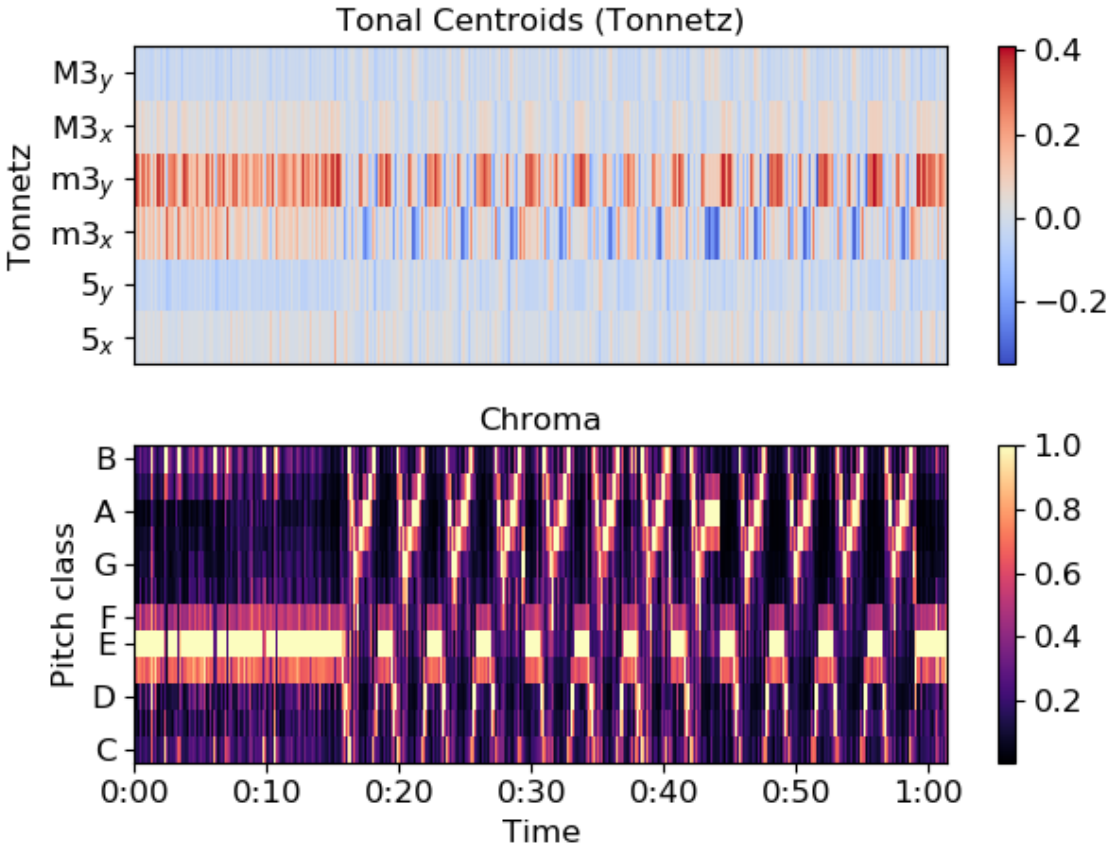
Examples

Compute tonnetz features from the harmonic component of a song

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y = librosa.effects.harmonic(y)
>>> tonnetz = librosa.feature.tonnetz(y=y, sr=sr)
>>> tonnetz
array([[-0.073, -0.053, ..., -0.054, -0.073],
       [ 0.001,  0.001, ..., -0.054, -0.062],
       ...,
       [ 0.039,  0.034, ...,  0.044,  0.064],
       [ 0.005,  0.002, ...,  0.011,  0.017]])
```

Compare the tonnetz features to `chroma_cqt`

```
>>> import matplotlib.pyplot as plt
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(tonnetz, y_axis='tonnetz')
>>> plt.colorbar()
>>> plt.title('Tonal Centroids (Tonnetz)')
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(librosa.feature.chroma_cqt(y, sr=sr),
...                          y_axis='chroma', x_axis='time')
>>> plt.colorbar()
>>> plt.title('Chroma')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.zero_crossing_rate

`librosa.feature.zero_crossing_rate`(*y*, *frame_length=2048*, *hop_length=512*, *center=True*, ***kwargs*)[source]

Compute the zero-crossing rate of an audio time series.

Parameters: **y** : np.ndarray [shape=(n,)]

Audio time series

frame_length : int > 0

Length of the frame over which to compute zero crossing rates

hop_length : int > 0

Number of samples to advance for each frame

center : bool

If *True*, frames are centered by padding the edges of *y*. This is similar to the padding in `librosa.core.stft`, but uses edge-value copies instead of reflection.

kwargs : additional keyword arguments

See `librosa.core.zero_crossings`

Note

By default, the *pad* parameter is set to *False*, which differs from the default specified by `librosa.core.zero_crossings`.

Returns: **zcr** : np.ndarray [shape=(1, t)]

zcr[0, i] is the fraction of zero crossings in the *i* th frame

See also

`librosa.core.zero_crossings`
Compute zero-crossings in a time-series

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> librosa.feature.zero_crossing_rate(y)
array([[ 0.134,  0.139, ...,  0.387,  0.322]])
```

librosa.feature.tempogram

`librosa.feature.tempogram`(*y=None, sr=22050, onset_envelope=None, hop_length=512, win_length=384, center=True, window='hann', norm=inf*)[source]

Compute the tempogram: local autocorrelation of the onset strength envelope. [1]

[1] Grosche, Peter, Meinard Müller, and Frank Kurth. “Cyclic tempogram - A mid-level tempo representation for music signals.” ICASSP, 2010.

Parameters: **y** : np.ndarray [shape=(n,)] or None

Audio time series.

sr : number > 0 [scalar]

sampling rate of *y*

onset_envelope : np.ndarray [shape=(n,) or (m, n)] or None

Optional pre-computed onset strength envelope as provided by *onset.onset_strength*.

If multi-dimensional, tempograms are computed independently for each band (first dimension).

hop_length : int > 0

number of audio samples between successive onset measurements

win_length : int > 0

length of the onset autocorrelation window (in frames/onset measurements) The default settings (384) corresponds to *384 * hop_length / sr ~= 8.9s*.

center : bool

If *True*, onset autocorrelation windows are centered. If *False*, windows are left-aligned.

window : string, function, number, tuple, or np.ndarray [shape=(win_length,)]

A window specification as in *core.stft*.

norm : {np.inf, -np.inf, 0, float > 0, None}

Normalization mode. Set to *None* to disable normalization.

Returns: **tempogram** : np.ndarray [shape=(win_length, n) or (m, win_length, n)]

Localized autocorrelation of the onset strength envelope.

If given multi-band input (*onset_envelope.shape==(m,n)*) then *tempogram[i]* is the tempogram of *onset_envelope[i]*.

Raises: ParameterError

if neither *y* nor *onset_envelope* are provided

if *win_length < 1*

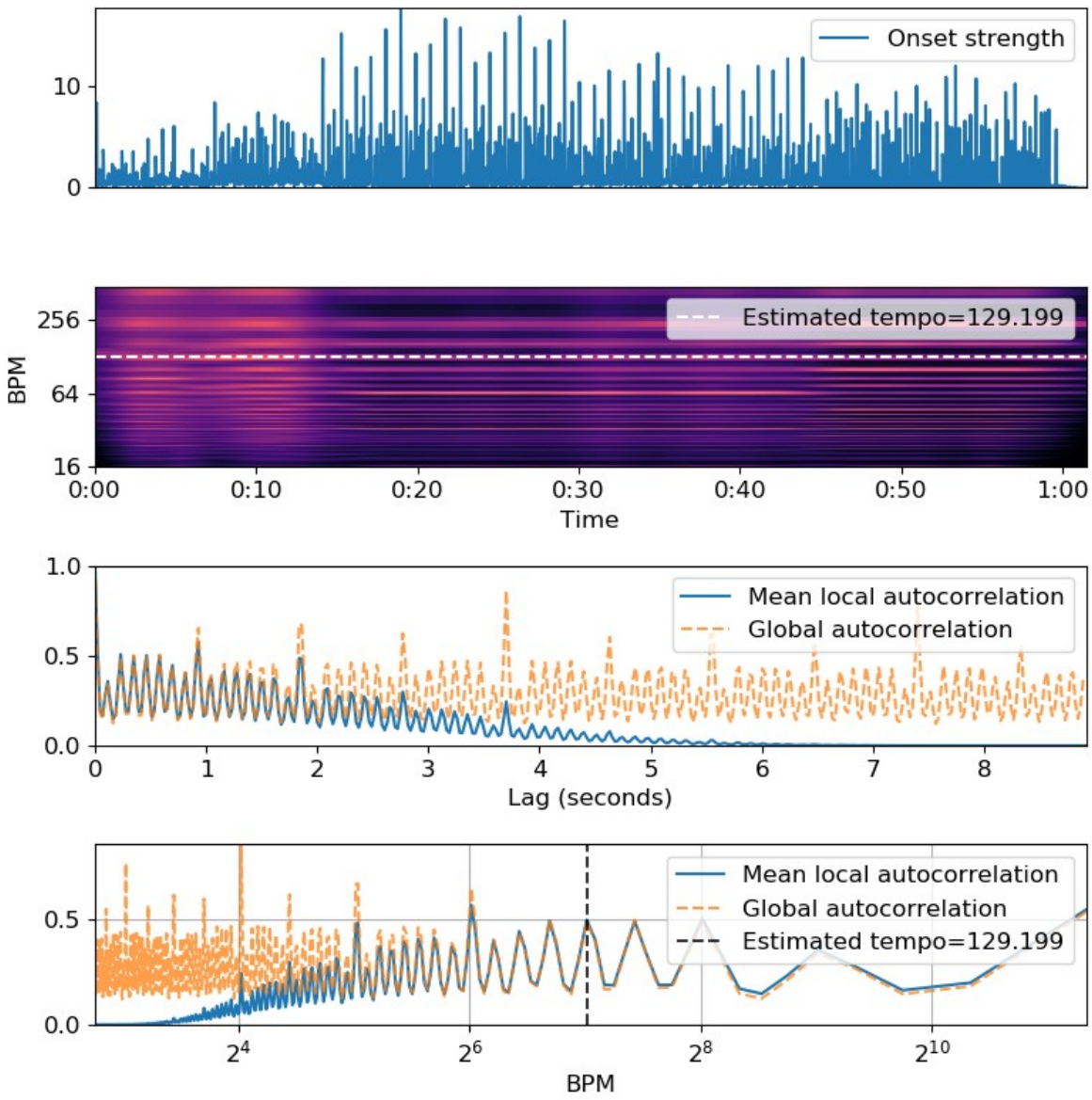
See also

`fourier_tempogram`
`librosa.onset.onset_strength`
`librosa.util.normalize`
`librosa.core.stft`

Examples

```
>>> # Compute local onset autocorrelation
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> hop_length = 512
>>> oenv = librosa.onset.onset_strength(y=y, sr=sr, hop_length=hop_length)
>>> tempogram = librosa.feature.tempogram(onset_envelope=oenv, sr=sr,
...                                       hop_length=hop_length)
>>> # Compute global onset autocorrelation
>>> ac_global = librosa.autocorrelate(oenv, max_size=tempogram.shape[0])
>>> ac_global = librosa.util.normalize(ac_global)
>>> # Estimate the global tempo for display purposes
>>> tempo = librosa.beat.tempo(onset_envelope=oenv, sr=sr,
```

```
...                             hop_length=hop_length)[0]

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 8))
>>> plt.subplot(4, 1, 1)
>>> plt.plot(oenv, label='Onset strength')
>>> plt.xticks([])
>>> plt.legend(frameon=True)
>>> plt.axis('tight')
>>> plt.subplot(4, 1, 2)
>>> # We'll truncate the display to a narrower range of tempi
>>> librosa.display.specshow(tempogram, sr=sr, hop_length=hop_length,
>>>                          x_axis='time', y_axis='tempo')
>>> plt.axhline(tempo, color='w', linestyle='--', alpha=1,
...             label='Estimated tempo={:g}'.format(tempo))
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.subplot(4, 1, 3)
>>> x = np.linspace(0, tempogram.shape[0] * float(hop_length) / sr,
...                 num=tempogram.shape[0])
>>> plt.plot(x, np.mean(tempogram, axis=1), label='Mean local
autocorrelation')
>>> plt.plot(x, ac_global, '--', alpha=0.75, label='Global
autocorrelation')
>>> plt.xlabel('Lag (seconds)')
>>> plt.axis('tight')
>>> plt.legend(frameon=True)
>>> plt.subplot(4,1,4)
>>> # We can also plot on a BPM axis
>>> freqs = librosa.tempo_frequencies(tempogram.shape[0],
hop_length=hop_length, sr=sr)
>>> plt.semilogx(freqs[1:], np.mean(tempogram[1:], axis=1),
...              label='Mean local autocorrelation', basex=2)
>>> plt.semilogx(freqs[1:], ac_global[1:], '--', alpha=0.75,
...              label='Global autocorrelation', basex=2)
>>> plt.axvline(tempo, color='black', linestyle='--', alpha=.8,
...             label='Estimated tempo={:g}'.format(tempo))
>>> plt.legend(frameon=True)
>>> plt.xlabel('BPM')
>>> plt.axis('tight')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.fourier_tempogram

`librosa.feature.fourier_tempogram`(*y=None, sr=22050, onset_envelope=None, hop_length=512, win_length=384, center=True, window='hann'*)[source]

Compute the Fourier tempogram: the short-time Fourier transform of the onset strength envelope. [1]

[1] Grosche, Peter, Meinard Müller, and Frank Kurth. “Cyclic tempogram - A mid-level tempo representation for music signals.” ICASSP, 2010.

Parameters: **y** : np.ndarray [shape=(n,)] or None

Audio time series.

sr : number > 0 [scalar]

sampling rate of *y*

onset_envelope : np.ndarray [shape=(n,)] or None

Optional pre-computed onset strength envelope as provided by *onset.onset_strength*.

hop_length : int > 0

number of audio samples between successive onset measurements

win_length : int > 0

length of the onset window (in frames/onset measurements) The default settings (384) corresponds to *384 * hop_length / sr ~= 8.9s*.

center : bool

If *True*, onset windows are centered. If *False*, windows are left-aligned.

window : string, function, number, tuple, or np.ndarray [shape=(win_length,)]

A window specification as in *core.stft*.

Returns: **tempogram** : np.ndarray [shape=(win_length // 2 + 1, n)]

Complex short-time Fourier transform of the onset envelope.

Raises: ParameterError

if neither *y* nor *onset_envelope* are provided

if *win_length < 1*

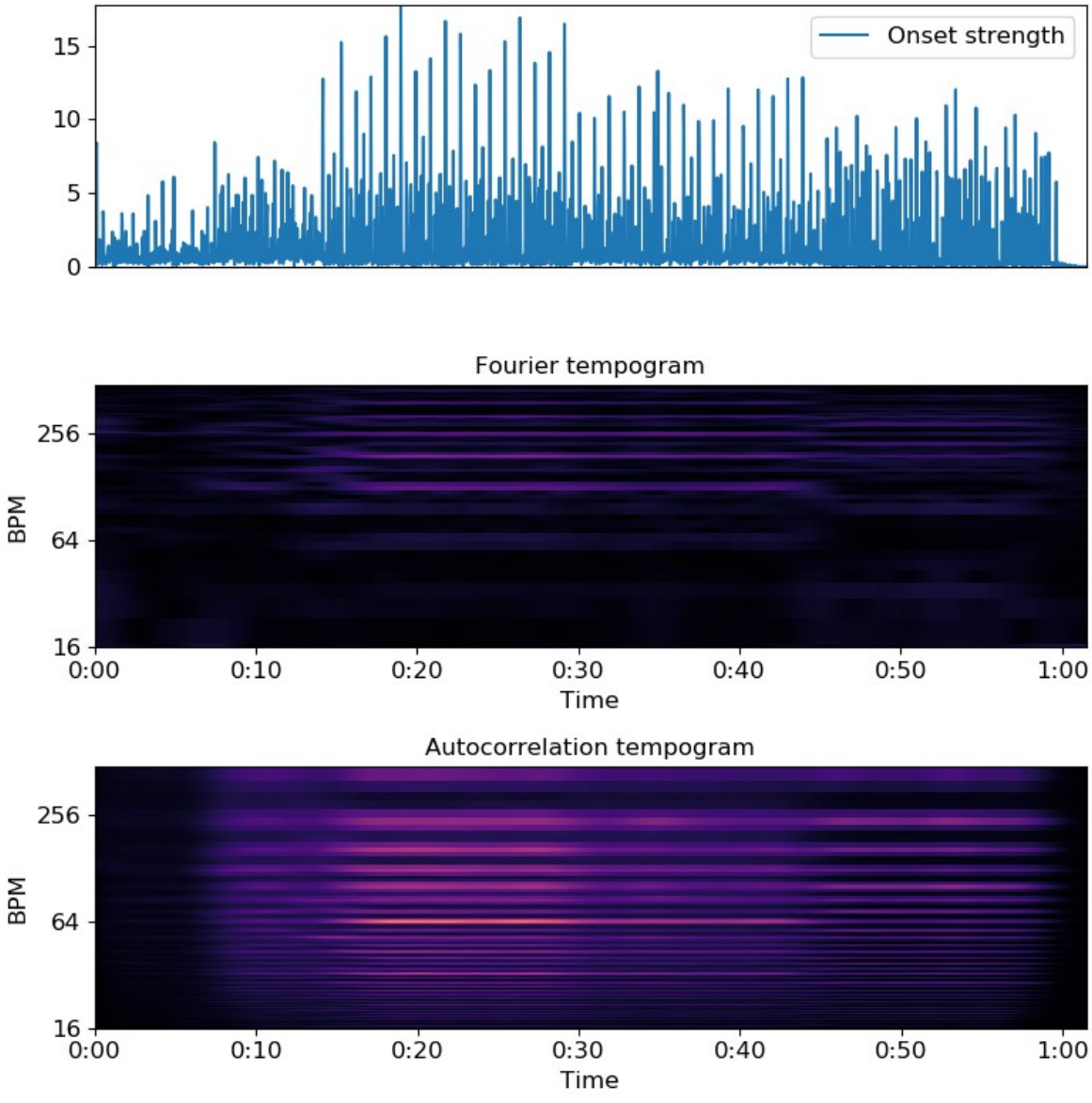
See also

`tempogram`
`librosa.onset.onset_strength`
`librosa.util.normalize`
`librosa.core.stft`

Examples

```
>>> # Compute local onset autocorrelation
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> hop_length = 512
>>> oenv = librosa.onset.onset_strength(y=y, sr=sr, hop_length=hop_length)
>>> tempogram = librosa.feature.fourier_tempogram(onset_envelope=oenv,
sr=sr,
...                                                hop_length=hop_length)
>>> # Compute the auto-correlation tempogram, unnormalized to make
comparison easier
>>> ac_tempogram = librosa.feature.tempogram(onset_envelope=oenv, sr=sr,
...                                          hop_length=hop_length,
norm=None)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 8))
>>> plt.subplot(3, 1, 1)
>>> plt.plot(oenv, label='Onset strength')
>>> plt.xticks([])
>>> plt.legend(frameon=True)
>>> plt.axis('tight')
>>> plt.subplot(3, 1, 2)
>>> librosa.display.specshow(np.abs(tempogram), sr=sr,
hop_length=hop_length,
>>>                          x_axis='time', y_axis='fourier_tempo',
cmap='magma')
>>> plt.title('Fourier tempogram')
>>> plt.subplot(3, 1, 3)
>>> librosa.display.specshow(ac_tempogram, sr=sr, hop_length=hop_length,
>>>                          x_axis='time', y_axis='tempo', cmap='magma')
>>> plt.title('Autocorrelation tempogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.delta

`librosa.feature.delta`(*data, width=9, order=1, axis=-1, mode='interp', **kwargs*) [source]

Compute delta features: local estimate of the derivative of the input data along the selected axis.

Delta features are computed Savitsky-Golay filtering.

Parameters: **data** : np.ndarray

the input data matrix (eg, spectrogram)

width : int, positive, odd [scalar]

Number of frames over which to compute the delta features. Cannot exceed the length of *data* along the specified axis. If *mode='interp'*, then *width* must be at least *data.shape[axis]*.

order : int > 0 [scalar]

the order of the difference operator. 1 for first derivative, 2 for second, etc.

axis : int [scalar]

the axis along which to compute deltas. Default is -1 (columns).

mode : str, {'interp', 'nearest', 'mirror', 'constant', 'wrap'}

Padding mode for estimating differences at the boundaries.

kwargs : additional keyword arguments

See `scipy.signal.savgol_filter`

Returns: **delta_data** : np.ndarray [shape=(d, t)]

delta matrix of *data* at specified order

See also

`scipy.signal.savgol_filter`

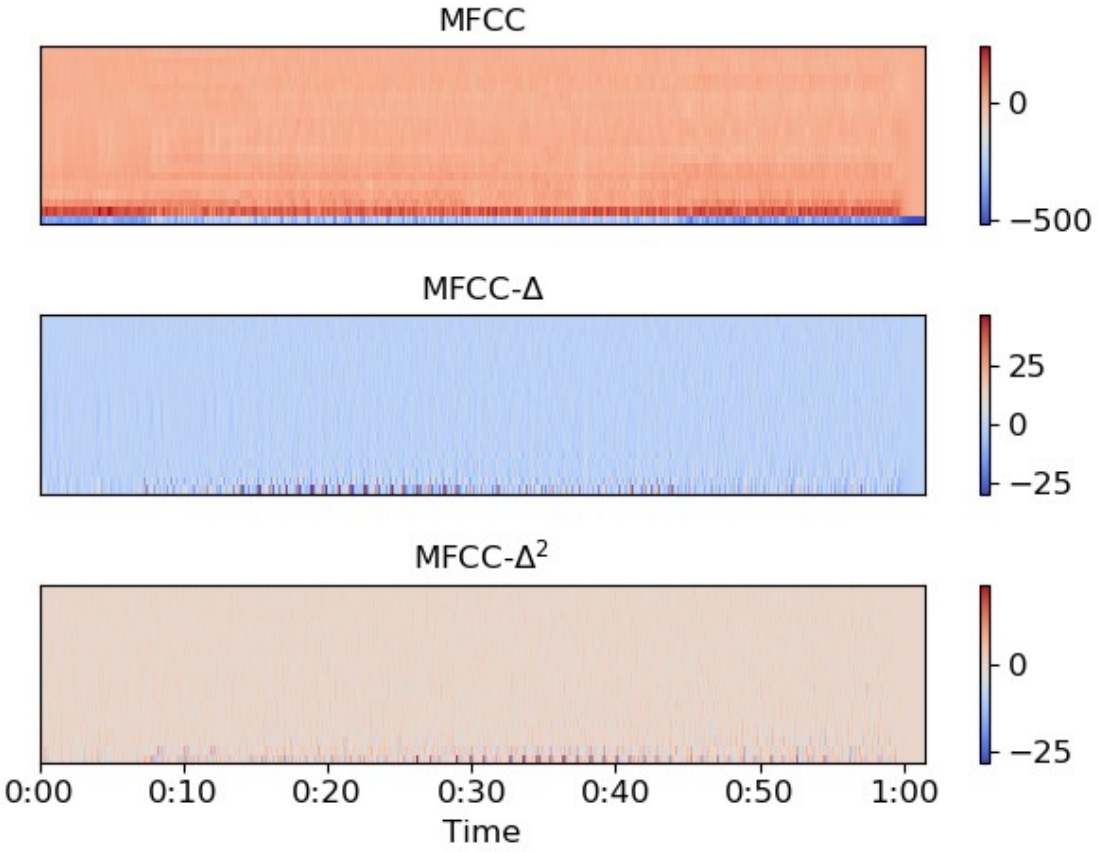
Notes

This function caches at level 40.

Examples

Compute MFCC deltas, delta-deltas

```
    >>> y, sr = librosa.load(librosa.util.example_audio_file())
    >>> mfcc = librosa.feature.mfcc(y=y, sr=sr)
    >>> mfcc_delta = librosa.feature.delta(mfcc)
    >>> mfcc_delta
    array([[  1.666e+01,   1.666e+01, ...,   1.869e-15,   1.869e-15],
           [  1.784e+01,   1.784e+01, ...,   6.085e-31,   6.085e-31],
           ...,
           [  7.262e-01,   7.262e-01, ...,   9.259e-31,   9.259e-31],
           [  6.578e-01,   6.578e-01, ...,   7.597e-31,   7.597e-31]])

    >>> mfcc_delta2 = librosa.feature.delta(mfcc, order=2)
    >>> mfcc_delta2
    array([[ -1.703e+01,  -1.703e+01, ...,   3.834e-14,   3.834e-14],
           [ -1.108e+01,  -1.108e+01, ...,  -1.068e-30,  -1.068e-30],
           ...,
           [  4.075e-01,   4.075e-01, ...,  -1.565e-30,  -1.565e-30],
           [  1.676e-01,   1.676e-01, ...,  -2.104e-30,  -2.104e-30]])

    >>> import matplotlib.pyplot as plt
    >>> plt.subplot(3, 1, 1)
    >>> librosa.display.specshow(mfcc)
    >>> plt.title('MFCC')
    >>> plt.colorbar()
    >>> plt.subplot(3, 1, 2)
    >>> librosa.display.specshow(mfcc_delta)
    >>> plt.title(r'MFCC-$\Delta$')
    >>> plt.colorbar()
    >>> plt.subplot(3, 1, 3)
    >>> librosa.display.specshow(mfcc_delta2, x_axis='time')
    >>> plt.title(r'MFCC-$\Delta^2$')
    >>> plt.colorbar()
    >>> plt.tight_layout()

>>> plt.show()
```



librosa.feature.stack_memory

`librosa.feature.stack_memory`(*data*, *n_steps=2*, *delay=1*, ***kwargs*)[source]

Short-term history embedding: vertically concatenate a data vector or matrix with delayed copies of itself.

Each column *data[:, i]* is mapped to:

```
data[:, i] ->  [data[:, i],
                data[:, i - delay],
                ...
                data[:, i - (n_steps-1)*delay]]
```

For columns *i < (n_steps - 1) * delay* , the data will be padded. By default, the data is padded with zeros, but this behavior can be overridden by supplying additional keyword arguments which are passed to *np.pad()*.

Parameters: **data** : np.ndarray [shape=(t,) or (d, t)]

Input data matrix. If *data* is a vector (*data.ndim == 1*), it will be interpreted as a row matrix and reshaped to *(1, t)*.

n_steps : int > 0 [scalar]

embedding dimension, the number of steps back in time to stack

delay : int != 0 [scalar]

the number of columns to step.

Positive values embed from the past (previous columns).

Negative values embed from the future (subsequent columns).

kwargs : additional keyword arguments

Additional arguments to pass to *np.pad*.

Returns: **data_history** : np.ndarray [shape=(m * d, t)]

data augmented with lagged copies of itself, where *m == n_steps - 1*.

Notes

This function caches at level 40.

Examples

Keep two steps (current and previous)

```
>>> data = np.arange(-3, 3)
>>> librosa.feature.stack_memory(data)
array([[-3, -2, -1,  0,  1,  2],
       [ 0, -3, -2, -1,  0,  1]])
```

Or three steps

```
>>> librosa.feature.stack_memory(data, n_steps=3)
array([[-3, -2, -1,  0,  1,  2],
       [ 0, -3, -2, -1,  0,  1],
       [ 0,  0, -3, -2, -1,  0]])
```

Use reflection padding instead of zero-padding

```
>>> librosa.feature.stack_memory(data, n_steps=3, mode='reflect')
array([[-3, -2, -1,  0,  1,  2],
       [-2, -3, -2, -1,  0,  1],
       [-1, -2, -3, -2, -1,  0]])
```

Or pad with edge-values, and delay by 2

```
>>> librosa.feature.stack_memory(data, n_steps=3, delay=2, mode='edge')
array([[-3, -2, -1,  0,  1,  2],
       [-3, -3, -3, -2, -1,  0],
       [-3, -3, -3, -3, -3, -2]])
```

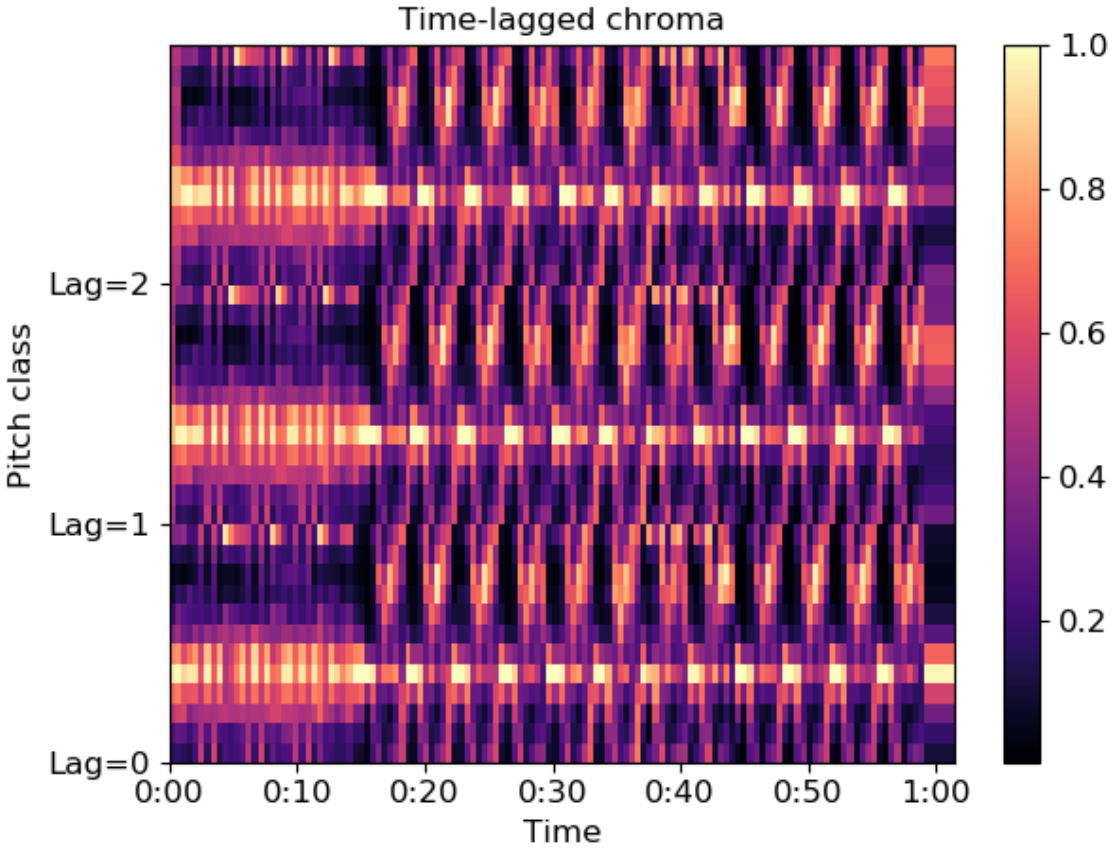
Stack time-lagged beat-synchronous chroma edge padding

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> chroma = librosa.feature.chroma_stft(y=y, sr=sr)
>>> tempo, beats = librosa.beat.beat_track(y=y, sr=sr, hop_length=512)
>>> beats = librosa.util.fix_frames(beats, x_min=0, x_max=chroma.shape[1])
```

```
>>> chroma_sync = librosa.util.sync(chroma, beats)
>>> chroma_lag = librosa.feature.stack_memory(chroma_sync, n_steps=3,
...                                           mode='edge')
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> beat_times = librosa.frames_to_time(beats, sr=sr, hop_length=512)
>>> librosa.display.specshow(chroma_lag, y_axis='chroma', x_axis='time',
...                          x_coords=beat_times)
>>> plt.yticks([0, 12, 24], ['Lag=0', 'Lag=1', 'Lag=2'])
>>> plt.title('Time-lagged chroma')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.inverse.mel_to_stft

`librosa.feature.inverse.mel_to_stft`(*M*, *sr=22050*, *n_fft=2048*, *power=2.0*, ***kwargs*)[source]

Approximate STFT magnitude from a Mel power spectrogram.

Parameters: **M** : np.ndarray [shape=(n_mels, n), non-negative]

The spectrogram as produced by *feature.melspectrogram*

sr : number > 0 [scalar]

sampling rate of the underlying signal

n_fft : int > 0 [scalar]

number of FFT components in the resulting STFT

power : float > 0 [scalar]

Exponent for the magnitude melspectrogram

kwargs : additional keyword arguments

Mel filter bank parameters. See `librosa.filters.mel` for details

Returns: **S** : np.ndarray [shape=(n_fft, t), non-negative]

An approximate linear magnitude spectrogram

See also

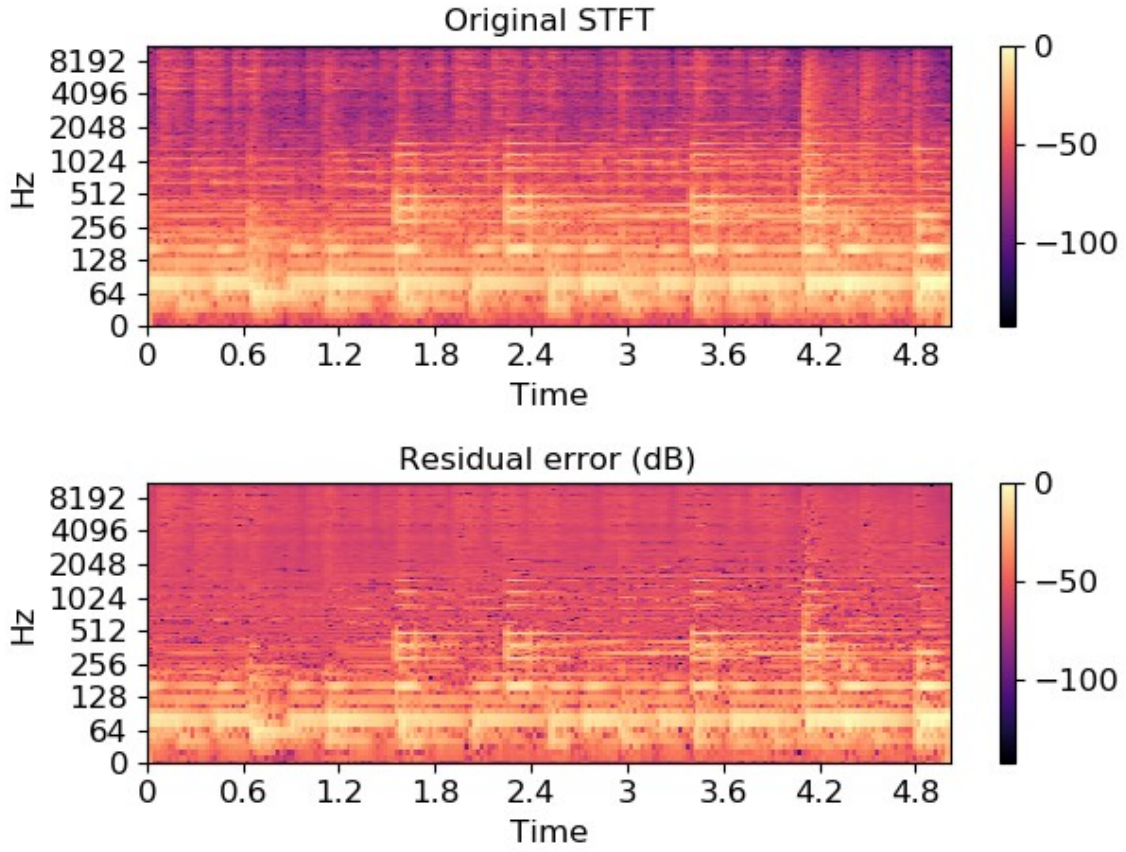
`feature.melspectrogram`
`core.stft`
`filters.mel`
`util.nnls`

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=5,
offset=10)
>>> S = np.abs(librosa.stft(y))
>>> mel_spec = librosa.feature.melspectrogram(S=S, sr=sr)
>>> S_inv = librosa.feature.inverse.mel_to_stft(mel_spec, sr=sr)
```

Compare the results visually

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(librosa.amplitude_to_db(S, ref=np.max,
top_db=None),
...                          y_axis='log', x_axis='time')
>>> plt.colorbar()
>>> plt.title('Original STFT')
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(librosa.amplitude_to_db(np.abs(S_inv - S),
...                                                  ref=S.max(),
top_db=None),
...                          vmax=0, y_axis='log', x_axis='time',
cmap='magma')
>>> plt.title('Residual error (dB)')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.feature.inverse.mel_to_audio

`librosa.feature.inverse.mel_to_audio`(*M, sr=22050, n_fft=2048, hop_length=512, win_length=None, window='hann', center=True, pad_mode='reflect', power=2.0, n_iter=32, length=None, dtype=<class 'numpy.float32'>, **kwargs*)[source]

Invert a mel power spectrogram to audio using Griffin-Lim.

This is primarily a convenience wrapper for:

```
>>> S = librosa.feature.inverse.mel_to_stft(M)
>>> y = librosa.griffinlim(S)
```

Parameters: **M** : np.ndarray [shape=(n_mels, n), non-negative]

The spectrogram as produced by *feature.melspectrogram*

sr : number > 0 [scalar]

sampling rate of the underlying signal

n_fft : int > 0 [scalar]

number of FFT components in the resulting STFT

hop_length : None or int > 0

The hop length of the STFT. If not provided, it will default to *n_fft // 4*

win_length : None or int > 0

The window length of the STFT. By default, it will equal *n_fft*

window : string, tuple, number, function, or np.ndarray [shape=(n_fft,)]

A window specification as supported by *stft* or *istft*

center : boolean

If *True*, the STFT is assumed to use centered frames. If *False*, the STFT is assumed to use left-aligned frames.

pad_mode : string

If *center=True*, the padding mode to use at the edges of the signal. By default, STFT uses reflection padding.

power : float > 0 [scalar]

Exponent for the magnitude melspectrogram

n_iter : int > 0

The number of iterations for Griffin-Lim

length : None or int > 0

If provided, the output *y* is zero-padded or clipped to exactly *length* samples.

dtype : np.dtype

Real numeric type for the time-domain signal. Default is 32-bit float.

kwargs : additional keyword arguments

Mel filter bank parameters

Returns: **y** : np.ndarray [shape(n,)]

time-domain signal reconstructed from *M*

See also

`core.griffinlim`
`feature.melspectrogram`

```
filters.mel
feature.inverse.mel_to_stft
```

librosa.feature.inverse.mfcc_to_mel

`librosa.feature.inverse.mfcc_to_mel`(*mfcc, n_mels=128, dct_type=2, norm='ortho', ref=1.0*)[source]

Invert Mel-frequency cepstral coefficients to approximate a Mel power spectrogram.

This inversion proceeds in two steps:

1. The inverse DCT is applied to the MFCCs
2. *core.db_to_power* is applied to map the dB-scaled result to a power spectrogram

Parameters: **mfcc** : np.ndarray [shape=(n_mfcc, n)]

The Mel-frequency cepstral coefficients

n_mels : int > 0

The number of Mel frequencies

dct_type : None or {1, 2, 3}

Discrete cosine transform (DCT) type By default, DCT type-2 is used.

norm : None or ‘ortho’

If *dct_type* is *2 or 3*, setting *norm=’ortho’* uses an orthonormal DCT basis.

Normalization is not supported for *dct_type=1*.

ref : number or callable

Reference power for (inverse) decibel calculation

Returns: **M** : np.ndarray [shape=(n_mels, n)]

An approximate Mel power spectrum recovered from *mfcc*

See also

```
mfcc
melspectrogram
scipy.fftpack.dct
```

librosa.feature.inverse.mfcc_to_audio

`librosa.feature.inverse.mfcc_to_audio`(*mfcc*, *n_mels=128*, *dct_type=2*, *norm='ortho'*, *ref=1.0*, ***kwargs*)[source]

Convert Mel-frequency cepstral coefficients to a time-domain audio signal

This function is primarily a convenience wrapper for the following steps:

1. Convert mfcc to Mel power spectrum (`mfcc_to_mel`)
2. Convert Mel power spectrum to time-domain audio (`mel_to_audio`)

Parameters:

mfcc : np.ndarray [shape=(n_mfcc, n)]

The Mel-frequency cepstral coefficients

n_mels : int > 0

The number of Mel frequencies

dct_type : None or {1, 2, 3}

Discrete cosine transform (DCT) type By default, DCT type-2 is used.

norm : None or ‘ortho’

If *dct_type* is *2 or 3*, setting *norm=’ortho’* uses an orthonormal DCT basis.

Normalization is not supported for *dct_type=1*.

ref : number or callable

Reference power for (inverse) decibel calculation

kwargs : additional keyword arguments

Parameters to pass through to `mel_to_audio`

Returns:

y : np.ndarray [shape=(n)]

A time-domain signal reconstructed from *mfcc*

See also

`mfcc_to_mel`
`mel_to_audio`
`feature.mfcc`
`core.griffinlim`

scipy.fftpack.dct

Onset detection

onset_detect([y, sr, onset_envelope, …])	Basic onset detector.
onset_backtrack(events, energy)	Backtrack detected onset events to the nearest preceding local minimum of an energy function.
onset_strength([y, sr, S, lag, max_size, …])	Compute a spectral flux onset strength envelope.
onset_strength_multi([y, sr, S, n_fft, …])	Compute a spectral flux onset strength envelope across multiple channels.

librosa.onset.onset_detect

librosa.onset.onset_detect(*y=None, sr=22050, onset_envelope=None, hop_length=512, backtrack=False, energy=None, units='frames', **kwargs*)[source]

Basic onset detector. Locate note onset events by picking peaks in an onset strength envelope.

The *peak_pick* parameters were chosen by large-scale hyper-parameter optimization over the dataset provided by [1].

[1] https://github.com/CPJKU/onset_db

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

onset_envelope : np.ndarray [shape=(m,)]

(optional) pre-computed onset strength envelope

hop_length : int > 0 [scalar]

hop length (in samples)

units : {‘frames’, ‘samples’, ‘time’}

The units to encode detected onset events in. By default, ‘frames’ are used.

backtrack : bool

If *True*, detected onset events are backtracked to the nearest preceding minimum of *energy*.

This is primarily useful when using onsets as slice points for segmentation.

energy : np.ndarray [shape=(m,)] (optional)

An energy function to use for backtracking detected onset events. If none is provided, then *onset_envelope* is used.

kwargs : additional keyword arguments

Additional parameters for peak picking.

See `librosa.util.peak_pick` for details.

Returns: **onsets** : np.ndarray [shape=(n_onsets,)]

estimated positions of detected onsets, in whichever units are specified. By default, frame indices.

Note

If no onset strength could be detected, onset_detect returns an empty list.

Raises: ParameterError

if neither *y* nor *onsets* are provided

or if *units* is not one of ‘frames’, ‘samples’, or ‘time’

See also

`onset_strength`
compute onset strength per-frame

`onset_backtrack`
backtracking onset events

`librosa.util.peak_pick`
pick peaks from a time series

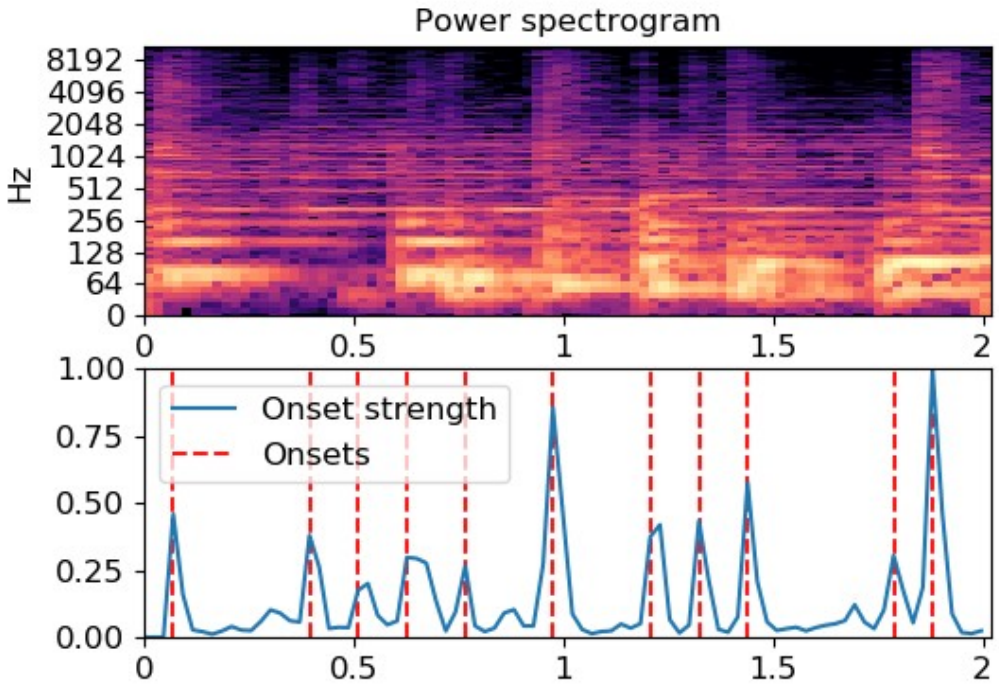
Examples

Get onset times from a signal

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=30, duration=2.0)
>>> onset_frames = librosa.onset.onset_detect(y=y, sr=sr)
>>> librosa.frames_to_time(onset_frames, sr=sr)
array([ 0.07 ,  0.395,  0.511,  0.627,  0.766,  0.975,
        1.207,  1.324,  1.44 ,  1.788,  1.881])
```

Or use a pre-computed onset envelope

```
>>> o_env = librosa.onset.onset_strength(y, sr=sr)
>>> times = librosa.times_like(o_env, sr=sr)
>>> onset_frames = librosa.onset.onset_detect(onset_envelope=o_env, sr=sr)

>>> import matplotlib.pyplot as plt
>>> D = np.abs(librosa.stft(y))
>>> plt.figure()
>>> ax1 = plt.subplot(2, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(D, ref=np.max),
...                          x_axis='time', y_axis='log')
>>> plt.title('Power spectrogram')
>>> plt.subplot(2, 1, 2, sharex=ax1)
>>> plt.plot(times, o_env, label='Onset strength')
>>> plt.vlines(times[onset_frames], 0, o_env.max(), color='r', alpha=0.9,
...            linestyle='--', label='Onsets')
>>> plt.axis('tight')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.show()
```



librosa.onset.onset_backtrack

`librosa.onset.onset_backtrack`(*events*, *energy*)[source]

Backtrack detected onset events to the nearest preceding local minimum of an energy function.

This function can be used to roll back the timing of detected onsets from a detected peak amplitude to the preceding minimum.

This is most useful when using onsets to determine slice points for segmentation, as described by [1].

[1] Jehan, Tristan. “Creating music by listening” Doctoral dissertation Massachusetts Institute of Technology, 2005.

Parameters: **events** : np.ndarray, dtype=int

List of onset event frame indices, as computed by `onset_detect`

energy : np.ndarray, shape=(m,)

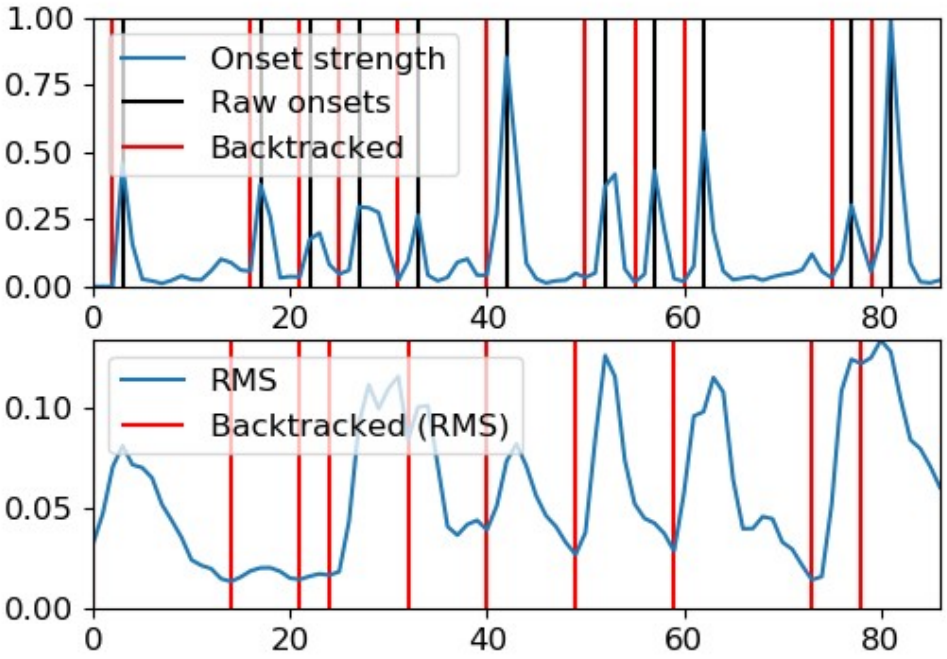
An energy function

Returns: **events_backtracked** : np.ndarray, shape=events.shape

The input events matched to nearest preceding minima of *energy*.

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=30, duration=2.0)
>>> oenv = librosa.onset.onset_strength(y=y, sr=sr)
>>> # Detect events without backtracking
>>> onset_raw = librosa.onset.onset_detect(onset_envelope=oenv,
...                                        backtrack=False)
>>> # Backtrack the events using the onset envelope
>>> onset_bt = librosa.onset.onset_backtrack(onset_raw, oenv)
>>> # Backtrack the events using the RMS values
>>> rms = librosa.feature.rms(S=np.abs(librosa.stft(y=y)))
>>> onset_bt_rms = librosa.onset.onset_backtrack(onset_raw, rms[0])

>>> # Plot the results
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(2,1,1)
>>> plt.plot(oenv, label='Onset strength')
>>> plt.vlines(onset_raw, 0, oenv.max(), label='Raw onsets')
>>> plt.vlines(onset_bt, 0, oenv.max(), label='Backtracked', color='r')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.subplot(2,1,2)
>>> plt.plot(rms[0], label='RMS')
>>> plt.vlines(onset_bt_rms, 0, rms.max(), label='Backtracked (RMS)',
color='r')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.show()
```



librosa.onset.onset_strength

`librosa.onset.onset_strength`(*y=None*, *sr=22050*, *S=None*, *lag=1*, *max_size=1*, *ref=None*, *detrend=False*, *center=True*, *feature=None*, *aggregate=None*, *centering=None*, ***kwargs*)[source]

Compute a spectral flux onset strength envelope.

Onset strength at time *t* is determined by:

mean_f max(0, S[f, t] - ref[f, t - lag])

where *ref* is *S* after local max filtering along the frequency axis [1].

By default, if a time series *y* is provided, S will be the log-power Mel spectrogram.

[1] Böck, Sebastian, and Gerhard Widmer. “Maximum filter vibrato suppression for onset detection.” 16th International Conference on Digital Audio Effects, Maynooth, Ireland. 2013.

Parameters **y** : np.ndarray [shape=(n,)]

:

audio time-series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, m)]

pre-computed (log-power) spectrogram

lag : int > 0

time lag for computing differences

max_size : int > 0

size (in frequency bins) of the local max filter. set to *1* to disable filtering.

ref : None or np.ndarray [shape=(d, m)]

An optional pre-computed reference spectrum, of the same shape as *S*. If not provided, it will be computed from *S*. If provided, it will override any local max filtering governed by *max_size*.

detrend : bool [scalar]

Filter the onset strength to remove the DC component

center : bool [scalar]

Shift the onset function by *n_fft / (2 * hop_length)* frames

feature : function

Function for computing time-series features, eg, scaled spectrograms. By default, uses `librosa.feature.melspectrogram` with *fmax=11025.0*

aggregate : function

Aggregation function to use when combining onsets at different frequency bins.

Default: `np.mean`

kwargs : additional keyword arguments

Additional parameters to *feature()*, if *S* is not provided.

Returns: **onset_envelope** : np.ndarray [shape=(m,)]

vector containing the onset strength envelope

Raises: ParameterError

if neither *(y, sr)* nor *S* are provided

or if *lag* or *max_size* are not positive integers

See also

`onset_detect`
`onset_strength_multi`

Examples

First, load some audio and plot the spectrogram

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=10.0)
>>> D = np.abs(librosa.stft(y))
>>> times = librosa.times_like(D)
>>> plt.figure()
>>> ax1 = plt.subplot(2, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(D, ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('Power spectrogram')
```

Construct a standard onset function

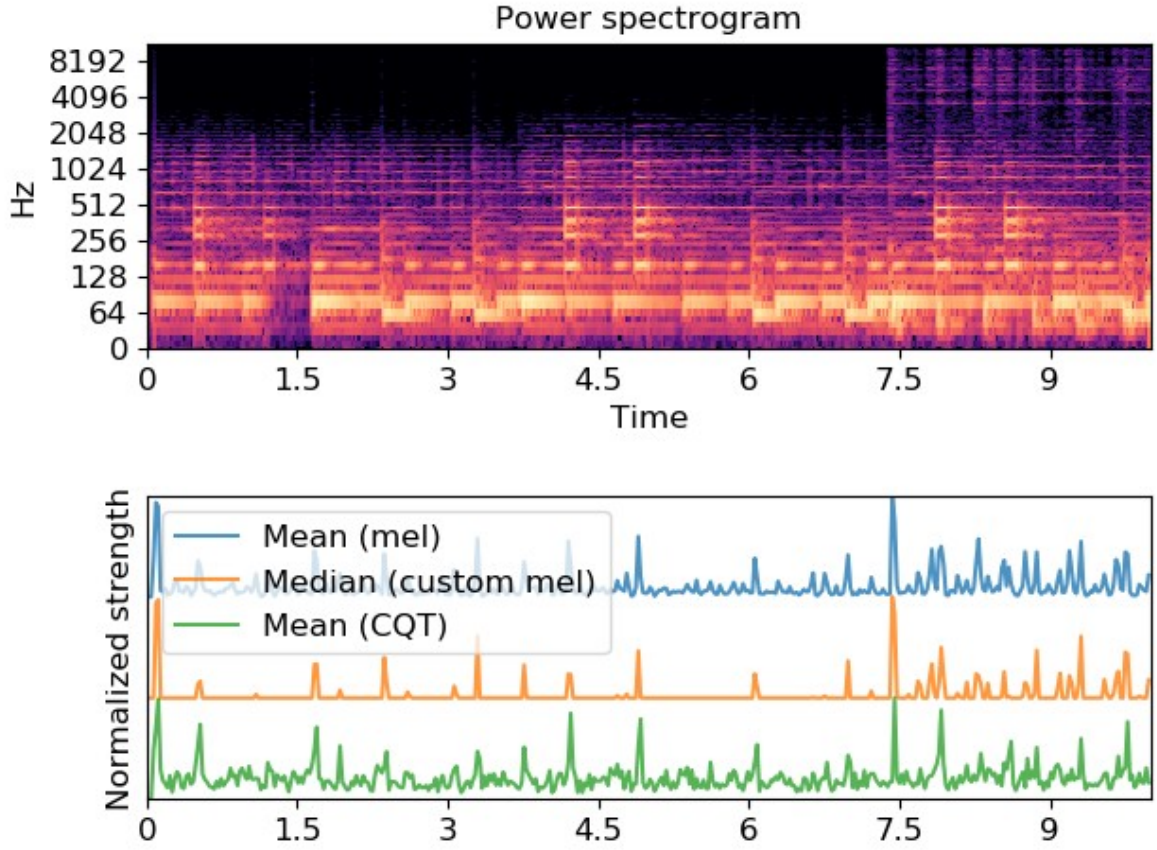
```
>>> onset_env = librosa.onset.onset_strength(y=y, sr=sr)
>>> plt.subplot(2, 1, 2, sharex=ax1)
>>> plt.plot(times, 2 + onset_env / onset_env.max(), alpha=0.8,
...          label='Mean (mel)')
```

Median aggregation, and custom mel options

```
>>> onset_env = librosa.onset.onset_strength(y=y, sr=sr,
...                                          aggregate=np.median,
...                                          fmax=8000, n_mels=256)
>>> plt.plot(times, 1 + onset_env / onset_env.max(), alpha=0.8,
...          label='Median (custom mel)')
```

Constant-Q spectrogram instead of Mel

```
>>> C = np.abs(librosa.cqt(y=y, sr=sr))
>>> onset_env = librosa.onset.onset_strength(sr=sr,
S=librosa.amplitude_to_db(C, ref=np.max))
>>> plt.plot(times, onset_env / onset_env.max(), alpha=0.8,
...          label='Mean (CQT)')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.ylabel('Normalized strength')
>>> plt.yticks([])
>>> plt.axis('tight')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.onset.onset_strength_multi

`librosa.onset.onset_strength_multi`(*y=None, sr=22050, S=None, n_fft=2048, hop_length=512, lag=1, max_size=1, ref=None, detrend=False, center=True, feature=None, aggregate=None, channels=None, **kwargs*)[source]

Compute a spectral flux onset strength envelope across multiple channels.

Onset strength for channel *i* at time *t* is determined by:

mean_{f in channels[i]} max(0, S[f, t+1] - S[f, t])

Parameters :

y : np.ndarray [shape=(n,)]

audio time-series

sr : number > 0 [scalar]

sampling rate of *y*

S : np.ndarray [shape=(d, m)]

pre-computed (log-power) spectrogram

n_fft : int > 0 [scalar]

FFT window size for use in *feature()* if *S* is not provided.

hop_length : int > 0 [scalar]

hop length for use in *feature()* if *S* is not provided.

lag : int > 0

time lag for computing differences

max_size : int > 0

size (in frequency bins) of the local max filter. set to *1* to disable filtering.

ref : None or np.ndarray [shape=(d, m)]

An optional pre-computed reference spectrum, of the same shape as *S*. If not provided, it will be computed from *S*. If provided, it will override any local max filtering governed by *max_size*.

detrend : bool [scalar]

Filter the onset strength to remove the DC component

center : bool [scalar]

Shift the onset function by *n_fft / (2 * hop_length)* frames

feature : function

Function for computing time-series features, eg, scaled spectrograms. By default, uses `librosa.feature.melspectrogram` with *fmax=11025.0*

Must support arguments: *y, sr, n_fft, hop_length*

aggregate : function or False

Aggregation function to use when combining onsets at different frequency bins.

If *False*, then no aggregation is performed.

Default: `np.mean`

channels : list or None

Array of channel boundaries or slice objects. If *None*, then a single channel is generated to span all bands.

kwargs : additional keyword arguments

Additional parameters to *feature()*, if *S* is not provided.

Returns: **onset_envelope** : np.ndarray [shape=(n_channels, m)]

array containing the onset strength envelope for each specified channel

Raises: ParameterError

if neither *(y, sr)* nor *S* are provided

See also

`onset_strength`

Notes

This function caches at level 30.

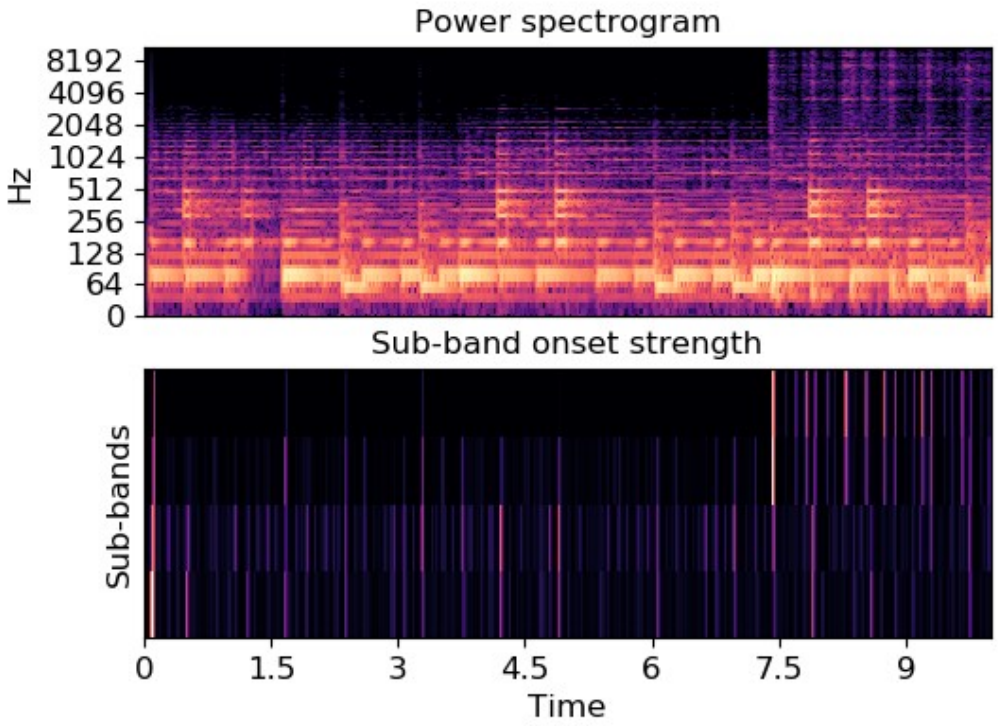
Examples

First, load some audio and plot the spectrogram

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=10.0)
>>> D = np.abs(librosa.stft(y))
>>> plt.figure()
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(D, ref=np.max),
...                          y_axis='log')
>>> plt.title('Power spectrogram')
```

Construct a standard onset function over four sub-bands

```
>>> onset_subbands = librosa.onset.onset_strength_multi(y=y, sr=sr,
...                                                     channels=[0, 32,
64, 96, 128])
>>> plt.subplot(2, 1, 2)
>>> librosa.display.specshow(onset_subbands, x_axis='time')
>>> plt.ylabel('Sub-bands')
>>> plt.title('Sub-band onset strength')
>>> plt.show()
```



Beat and tempo

beat_track([y, sr, onset_envelope, …])	Dynamic programming beat tracker.
plp([y, sr, onset_envelope, hop_length, …])	Predominant local pulse (PLP) estimation.
tempo([y, sr, onset_envelope, hop_length, …])	Estimate the tempo (beats per minute)

librosa.beat.beat_track

librosa.beat.beat_track(*y=None, sr=22050, onset_envelope=None, hop_length=512, start_bpm=120.0, tightness=100, trim=True, bpm=None, prior=None, units='frames'*)[source]

Dynamic programming beat tracker.

Beats are detected in three stages, following the method of [1]:

1. Measure onset strength
2. Estimate tempo from onset correlation
3. Pick peaks in onset strength approximately consistent with estimated tempo

[1] Ellis, Daniel PW. “Beat tracking by dynamic programming.” Journal of New Music Research 36.1 (2007): 51-60. http://labrosa.ee.columbia.edu/projects/beattrack/

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

onset_envelope : np.ndarray [shape=(n,)] or None

(optional) pre-computed onset strength envelope.

hop_length : int > 0 [scalar]

number of audio samples between successive *onset_envelope* values

start_bpm : float > 0 [scalar]

initial guess for the tempo estimator (in beats per minute)

tightness : float [scalar]

tightness of beat distribution around tempo

trim : bool [scalar]

trim leading/trailing beats with weak onsets

bpm : float [scalar]

(optional) If provided, use *bpm* as the tempo instead of estimating it from *onsets*.

prior : scipy.stats.rv_continuous [optional]

An optional prior distribution over tempo. If provided, *start_bpm* will be ignored.

units : {‘frames’, ‘samples’, ‘time’}

The units to encode detected beat events in. By default, ‘frames’ are used.

Returns: **tempo** : float [scalar, non-negative]

estimated global tempo (in beats per minute)

beats : np.ndarray [shape=(m,)]

estimated beat event locations in the specified units (default is frame indices)

Note

If no onset strength could be detected, beat_tracker estimates 0 BPM and returns an empty list.

Raises: ParameterError

if neither *y* nor *onset_envelope* are provided, or if *units* is not one of ‘frames’, ‘samples’, or ‘time’

See also

librosa.onset.onset_strength

Examples

Track beats using time series input

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())

>>> tempo, beats = librosa.beat.beat_track(y=y, sr=sr)
>>> tempo
64.599609375
```

Print the first 20 beat frames

```
>>> beats[:20]
array([ 320,  357,  397,  436,  480,  525,  569,  609,  658,
        698,  737,  777,  817,  857,  896,  936,  976, 1016,
       1055, 1095])
```

Or print them as timestamps

```
>>> librosa.frames_to_time(beats[:20], sr=sr)
array([  7.43 ,   8.29 ,   9.218,  10.124,  11.146,  12.19 ,
        13.212,  14.141,  15.279,  16.208,  17.113,  18.042,
        18.971,  19.9  ,  20.805,  21.734,  22.663,  23.591,
        24.497,  25.426])
```

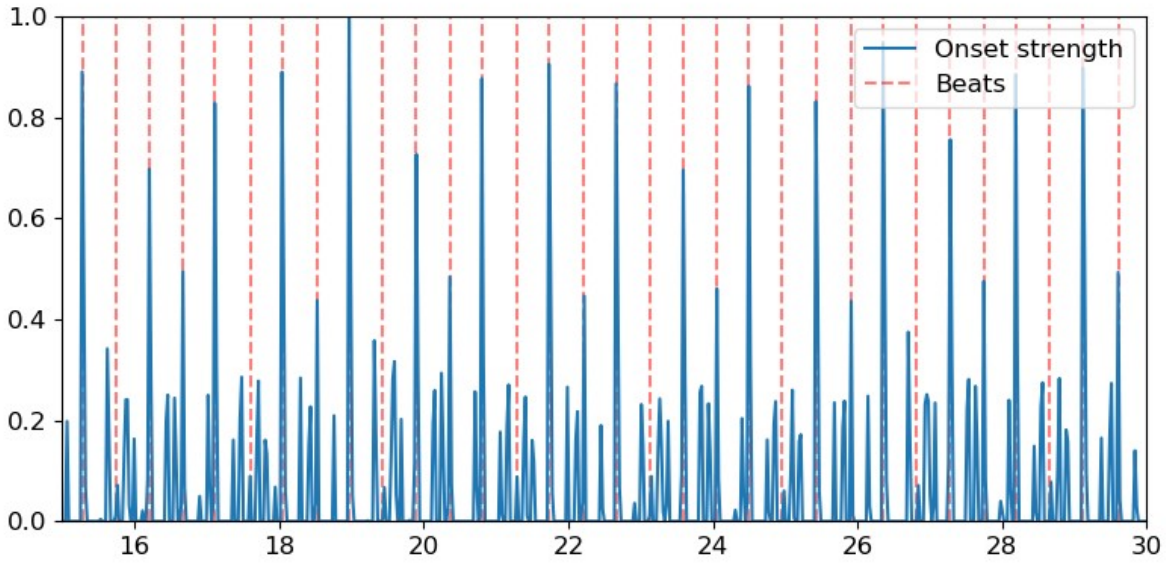
Track beats using a pre-computed onset envelope

```
>>> onset_env = librosa.onset.onset_strength(y, sr=sr,
...                                          aggregate=np.median)
>>> tempo, beats = librosa.beat.beat_track(onset_envelope=onset_env,
...                                        sr=sr)
>>> tempo
```

```
64.599609375
>>> beats[:20]
array([ 320,  357,  397,  436,  480,  525,  569,  609,  658,
        698,  737,  777,  817,  857,  896,  936,  976, 1016,
       1055, 1095])
```

Plot the beat events against the onset strength envelope

```
>>> import matplotlib.pyplot as plt
>>> hop_length = 512
>>> plt.figure(figsize=(8, 4))
>>> times = librosa.times_like(onset_env, sr=sr, hop_length=hop_length)
>>> plt.plot(times, librosa.util.normalize(onset_env),
...          label='Onset strength')
>>> plt.vlines(times[beats], 0, 1, alpha=0.5, color='r',
...            linestyle='--', label='Beats')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> # Limit the plot to a 15-second window
>>> plt.xlim(15, 30)
>>> plt.gca().xaxis.set_major_formatter(librosa.display.TimeFormatter())
>>> plt.tight_layout()
>>> plt.show()
```



librosa.beat.plp

`librosa.beat.plp`(*y=None, sr=22050, onset_envelope=None, hop_length=512, win_length=384, tempo_min=30, tempo_max=300, prior=None*)[source]

Predominant local pulse (PLP) estimation. [1]

The PLP method analyzes the onset strength envelope in the frequency domain to find a locally stable tempo for each frame. These local periodicities are used to synthesize local half-waves, which are combined such that peaks coincide with rhythmically salient frames (e.g. onset events on a musical time grid). The local maxima of the pulse curve can be taken as estimated beat positions.

This method may be preferred over the dynamic programming method of `beat_track` when either the tempo is expected to vary significantly over time. Additionally, since `plp`

does not require the entire signal to make predictions, it may be preferable when beat-tracking long recordings in a streaming setting.

[1] Grosche, P., & Muller, M. (2011). “Extracting predominant local pulse information from music recordings.” IEEE Transactions on Audio, Speech, and Language Processing, 19(6), 1688-1701.

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

sampling rate of *y*

onset_envelope : np.ndarray [shape=(n,)] or None

(optional) pre-computed onset strength envelope

hop_length : int > 0 [scalar]

number of audio samples between successive *onset_envelope* values

win_length : int > 0 [scalar]

number of frames to use for tempogram analysis. By default, 384 frames (at *sr=22050* and *hop_length=512*) corresponds to about 8.9 seconds.

tempo_min, tempo_max : numbers > 0 [scalar], optional

Minimum and maximum permissible tempo values. *tempo_max* must be at least *tempo_min*.

Set either (or both) to *None* to disable this constraint.

prior : scipy.stats.rv_continuous [optional]

A prior distribution over tempo (in beats per minute). By default, a uniform prior over *[tempo_min, tempo_max]* is used.

Returns: **pulse** : np.ndarray, shape=[(n,)]

The estimated pulse curve. Maxima correspond to rhythmically salient points of time.

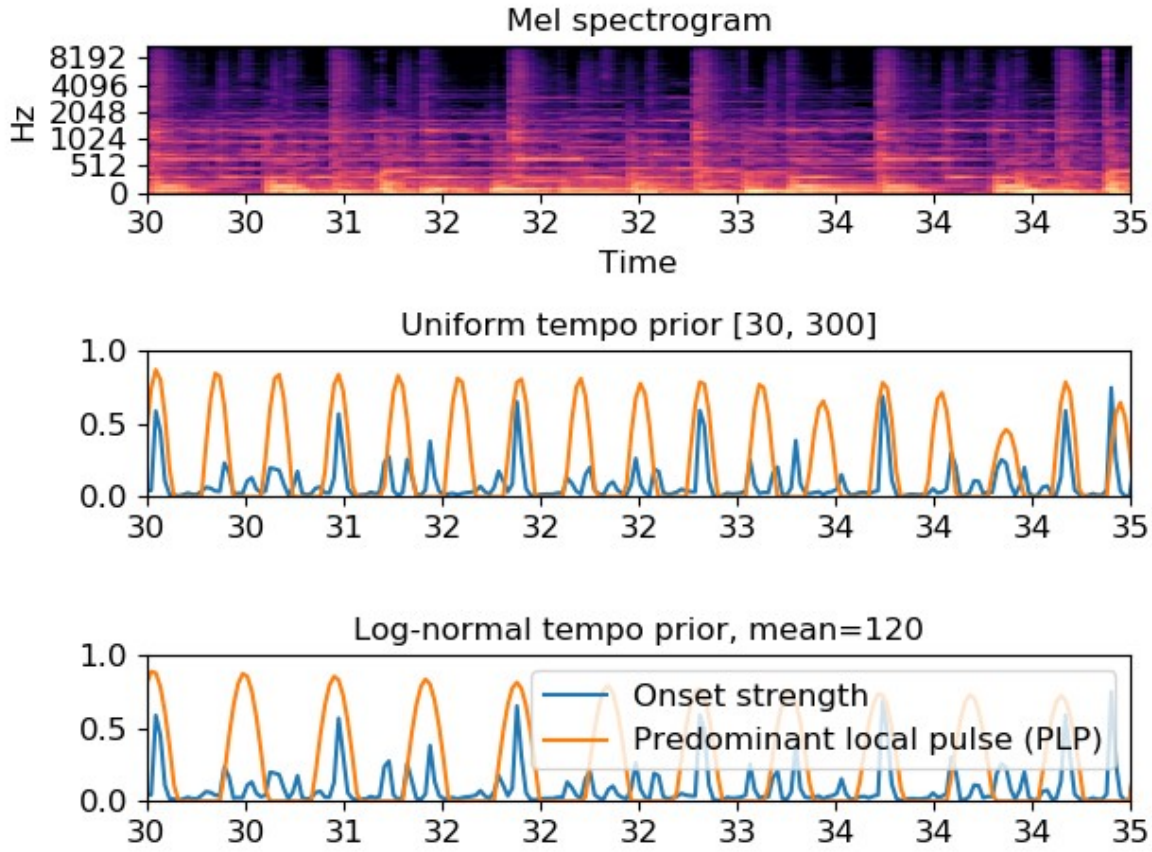
See also

`beat_track`
`librosa.onset.onset_strength`
`librosa.feature.fourier_tempogram`

Examples

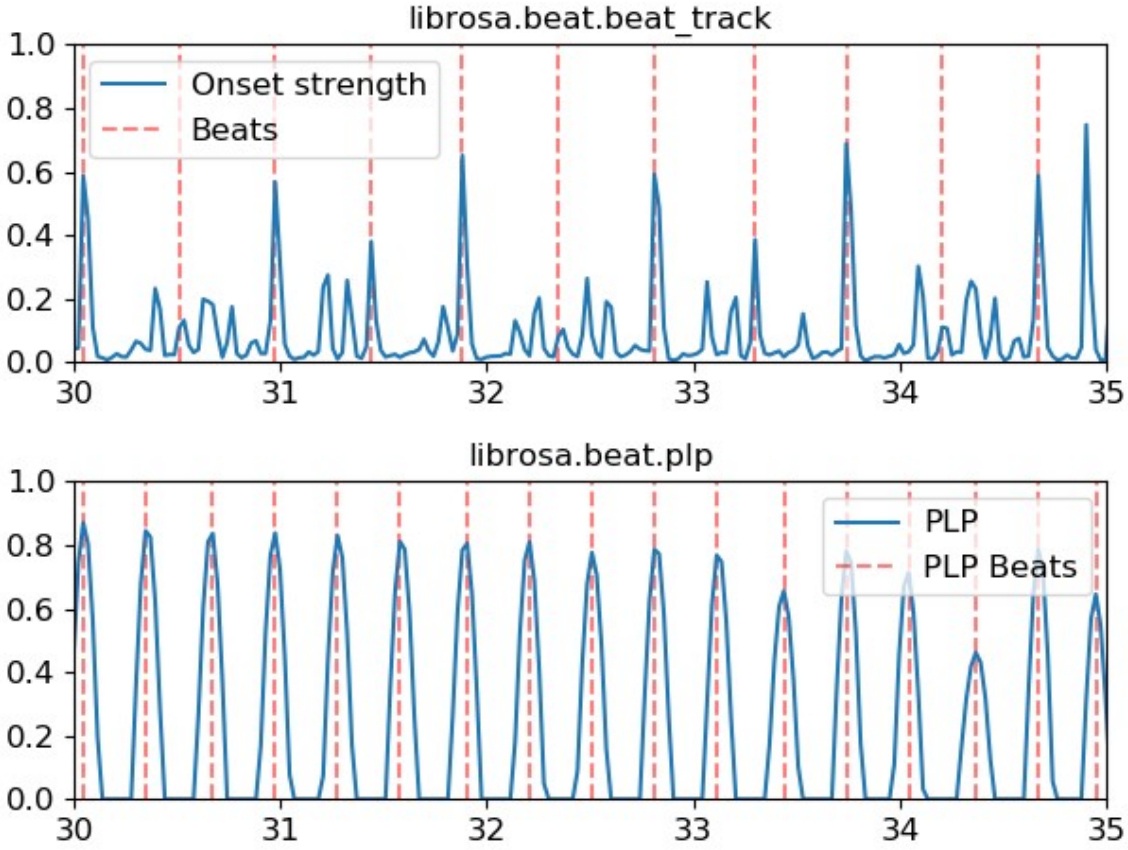
Visualize the PLP compared to an onset strength envelope. Both are normalized here to make comparison easier.

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> onset_env = librosa.onset.onset_strength(y=y, sr=sr)
>>> pulse = librosa.beat.plp(onset_envelope=onset_env, sr=sr)
>>> # Or compute pulse with an alternate prior, like log-normal
>>> import scipy.stats
>>> prior = scipy.stats.lognorm(loc=np.log(120), scale=120, s=1)
>>> pulse_lognorm = librosa.beat.plp(onset_envelope=onset_env, sr=sr,
...                                  prior=prior)
>>> melspec = librosa.feature.melspectrogram(y=y, sr=sr)
>>> import matplotlib.pyplot as plt
>>> ax = plt.subplot(3,1,1)
>>> librosa.display.specshow(librosa.power_to_db(melspec,
...                                              ref=np.max),
...                          x_axis='time', y_axis='mel')
>>> plt.title('Mel spectrogram')
>>> plt.subplot(3,1,2, sharex=ax)
>>> plt.plot(librosa.times_like(onset_env),
...          librosa.util.normalize(onset_env),
...          label='Onset strength')
>>> plt.plot(librosa.times_like(pulse),
...          librosa.util.normalize(pulse),
...          label='Predominant local pulse (PLP)')
>>> plt.title('Uniform tempo prior [30, 300]')
>>> plt.subplot(3,1,3, sharex=ax)
>>> plt.plot(librosa.times_like(onset_env),
...          librosa.util.normalize(onset_env),
...          label='Onset strength')
>>> plt.plot(librosa.times_like(pulse_lognorm),
...          librosa.util.normalize(pulse_lognorm),
...          label='Predominant local pulse (PLP)')
>>> plt.title('Log-normal tempo prior, mean=120')
>>> plt.legend()
>>> plt.xlim([30, 35])
>>> plt.tight_layout()
>>> plt.show()
```



PLP local maxima can be used as estimates of beat positions.

```
>>> tempo, beats = librosa.beat.beat_track(onset_envelope=onset_env)
>>> beats_plp = np.flatnonzero(librosa.util.localmax(pulse))
>>> import matplotlib.pyplot as plt
>>> ax = plt.subplot(2,1,1)
>>> times = librosa.times_like(onset_env, sr=sr)
>>> plt.plot(times, librosa.util.normalize(onset_env),
...          label='Onset strength')
>>> plt.vlines(times[beats], 0, 1, alpha=0.5, color='r',
...            linestyle='--', label='Beats')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.title('librosa.beat.beat_track')
>>> # Limit the plot to a 15-second window
>>> plt.subplot(2,1,2, sharex=ax)
>>> times = librosa.times_like(pulse, sr=sr)
>>> plt.plot(times, librosa.util.normalize(pulse),
...          label='PLP')
>>> plt.vlines(times[beats_plp], 0, 1, alpha=0.5, color='r',
...            linestyle='--', label='PLP Beats')
>>> plt.legend(frameon=True, framealpha=0.75)
>>> plt.title('librosa.beat.plp')
>>> plt.xlim(30, 35)
>>> ax.xaxis.set_major_formatter(librosa.display.TimeFormatter())
>>> plt.tight_layout()
>>> plt.show()
```



librosa.beat.tempo

`librosa.beat.tempo`(*y=None, sr=22050, onset_envelope=None, hop_length=512, start_bpm=120, std_bpm=1.0, ac_size=8.0, max_tempo=320.0, aggregate=<function mean at 0x7fcba2eb26a8>, prior=None*)[source]

Estimate the tempo (beats per minute)

Parameters: **y** : np.ndarray [shape=(n,)] or None

audio time series

sr : number > 0 [scalar]

sampling rate of the time series

onset_envelope : np.ndarray [shape=(n,)]

pre-computed onset strength envelope

hop_length : int > 0 [scalar]

hop length of the time series

start_bpm : float [scalar]

initial guess of the BPM

std_bpm : float > 0 [scalar]

standard deviation of tempo distribution

ac_size : float > 0 [scalar]

length (in seconds) of the auto-correlation window

max_tempo : float > 0 [scalar, optional]

If provided, only estimate tempo below this threshold

aggregate : callable [optional]

Aggregation function for estimating global tempo. If *None*, then tempo is estimated independently for each frame.

prior : scipy.stats.rv_continuous [optional]

A prior distribution over tempo (in beats per minute). By default, a pseudo-log-normal prior is used. If given, *start_bpm* and *std_bpm* will be ignored.

Returns: **tempo** : np.ndarray [scalar]

estimated tempo (beats per minute)

See also

`librosa.onset.onset_strength`
`librosa.feature.tempogram`

Notes

This function caches at level 30.

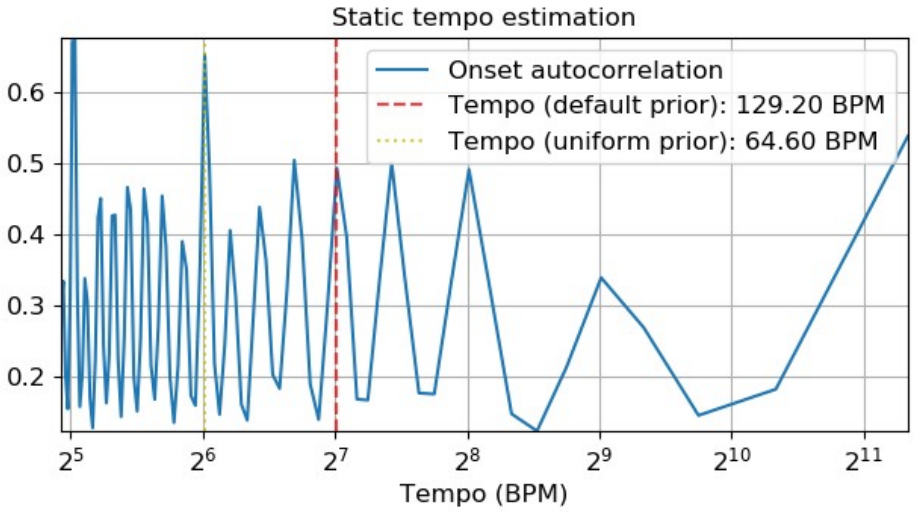
Examples

```
>>> # Estimate a static tempo
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> onset_env = librosa.onset.onset_strength(y, sr=sr)
>>> tempo = librosa.beat.tempo(onset_envelope=onset_env, sr=sr)
>>> tempo
array([129.199])

>>> # Or a static tempo with a uniform prior instead
>>> import scipy.stats
>>> prior = scipy.stats.uniform(30, 300)  # uniform over 30-300 BPM
>>> utempo = librosa.beat.tempo(onset_envelope=onset_env, sr=sr,
prior=prior)
>>> utempo
array([64.6])
```

```
>>> # Or a dynamic tempo
>>> dtempo = librosa.beat.tempo(onset_envelope=onset_env, sr=sr,
...                             aggregate=None)
>>> dtempo
array([ 143.555,  143.555,  143.555, ...,  161.499,  161.499,
        172.266])

>>> # Dynamic tempo with a proper log-normal prior
>>> prior_lognorm = scipy.stats.lognorm(loc=np.log(120), scale=120, s=1)
>>> dtempo_lognorm = librosa.beat.tempo(onset_envelope=onset_env, sr=sr,
...                                     aggregate=None,
...                                     prior=prior_lognorm)
>>> dtempo_lognorm
array([ 86.133,  86.133, ..., 129.199, 129.199])
```

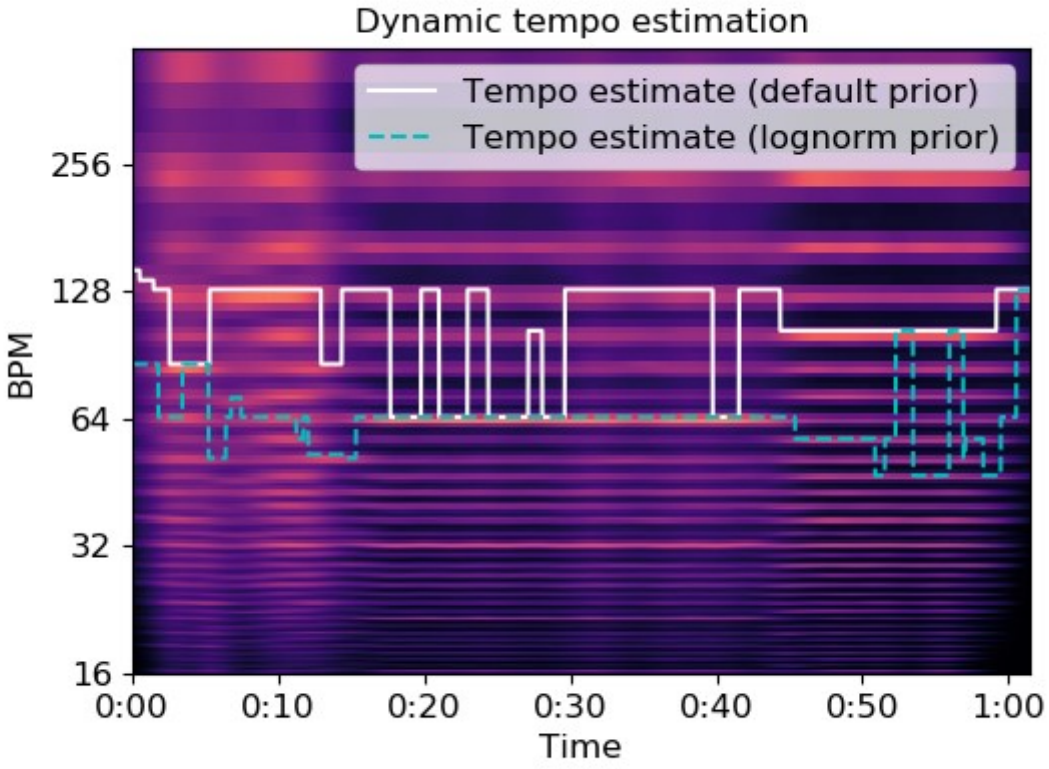
Plot the estimated tempo against the onset autocorrelation

```
>>> import matplotlib.pyplot as plt
>>> # Convert to scalar
>>> tempo = tempo.item()
>>> utempo = utempo.item()
>>> # Compute 2-second windowed autocorrelation
>>> hop_length = 512
>>> ac = librosa.autocorrelate(onset_env, 2 * sr // hop_length)
>>> freqs = librosa.tempo_frequencies(len(ac), sr=sr,
...                                   hop_length=hop_length)
>>> # Plot on a BPM axis.  We skip the first (0-lag) bin.
>>> plt.figure(figsize=(8,4))
>>> plt.semilogx(freqs[1:], librosa.util.normalize(ac)[1:],
...              label='Onset autocorrelation', basex=2)
>>> plt.axvline(tempo, 0, 1, color='r', alpha=0.75, linestyle='--',
...             label='Tempo (default prior): {:.2f} BPM'.format(tempo))
>>> plt.axvline(utempo, 0, 1, color='y', alpha=0.75, linestyle=':',
...             label='Tempo (uniform prior): {:.2f} BPM'.format(utempo))
>>> plt.xlabel('Tempo (BPM)')
>>> plt.grid()
>>> plt.title('Static tempo estimation')
>>> plt.legend(frameon=True)
>>> plt.axis('tight')
>>> plt.show()
```



Plot dynamic tempo estimates over a tempogram

```
>>> plt.figure()
>>> tg = librosa.feature.tempogram(onset_envelope=onset_env, sr=sr,
...                                hop_length=hop_length)
>>> librosa.display.specshow(tg, x_axis='time', y_axis='tempo')
>>> plt.plot(librosa.times_like(dtempo), dtempo,
...          color='w', linewidth=1.5, label='Tempo estimate (default prior)')
>>> plt.plot(librosa.times_like(dtempo_lognorm), dtempo_lognorm,
...          color='c', linewidth=1.5, linestyle='--',
...          label='Tempo estimate (lognorm prior)')
>>> plt.title('Dynamic tempo estimation')
>>> plt.legend(frameon=True, framealpha=0.75)
```



Spectrogram decomposition

decompose(S[, n_components, transformer, …])	Decompose a feature matrix.
hpss(S[, kernel_size, power, mask, margin])	Median-filtering harmonic percussive source separation (HPSS).
nn_filter(S[, rec, aggregate, axis])	Filtering by nearest-neighbors.

librosa.decompose.decompose

`librosa.decompose.decompose`(*S*, *n_components=None*, *transformer=None*, *sort=False*, *fit=True*, ***kwargs*)[source]

Decompose a feature matrix.

Given a spectrogram *S*, produce a decomposition into *components* and *activations* such that *S ~= components.dot(activations)*.

By default, this is done with with non-negative matrix factorization (NMF), but any `sklearn.decomposition`-type object will work.

Parameters: S : np.ndarray [shape=(n_features, n_samples), dtype=float]

The input feature matrix (e.g., magnitude spectrogram)

n_components : int > 0 [scalar] or None

number of desired components

if None, then *n_features* components are used

transformer : None or object

If None, use `sklearn.decomposition.NMF`

Otherwise, any object with a similar interface to NMF should work. *transformer* must follow the scikit-learn convention, where input data is *(n_samples, n_features)*.

transformer.fit_transform() will be run on *S.T* (not *S*), the return value of which is stored (transposed) as *activations*

The components will be retrieved as *transformer.components_.T*

S ~= np.dot(activations, transformer.components_).T

or equivalently: *S ~= np.dot(transformer.components_.T, activations.T)*

sort : bool

If *True*, components are sorted by ascending peak frequency.

Note

If used with *transformer*, sorting is applied to copies of the

decomposition parameters, and not to *transformer*'s internal parameters.

fit : bool

If *True*, components are estimated from the input `S`.

If *False*, components are assumed to be pre-computed and stored in `transformer`, and are not changed.

kwargs : Additional keyword arguments to the default transformer

`sklearn.decomposition.NMF`

Returns: components: np.ndarray [shape=(n_features, n_components)]

matrix of components (basis elements).

activations: np.ndarray [shape=(n_components, n_samples)]

transformed matrix/activation matrix

Raises: ParameterError

if *fit* is False and no *transformer* object is provided.

See also

`sklearn.decomposition`
SciKit-Learn matrix decomposition modules

Examples

Decompose a magnitude spectrogram into 32 components with NMF

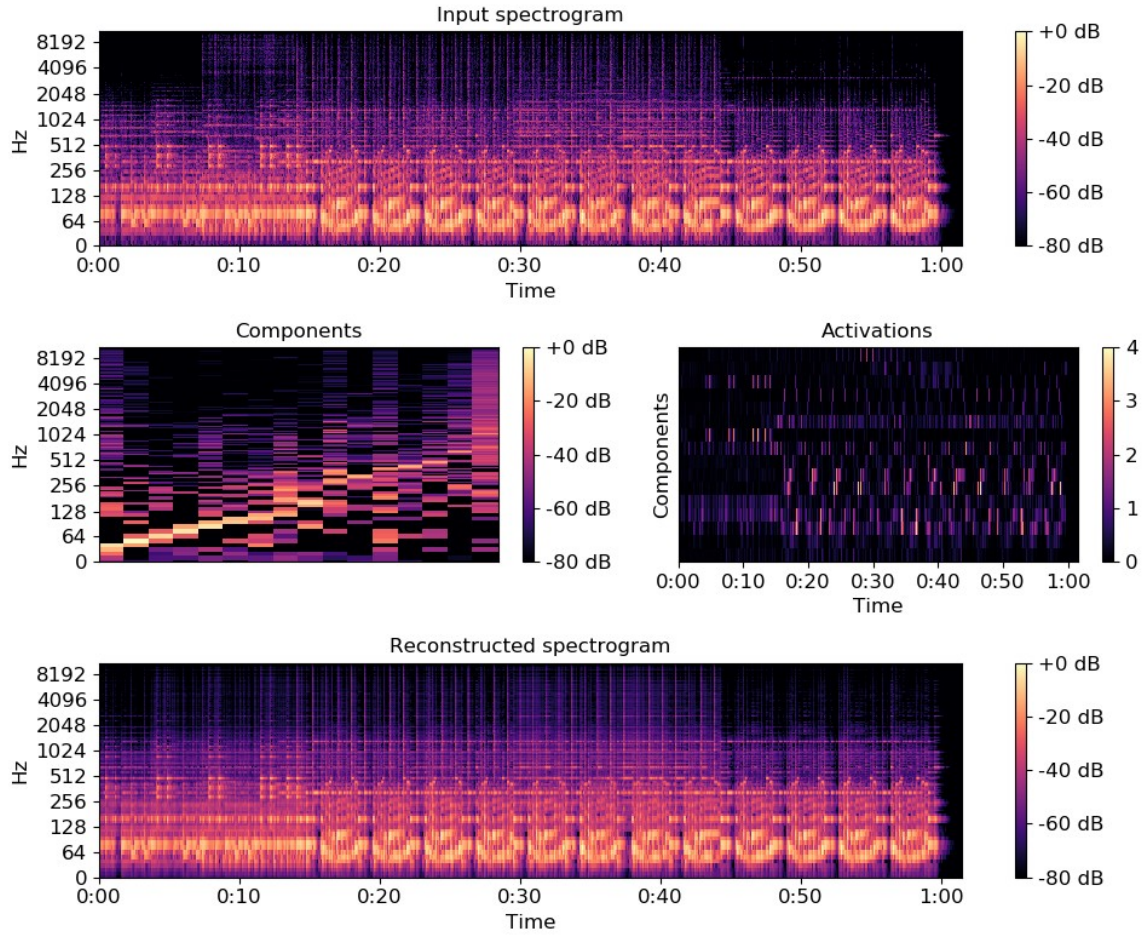
```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
>>> comps, acts = librosa.decompose.decompose(S, n_components=8)
>>> comps
array([[  1.876e-01,   5.559e-02, ...,   1.687e-01,   4.907e-02],
       [  3.148e-01,   1.719e-01, ...,   2.314e-01,   9.493e-02],
       ...,
       [  1.561e-07,   8.564e-08, ...,   7.167e-08,   4.997e-08],
       [  1.531e-07,   7.880e-08, ...,   5.632e-08,   4.028e-08]])
>>> acts
array([[  4.197e-05,   8.512e-03, ...,   3.056e-05,   9.159e-06],
       [  9.568e-06,   1.718e-02, ...,   3.322e-05,   7.869e-06],
       ...,
       [  5.982e-05,   1.311e-02, ...,  -0.000e+00,   6.323e-06],
       [  3.782e-05,   7.056e-03, ...,   3.290e-05,  -0.000e+00]])
```

Sort components by ascending peak frequency

```
>>> comps, acts = librosa.decompose.decompose(S, n_components=16,
...                                           sort=True)
```

Or with sparse dictionary learning

```
>>> import sklearn.decomposition
>>> T = sklearn.decomposition.MiniBatchDictionaryLearning(n_components=16)
>>> scomps, sacts = librosa.decompose.decompose(S, transformer=T,
sort=True)

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10,8))
>>> plt.subplot(3, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(S,
...                                                  ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.title('Input spectrogram')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.subplot(3, 2, 3)
>>> librosa.display.specshow(librosa.amplitude_to_db(comps,
...                                                  ref=np.max),
...                          y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Components')
>>> plt.subplot(3, 2, 4)
>>> librosa.display.specshow(acts, x_axis='time')
>>> plt.ylabel('Components')
>>> plt.title('Activations')
>>> plt.colorbar()
>>> plt.subplot(3, 1, 3)
>>> S_approx = comps.dot(acts)
>>> librosa.display.specshow(librosa.amplitude_to_db(S_approx,
...                                                  ref=np.max),
...                          y_axis='log', x_axis='time')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Reconstructed spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.decompose.hpss

`librosa.decompose.hpss`(*S*, *kernel_size=31*, *power=2.0*, *mask=False*, *margin=1.0*)[source]

Median-filtering harmonic percussive source separation (HPSS).

If *margin = 1.0*, decomposes an input spectrogram $S = H + P$ where H contains the harmonic components, and P contains the percussive components.

If *margin > 1.0*, decomposes an input spectrogram $S = H + P + R$ where R contains residual components not included in H or P.

This implementation is based upon the algorithm described by [1] and [2].

[1] Fitzgerald, Derry. “Harmonic/percussive separation using median filtering.” 13th International Conference on Digital Audio Effects (DAFX10), Graz, Austria, 2010.

[2] *(1, 2)* Driedger, Müller, Disch. “Extending harmonic-percussive separation of audio.” 15th International Society for Music Information Retrieval Conference (ISMIR 2014), Taipei, Taiwan, 2014.

Parameters: S : np.ndarray [shape=(d, n)]

input spectrogram. May be real (magnitude) or complex.

kernel_size : int or tuple (kernel_harmonic, kernel_percussive)

kernel size(s) for the median filters.

- If scalar, the same size is used for both harmonic and percussive.
- If tuple, the first value specifies the width of the harmonic filter, and the second value specifies the width of the percussive filter.

power : float > 0 [scalar]

Exponent for the Wiener filter when constructing soft mask matrices.

mask : bool

Return the masking matrices instead of components.

Masking matrices contain non-negative real values that can be used to measure the assignment of energy from *S* into harmonic or percussive components.

Components can be recovered by multiplying *S* * *mask_H* or *S* * *mask_P*.

margin : float or tuple (margin_harmonic, margin_percussive)

margin size(s) for the masks (as described in [2])

- If scalar, the same size is used for both harmonic and percussive.
- If tuple, the first value specifies the margin of the harmonic mask, and the second value specifies the margin of the percussive mask.

Returns: **harmonic** : np.ndarray [shape=(d, n)]

harmonic component (or mask)

percussive : np.ndarray [shape=(d, n)]

percussive component (or mask)

See also

`util.softmask`

Notes

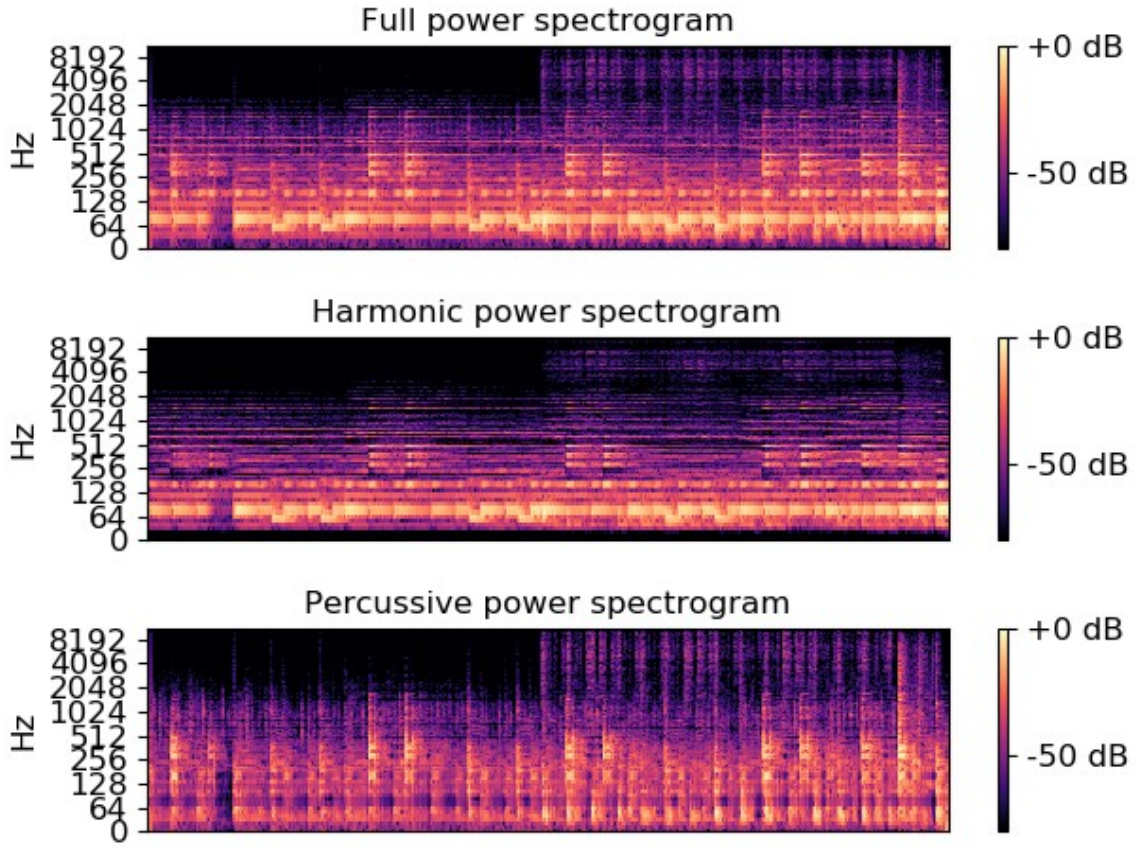
This function caches at level 30.

Examples

Separate into harmonic and percussive

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=15)
```

```
>>> D = librosa.stft(y)
>>> H, P = librosa.decompose.hpss(D)

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.subplot(3, 1, 1)
>>> librosa.display.specshow(librosa.amplitude_to_db(np.abs(D),
...                                                  ref=np.max),
...                          y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Full power spectrogram')
>>> plt.subplot(3, 1, 2)
>>> librosa.display.specshow(librosa.amplitude_to_db(np.abs(H),
...                                                  ref=np.max),
...                          y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Harmonic power spectrogram')
>>> plt.subplot(3, 1, 3)
>>> librosa.display.specshow(librosa.amplitude_to_db(np.abs(P),
...                                                  ref=np.max),
...                          y_axis='log')
>>> plt.colorbar(format='%+2.0f dB')
>>> plt.title('Percussive power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



Or with a narrower horizontal filter

```
>>> H, P = librosa.decompose.hpss(D, kernel_size=(13, 31))
```

Just get harmonic/percussive masks, not the spectra

```
>>> mask_H, mask_P = librosa.decompose.hpss(D, mask=True)
>>> mask_H
array([[  1.000e+00,   1.469e-01, ...,   2.648e-03,   2.164e-03],
       [  1.000e+00,   2.368e-01, ...,   9.413e-03,   7.703e-03],
       ...,
       [  8.869e-01,   5.673e-02, ...,   4.603e-02,   1.247e-05],
       [  7.068e-01,   2.194e-02, ...,   4.453e-02,   1.205e-05]],
dtype=float32)
>>> mask_P
array([[  2.858e-05,   8.531e-01, ...,   9.974e-01,   9.978e-01],
       [  1.586e-05,   7.632e-01, ...,   9.906e-01,   9.923e-01],
       ...,
       [  1.131e-01,   9.433e-01, ...,   9.540e-01,   1.000e+00],
       [  2.932e-01,   9.781e-01, ...,   9.555e-01,   1.000e+00]],
dtype=float32)
```

Separate into harmonic/percussive/residual components by using a margin > 1.0

```
>>> H, P = librosa.decompose.hpss(D, margin=3.0)
>>> R = D - (H+P)
>>> y_harm = librosa.core.istft(H)
>>> y_perc = librosa.core.istft(P)
>>> y_resi = librosa.core.istft(R)
```

Get a more isolated percussive component by widening its margin

```
>>> H, P = librosa.decompose.hpss(D, margin=(1.0,5.0))
```

librosa.decompose.nn_filter

`librosa.decompose.nn_filter`(*S*, *rec=None*, *aggregate=None*, *axis=-1*, ***kwargs*) [source]

Filtering by nearest-neighbors.

Each data point (e.g, spectrogram column) is replaced by aggregating its nearest neighbors in feature space.

This can be useful for de-noising a spectrogram or feature matrix.

The non-local means method [1] can be recovered by providing a weighted recurrence matrix as input and specifying *aggregate=np.average*.

Similarly, setting *aggregate=np.median* produces sparse de-noising as in REPET-SIM [2].

[1] Buades, A., Coll, B., & Morel, J. M. (2005, June). A non-local algorithm for image denoising. In Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on (Vol. 2, pp. 60-65). IEEE.

[2] Rafii, Z., & Pardo, B. (2012, October). “Music/Voice Separation Using the Similarity Matrix.” International Society for Music Information Retrieval Conference, 2012.

Parameters: S : np.ndarray

The input data (spectrogram) to filter

rec : (optional) scipy.sparse.spmatrix or np.ndarray

Optionally, a pre-computed nearest-neighbor matrix as provided by `librosa.segment.recurrence_matrix`

aggregate : function

aggregation function (default: `np.mean`)

If *aggregate=np.average*, then a weighted average is computed according to the (per-row) weights in *rec*.

For all other aggregation functions, all neighbors are treated equally.

axis : int

The axis along which to filter (by default, columns)

kwargs

Additional keyword arguments provided to `librosa.segment.recurrence_matrix` if *rec* is not provided

Returns: **S_filtered** : np.ndarray

The filtered data

Raises: ParameterError

if *rec* is provided and its shape is incompatible with *S*.

See also

`decompose`
`hpss`
`librosa.segment.recurrence_matrix`

Notes

This function caches at level 30.

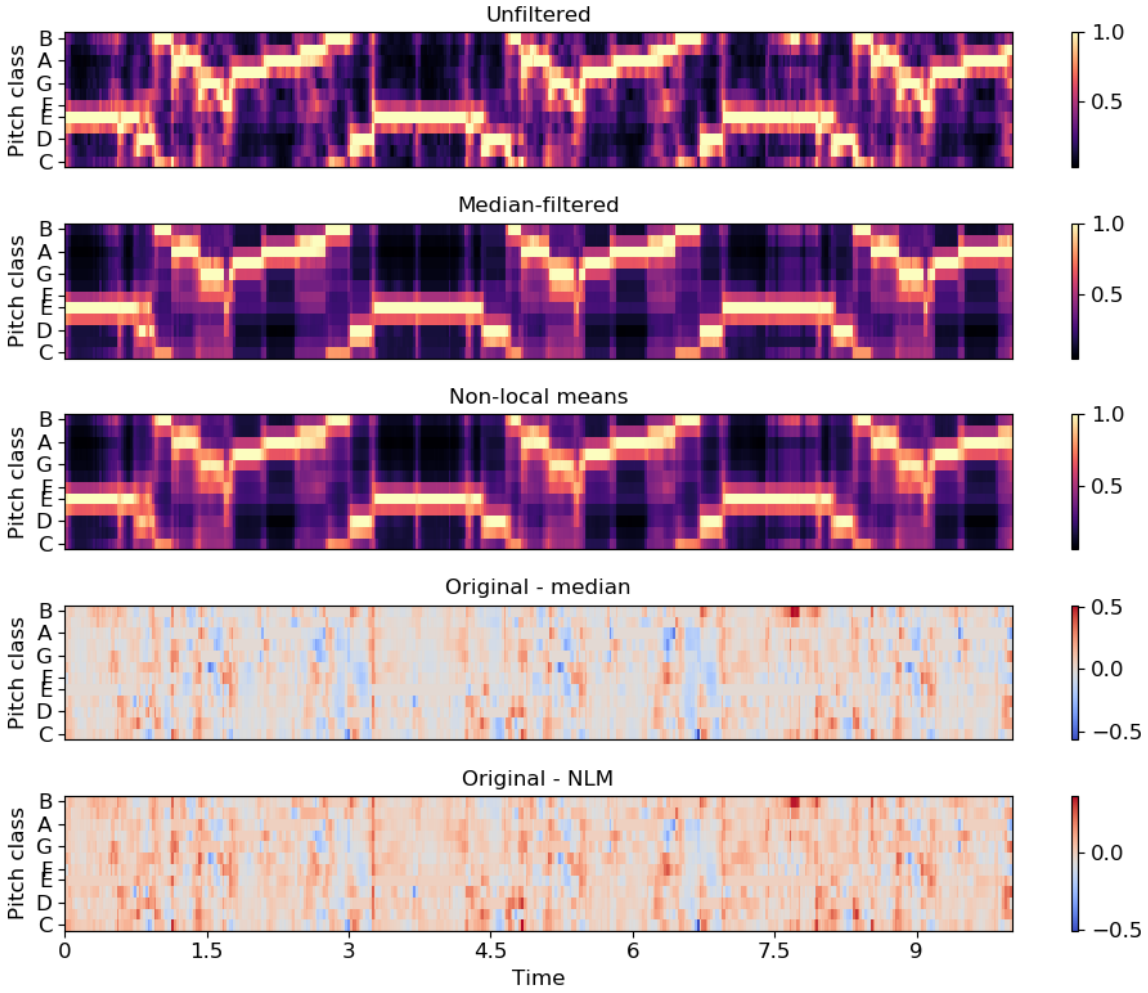
Examples

De-noise a chromagram by non-local median filtering. By default this would use euclidean distance to select neighbors, but this can be overridden directly by setting the *metric* parameter.

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=30, duration=10)
>>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr)
>>> chroma_med = librosa.decompose.nn_filter(chroma,
...                                          aggregate=np.median,
...                                          metric='cosine')
```

To use non-local means, provide an affinity matrix and *aggregate=np.average*.

```
>>> rec = librosa.segment.recurrence_matrix(chroma, mode='affinity',
...                                         metric='cosine', sparse=True)
>>> chroma_nlm = librosa.decompose.nn_filter(chroma, rec=rec,
...                                          aggregate=np.average)

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10, 8))
>>> plt.subplot(5, 1, 1)
>>> librosa.display.specshow(chroma, y_axis='chroma')
>>> plt.colorbar()
>>> plt.title('Unfiltered')
>>> plt.subplot(5, 1, 2)
>>> librosa.display.specshow(chroma_med, y_axis='chroma')
>>> plt.colorbar()
>>> plt.title('Median-filtered')
>>> plt.subplot(5, 1, 3)
>>> librosa.display.specshow(chroma_nlm, y_axis='chroma')
>>> plt.colorbar()
>>> plt.title('Non-local means')
>>> plt.subplot(5, 1, 4)
>>> librosa.display.specshow(chroma - chroma_med,
...                          y_axis='chroma')
>>> plt.colorbar()
>>> plt.title('Original - median')
>>> plt.subplot(5, 1, 5)
>>> librosa.display.specshow(chroma - chroma_nlm,
...                          y_axis='chroma', x_axis='time')
>>> plt.colorbar()
>>> plt.title('Original - NLM')
>>> plt.tight_layout()
>>> plt.show()
```


Unfiltered
Median-filtered
Non-local means
Original - median
Original - NLM
Pitch class
B
A
G
F♯
E
D
C
1.0
0.5
0.0
−0.5
0
1.5
3
4.5
6
7.5
9
Time

Effects

Harmonic-percussive source separation

hpss(y, **kwargs)	Decompose an audio time series into harmonic and percussive components.
harmonic(y, **kwargs)	Extract harmonic elements from an audio time-series.
percussive(y, **kwargs)	Extract percussive elements from an audio time-series.

Time and frequency

time_stretch(y, rate, **kwargs)	Time-stretch an audio series by a fixed rate.
pitch_shift(y, sr, n_steps[, …])	Shift the pitch of a waveform by *n_steps* semitones.

Miscellaneous

remix(y, intervals[, align_zeros])	Remix an audio signal by re-ordering time intervals.
trim(y[, top_db, ref, frame_length, hop_length])	Trim leading and trailing silence from an audio signal.
split(y[, top_db, ref, frame_length, hop_length])	Split an audio signal into non-silent intervals.
preemphasis(y[, coef, zi, return_zf])	Pre-emphasize an audio signal with a first-order auto-regressive filter:

librosa.effects.hpss

`librosa.effects.hpss`(*y*, ***kwargs*)[source]

Decompose an audio time series into harmonic and percussive components.

This function automates the STFT->HPSS->ISTFT pipeline, and ensures that the output waveforms have equal length to the input waveform *y*.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

kwargs : additional keyword arguments.

See `librosa.decompose.hpss` for details.

Returns: **y_harmonic** : np.ndarray [shape=(n,)]

audio time series of the harmonic elements

y_percussive : np.ndarray [shape=(n,)]

audio time series of the percussive elements

See also

`harmonic`
Extract only the harmonic component
`percussive`
Extract only the percussive component
`librosa.decompose.hpss`
HPSS on spectrograms

Examples

```
>>> # Extract harmonic and percussive components
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y_harmonic, y_percussive = librosa.effects.hpss(y)

>>> # Get a more isolated percussive component by widening its margin
>>> y_harmonic, y_percussive = librosa.effects.hpss(y, margin=(1.0,5.0))
```

librosa.effects.harmonic

`librosa.effects.harmonic`(*y*, ***kwargs*)[source]

Extract harmonic elements from an audio time-series.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

kwargs : additional keyword arguments.

See `librosa.decompose.hpss` for details.

Returns: **y_harmonic** : np.ndarray [shape=(n,)]

audio time series of just the harmonic portion

See also

`hpss`
Separate harmonic and percussive components
`percussive`

Extract only the percussive component

`librosa.decompose.hpss`
HPSS for spectrograms

Examples

```
>>> # Extract harmonic component
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y_harmonic = librosa.effects.harmonic(y)

>>> # Use a margin > 1.0 for greater harmonic separation
>>> y_harmonic = librosa.effects.harmonic(y, margin=3.0)
```

librosa.effects.percussive

`librosa.effects.percussive`(*y*, ***kwargs*)[source]

Extract percussive elements from an audio time-series.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

kwargs : additional keyword arguments.

See `librosa.decompose.hpss` for details.

Returns: **y_percussive** : np.ndarray [shape=(n,)]

audio time series of just the percussive portion

See also

`hpss`
Separate harmonic and percussive components

`harmonic`
Extract only the harmonic component

`librosa.decompose.hpss`
HPSS for spectrograms

Examples

```
>>> # Extract percussive component
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y_percussive = librosa.effects.percussive(y)

>>> # Use a margin > 1.0 for greater percussive separation
>>> y_percussive = librosa.effects.percussive(y, margin=3.0)
```

librosa.effects.time_stretch

librosa.effects.time_stretch(*y*, *rate*, ***kwargs*)[source]

Time-stretch an audio series by a fixed rate.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

rate : float > 0 [scalar]

Stretch factor. If *rate > 1*, then the signal is sped up. If *rate < 1*, then the signal is slowed down.

kwargs : additional keyword arguments.

See *librosa.decompose.stft* for details.

Returns: **y_stretch** : np.ndarray [shape=(round(n/rate),)]

audio time series stretched by the specified rate

See also

pitch_shift
: pitch shifting

librosa.core.phase_vocoder
: spectrogram phase vocoder

pyrubberband.pyrb.time_stretch
: high-quality time stretching using RubberBand

Examples

Compress to be twice as fast

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y_fast = librosa.effects.time_stretch(y, 2.0)
```

Or half the original speed

```
>>> y_slow = librosa.effects.time_stretch(y, 0.5)
```

librosa.effects.pitch_shift

librosa.effects.pitch_shift(*y*, *sr*, *n_steps*, *bins_per_octave=12*, *res_type='kaiser_best'*, ***kwargs*)[source]

Shift the pitch of a waveform by *n_steps* semitones.

Parameters: **y** : np.ndarray [shape=(n,)]

audio time series

sr : number > 0 [scalar]

audio sampling rate of *y*

n_steps : float [scalar]

how many (fractional) half-steps to shift *y*

bins_per_octave : float > 0 [scalar]

how many steps per octave

res_type : string

Resample type. Possible options: ‘kaiser_best’, ‘kaiser_fast’, and ‘scipy’, ‘polyphase’, ‘fft’. By default, ‘kaiser_best’ is used.

See `core.resample` for more information.

kwargs: additional keyword arguments.

See *librosa.decompose.stft* for details.

Returns: **y_shift** : np.ndarray [shape=(n,)]

The pitch-shifted audio time-series

See also

`time_stretch`
time stretching

`librosa.core.phase_vocoder`
spectrogram phase vocoder

`pyrubberband.pyrb.pitch_shift`
high-quality pitch shifting using RubberBand

Examples

Shift up by a major third (four half-steps)

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> y_third = librosa.effects.pitch_shift(y, sr, n_steps=4)
```

Shift down by a tritone (six half-steps)

```
>>> y_tritone = librosa.effects.pitch_shift(y, sr, n_steps=-6)
```

Shift up by 3 quarter-tones

```
>>> y_three_qt = librosa.effects.pitch_shift(y, sr, n_steps=3,
...                                          bins_per_octave=24)
```

librosa.effects.remix

`librosa.effects.remix`(*y*, *intervals*, *align_zeros=True*)[source]

Remix an audio signal by re-ordering time intervals.

Parameters: **y** : np.ndarray [shape=(t,) or (2, t)]

Audio time series

intervals : iterable of tuples (start, end)

An iterable (list-like or generator) where the *i`th item `intervals[i]* indicates the start and end (in samples) of a slice of *y*.

align_zeros : boolean

If *True*, interval boundaries are mapped to the closest zero-crossing in *y*. If *y* is stereo, zero-crossings are computed after converting to mono.

Returns: **y_remix** : np.ndarray [shape=(d,) or (2, d)]

y remixed in the order specified by *intervals*

Examples

Load in the example track and reverse the beats

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
```

Compute beats

```
>>> _, beat_frames = librosa.beat.beat_track(y=y, sr=sr,
...                                          hop_length=512)
```

Convert from frames to sample indices

```
>>> beat_samples = librosa.frames_to_samples(beat_frames)
```

Generate intervals from consecutive events

```
>>> intervals = librosa.util.frame(beat_samples, frame_length=2,
...                                hop_length=1).T
```

Reverse the beat intervals

```
>>> y_out = librosa.effects.remix(y, intervals[::-1])
```

librosa.effects.trim

`librosa.effects.trim`(*y, top_db=60, ref=<function amax at 0x7fcba2eb3d90>, frame_length=2048, hop_length=512*)[source]

Trim leading and trailing silence from an audio signal.

Parameters: **y** : np.ndarray, shape=(n,) or (2,n)

Audio signal, can be mono or stereo

top_db : number > 0

The threshold (in decibels) below reference to consider as silence

ref : number or callable

The reference power. By default, it uses `np.max` and compares to the peak power in the signal.

frame_length : int > 0

The number of samples per analysis frame

hop_length : int > 0

The number of samples between analysis frames

Returns: **y_trimmed** : np.ndarray, shape=(m,) or (2, m)

The trimmed signal

index : np.ndarray, shape=(2,)

the interval of *y* corresponding to the non-silent region: *y_trimmed = y[index[0]:index[1]]* (for mono) or *y_trimmed = y[:, index[0]:index[1]]* (for stereo).

Examples

```
>>> # Load some audio
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> # Trim the beginning and ending silence
>>> yt, index = librosa.effects.trim(y)
>>> # Print the durations
>>> print(librosa.get_duration(y), librosa.get_duration(yt))
61.45886621315193 60.58086167800454
```

librosa.effects.split

`librosa.effects.split`(*y, top_db=60, ref=<function amax at 0x7fcba2eb3d90>, frame_length=2048, hop_length=512*)[source]

Split an audio signal into non-silent intervals.

Parameters: **y** : np.ndarray, shape=(n,) or (2, n)

An audio signal

top_db : number > 0

The threshold (in decibels) below reference to consider as silence

ref : number or callable

The reference power. By default, it uses `np.max` and compares to the peak power in the signal.

frame_length : int > 0

The number of samples per analysis frame

hop_length : int > 0

The number of samples between analysis frames

Returns: **intervals** : np.ndarray, shape=(m, 2)

intervals[i] == (start_i, end_i) are the start and end time (in samples) of non-silent interval *i*.

librosa.effects.preemphasis

`librosa.effects.preemphasis`(*y*, *coef=0.97*, *zi=None*, *return_zf=False*)[source]

Pre-emphasize an audio signal with a first-order auto-regressive filter:

y[n] -> y[n] - coef * y[n-1]

Parameters: **y** : np.ndarray

Audio signal

coef : positive number

Pre-emphasis coefficient. Typical values of *coef* are between 0 and 1.

At the limit *coef=0*, the signal is unchanged.

At *coef=1*, the result is the first-order difference of the signal.

zi : number

Initial filter state

return_zf : boolean

If *True*, return the final filter state. If *False*, only return the pre-emphasized signal.

Returns: **`y_out`** : np.ndarray

pre-emphasized signal

zf : number

if *return_zf=True*, the final filter state is also returned

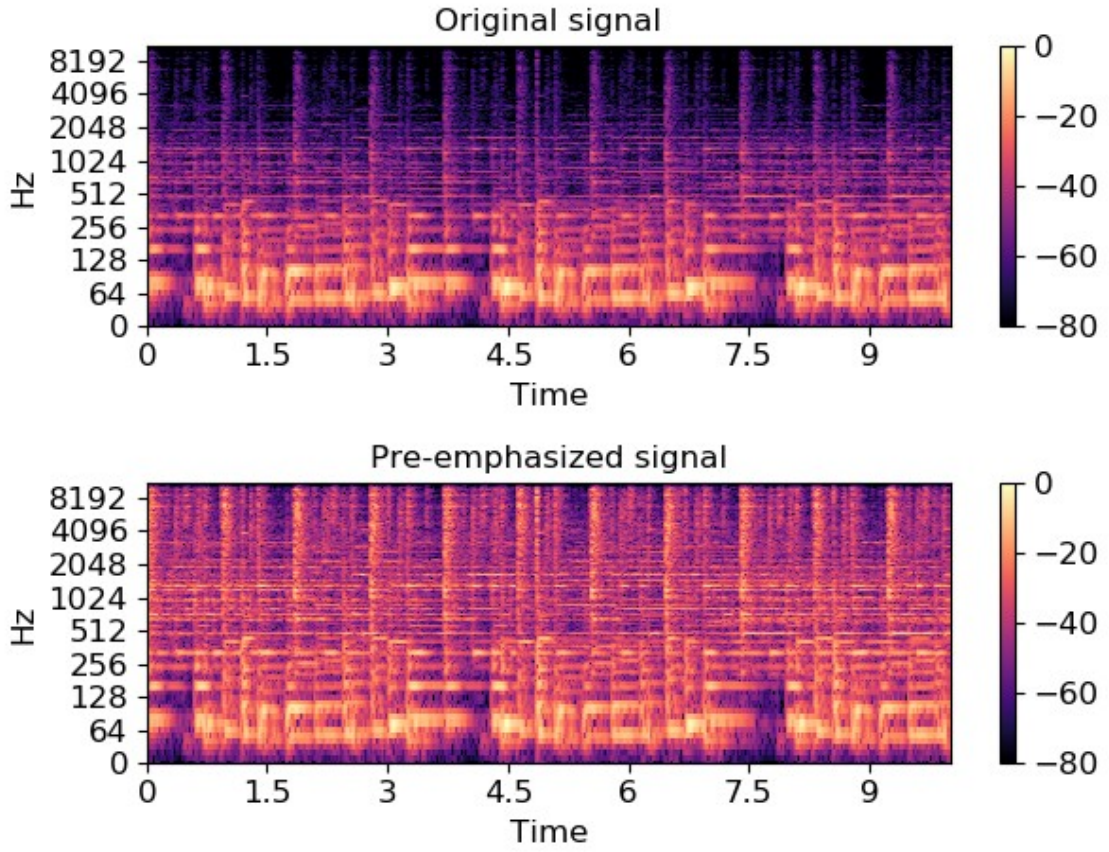
Examples

Apply a standard pre-emphasis filter

```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=30,
duration=10)
>>> y_filt = librosa.effects.preemphasis(y)
>>> # and plot the results for comparison
>>> S_orig = librosa.amplitude_to_db(np.abs(librosa.stft(y)), ref=np.max)
>>> S_preemph = librosa.amplitude_to_db(np.abs(librosa.stft(y_filt)),
ref=np.max)
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(S_orig, y_axis='log', x_axis='time')
>>> plt.title('Original signal')
>>> plt.colorbar()
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(S_preemph, y_axis='log', x_axis='time')
>>> plt.title('Pre-emphasized signal')
>>> plt.colorbar()
>>> plt.tight_layout();
```

Apply pre-emphasis in pieces for block streaming. Note that the second block initializes *zi* with the final state *zf* returned by the first call.

```
>>> y_filt_1, zf = librosa.effects.preemphasis(y[:1000], return_zf=True)
>>> y_filt_2, zf = librosa.effects.preemphasis(y[1000:], zi=zf,
return_zf=True)
>>> np.allclose(y_filt, np.concatenate([y_filt_1, y_filt_2]))
True
```



Output

Note: the `librosa.output` module is deprecated as of version 0.7.0, and it will be removed entirely in version 0.8.

Text output

`annotation`(path, intervals[, annotations, …])	Save annotations in a 3-column format.
`times_csv`(path, times[, annotations, …])	Save time steps as in CSV format.

Audio output

`write_wav`(path, y, sr[, norm])	Output a time series as a .wav file

librosa.output.annotation

`librosa.output.annotation`(*path*, *intervals*, *annotations=None*, *delimiter=', '*, *fmt='%0.3f'*)[source]

Save annotations in a 3-column format:

```
intervals[0, 0],intervals[0, 1],annotations[0]\n
intervals[1, 0],intervals[1, 1],annotations[1]\n
```

```
intervals[2, 0],intervals[2, 1],annotations[2]\n
...
```

This can be used for segment or chord annotations.

Warning

This function is deprecated in librosa 0.7.0. It will be removed in 0.8.

Parameters: **path** : str

path to save the output CSV file

intervals : np.ndarray [shape=(n, 2)]

array of interval start and end-times.

intervals[i, 0] marks the start time of interval *i*

intervals[i, 1] marks the end time of interval *i*

annotations : None or list-like [shape=(n,)]

optional list of annotation strings. *annotations[i]* applies to the time range *intervals[i, 0]* to *intervals[i, 1]*

delimiter : str

character to separate fields

fmt : str

format-string for rendering time data

Raises: ParameterError

if *annotations* is not *None* and length does not match *intervals*

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> data = librosa.feature.mfcc(y=y, sr=sr, hop_length=512)
```

Detect segment boundaries

```
>>> boundaries = librosa.segment.agglomerative(data, k=10)
```

Convert to time

```
>>> boundary_times = librosa.frames_to_time(boundaries, sr=sr,
...                                         hop_length=512)
```

Convert events boundaries to intervals

```
>>> intervals = np.hstack([boundary_times[:-1, np.newaxis],
...                        boundary_times[1:, np.newaxis]])
```

Make some fake annotations

```
>>> labels = ['Seg #{:03d}'.format(i) for i in range(len(intervals))]
```

Save the output

```
>>> librosa.output.annotation('segments.csv', intervals,
...                           annotations=labels)
```

librosa.output.times_csv

`librosa.output.times_csv`(*path*, *times*, *annotations=None*, *delimiter=', '*, *fmt='%0.3f'*) [source]

Save time steps as in CSV format. This can be used to store the output of a beat-tracker or segmentation algorithm.

If only *times* are provided, the file will contain each value of *times* on a row:

```
times[0]\n
times[1]\n
times[2]\n
...
```

If *annotations* are also provided, the file will contain delimiter-separated values:

```
times[0],annotations[0]\n
times[1],annotations[1]\n
times[2],annotations[2]\n
...
```

Warning

This function is deprecated in librosa 0.7.0. It will be removed in 0.8.

Parameters: **path** : string

path to save the output CSV file

times : list-like of floats

list of frame numbers for beat events

annotations : None or list-like

optional annotations for each time step

delimiter : str

character to separate fields

fmt : str

format-string for rendering time

Raises: ParameterError

if *annotations* is not *None* and length does not match *times*

Examples

Write beat-tracker time to CSV

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> tempo, beats = librosa.beat.beat_track(y, sr=sr, units='time')
>>> librosa.output.times_csv('beat_times.csv', beats)
```

librosa.output.write_wav

`librosa.output.write_wav`(*path*, *y*, *sr*, *norm=False*)[source]

Output a time series as a .wav file

Note: only mono or stereo, floating-point data is supported.

Warning

This function is deprecated in librosa 0.7.0. It will be removed in 0.8. Usage of `write_wav` should be replaced by `soundfile.write`.

Parameters: **path** : str

path to save the output wav file

y : np.ndarray [shape=(n,) or (2,n), dtype=np.float]

audio time series (mono or stereo).

Note that only floating-point values are supported.

sr : int > 0 [scalar]

sampling rate of *y*

norm : boolean [scalar]

enable amplitude normalization. For floating point *y*, scale the data to the range [-1, +1].

See also

`soundfile.write`

Examples

Trim a signal to 5 seconds and save it back

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=5.0)
>>> librosa.output.write_wav('file_trim_5s.wav', y, sr)
```

Temporal segmentation

Recurrence and self-similarity

cross_similarity(data, data_ref[, k, …])	Compute cross-similarity from one data sequence to a reference sequence.
recurrence_matrix(data[, k, width, metric, …])	Compute a recurrence matrix from a data matrix.
recurrence_to_lag(rec[, pad, axis])	Convert a recurrence matrix into a lag matrix.
lag_to_recurrence(lag[, axis])	Convert a lag matrix into a recurrence matrix.
timelag_filter(function[, pad, index])	Filtering in the time-lag domain.
path_enhance(R, n[, window, max_ratio, …])	Multi-angle path enhancement for self- and cross-similarity matrices.

Temporal clustering

agglomerative(data, k[, clusterer, axis])	Bottom-up temporal segmentation.
subsegment(data, frames[, n_segments, axis])	Sub-divide a segmentation by feature clustering.

librosa.segment.cross_similarity

librosa.segment.cross_similarity(*data*, *data_ref*, *k=None*, *metric='euclidean'*, *sparse=False*, *mode='connectivity'*, *bandwidth=None*)[source]

Compute cross-similarity from one data sequence to a reference sequence.

The output is a matrix *xsim*:

xsim[i, j] is non-zero if *data_ref[:, i]* is a k-nearest neighbor of *data[:, j]*.

Parameters: data : np.ndarray [shape=(d, n)]

A feature matrix for the comparison sequence

data_ref : np.ndarray [shape=(d, n_ref)]

A feature matrix for the reference sequence

k : int > 0 [scalar] or None

the number of nearest-neighbors for each sample

Default: *k = 2 * ceil(sqrt(n_ref))*, or *k = 2* if *n_ref <= 3*

metric : str

Distance metric to use for nearest-neighbor calculation.

See `sklearn.neighbors.NearestNeighbors` for details.

sparse : bool [scalar]

if False, returns a dense type (ndarray) if True, returns a sparse type (scipy.sparse.csc_matrix)

mode : str, {‘connectivity’, ‘distance’, ‘affinity’}

If ‘connectivity’, a binary connectivity matrix is produced.

If ‘distance’, then a non-zero entry contains the distance between points.

If ‘affinity’, then non-zero entries are mapped to *exp(- distance(i, j) / bandwidth)* where *bandwidth* is as specified below.

bandwidth : None or float > 0

If using `mode='affinity'`, this can be used to set the bandwidth on the affinity kernel.

If no value is provided, it is set automatically to the median distance to the k’th nearest neighbor of each *data[:, i]*.

Returns: **xsim** : np.ndarray or scipy.sparse.csc_matrix, [shape=(n_ref, n)]

Cross-similarity matrix

See also

`recurrence_matrix`
`recurrence_to_lag`
`feature.stack_memory`
`sklearn.neighbors.NearestNeighbors`
`scipy.spatial.distance.cdist`

Notes

This function caches at level 30.

Examples

Find nearest neighbors in MFCC space between two sequences

```
>>> hop_length = 1024
>>> y_ref, sr = librosa.load(librosa.util.example_audio_file())
>>> y_comp, sr = librosa.load(librosa.util.example_audio_file(), offset=10)
>>> mfcc_ref = librosa.feature.mfcc(y=y_ref, sr=sr, hop_length=hop_length)
>>> mfcc_comp = librosa.feature.mfcc(y=y_comp, sr=sr,
hop_length=hop_length)
>>> xsim = librosa.segment.cross_similarity(mfcc_comp, mfcc_ref)
```

Or fix the number of nearest neighbors to 5

```
>>> xsim = librosa.segment.cross_similarity(mfcc_comp, mfcc_ref, k=5)
```

Use cosine similarity instead of Euclidean distance

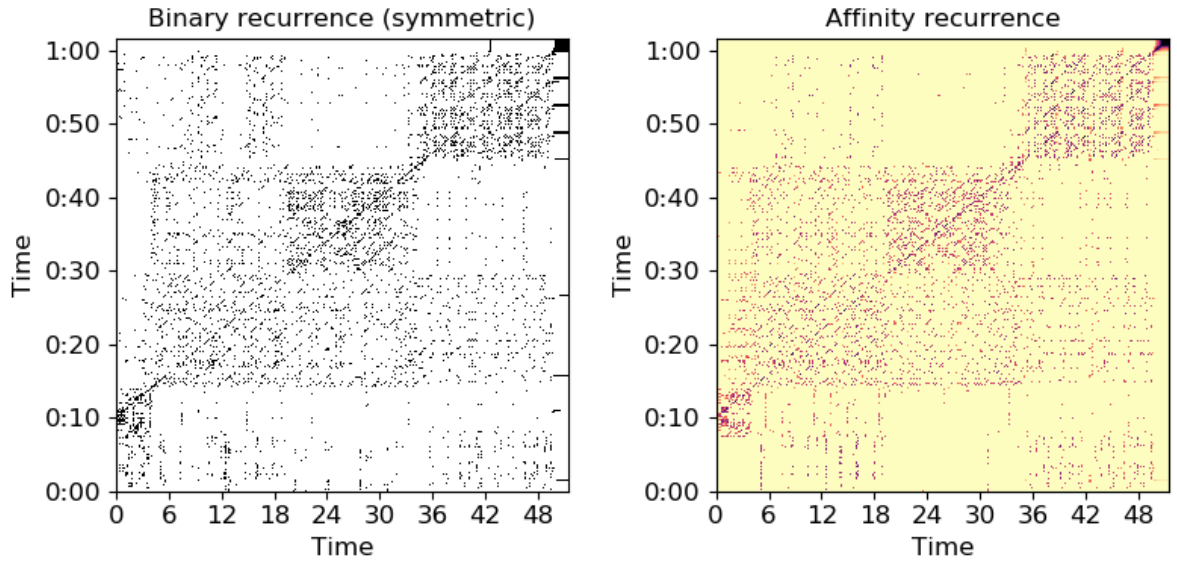
```
>>> xsim = librosa.segment.cross_similarity(mfcc_comp, mfcc_ref,
metric='cosine')
```

Use an affinity matrix instead of binary connectivity

```
>>> xsim_aff = librosa.segment.cross_similarity(mfcc_comp, mfcc_ref,
mode='affinity')
```

Plot the feature and recurrence matrices

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(1, 2, 1)
>>> librosa.display.specshow(xsim, x_axis='time', y_axis='time',
hop_length=hop_length)
>>> plt.title('Binary recurrence (symmetric)')
>>> plt.subplot(1, 2, 2)
>>> librosa.display.specshow(xsim_aff, x_axis='time', y_axis='time',
...                          cmap='magma_r', hop_length=hop_length)
>>> plt.title('Affinity recurrence')
>>> plt.tight_layout()
```



librosa.segment.recurrence_matrix

`librosa.segment.recurrence_matrix`(*data, k=None, width=1, metric='euclidean', sym=False, sparse=False, mode='connectivity', bandwidth=None, self=False, axis=-1*)[source]

Compute a recurrence matrix from a data matrix.

rec[i, j] is non-zero if *data[:, i]* is one of *data[:, j]*'s k-nearest-neighbors and $|i - j| >= width$

The specific value of *rec[i, j]* can have several forms, governed by the *mode* parameter below:

- Connectivity: *rec[i, j] = 1 or 0* indicates that frames *i* and *j* are repetitions
- Affinity: *rec[i, j] > 0* measures how similar frames *i* and *j* are. This is also known as a (sparse) self-similarity matrix.
- Distance: *rec[i, j] > 0* measures how distant frames *i* and *j* are. This is also known as a (sparse) self-distance matrix.

The general term *recurrence matrix* can refer to any of the three forms above.

Parameters: data : np.ndarray

A feature matrix

k : int > 0 [scalar] or None

the number of nearest-neighbors for each sample

Default: *k = 2 * ceil(sqrt(t - 2 * width + 1))*, or *k = 2* if *t <= 2 * width + 1*

width : int >= 1 [scalar]

only link neighbors *(data[:, i], data[:, j])* if $|i - j| >= width$

width cannot exceed the length of the data.

metric : str

Distance metric to use for nearest-neighbor calculation.

See `sklearn.neighbors.NearestNeighbors` for details.

sym : bool [scalar]

set *sym=True* to only link mutual nearest-neighbors

sparse : bool [scalar]

if False, returns a dense type (ndarray) if True, returns a sparse type (scipy.sparse.csc_matrix)

mode : str, {‘connectivity’, ‘distance’, ‘affinity’}

If ‘connectivity’, a binary connectivity matrix is produced.

If ‘distance’, then a non-zero entry contains the distance between points.

If ‘affinity’, then non-zero entries are mapped to *exp(- distance(i, j) / bandwidth)* where *bandwidth* is as specified below.

bandwidth : None or float > 0

If using `mode='affinity'`, this can be used to set the bandwidth on the affinity kernel.

If no value is provided, it is set automatically to the median distance between furthest nearest neighbors.

self : bool

If *True*, then the main diagonal is populated with self-links: 0 if `mode='distance'`, and 1 otherwise.

If *False*, the main diagonal is left empty.

axis : int

The axis along which to compute recurrence. By default, the last index (-1) is taken.

Returns: **rec** : np.ndarray or scipy.sparse.csc_matrix, [shape=(t, t)]

Recurrence matrix

See also

sklearn.neighbors.NearestNeighbors
scipy.spatial.distance.cdist
librosa.feature.stack_memory
recurrence_to_lag

Notes

This function caches at level 30.

Examples

Find nearest neighbors in MFCC space

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> hop_length = 1024
>>> mfcc = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length)
>>> R = librosa.segment.recurrence_matrix(mfcc)
```

Or fix the number of nearest neighbors to 5

```
>>> R = librosa.segment.recurrence_matrix(mfcc, k=5)
```

Suppress neighbors within +- 7 frames

```
>>> R = librosa.segment.recurrence_matrix(mfcc, width=7)
```

Use cosine similarity instead of Euclidean distance

```
>>> R = librosa.segment.recurrence_matrix(mfcc, metric='cosine')
```

Require mutual nearest neighbors

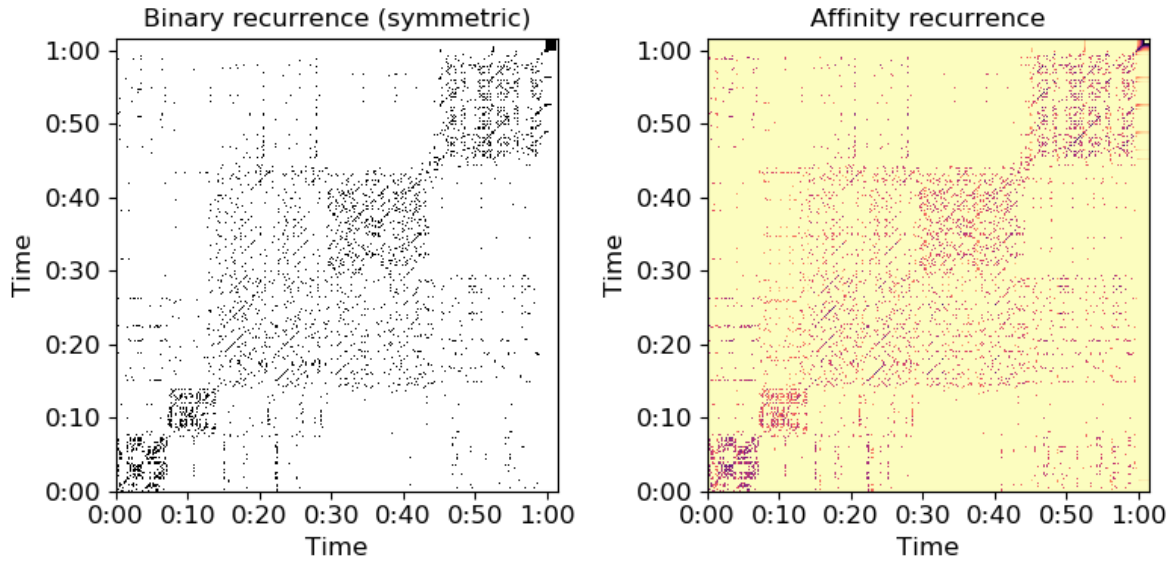
```
>>> R = librosa.segment.recurrence_matrix(mfcc, sym=True)
```

Use an affinity matrix instead of binary connectivity

```
>>> R_aff = librosa.segment.recurrence_matrix(mfcc, mode='affinity')
```

Plot the feature and recurrence matrices

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(1, 2, 1)
>>> librosa.display.specshow(R, x_axis='time', y_axis='time',
hop_length=hop_length)
>>> plt.title('Binary recurrence (symmetric)')
>>> plt.subplot(1, 2, 2)
>>> librosa.display.specshow(R_aff, x_axis='time', y_axis='time',
...                          hop_length=hop_length, cmap='magma_r')
>>> plt.title('Affinity recurrence')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.recurrence_to_lag

`librosa.segment.recurrence_to_lag`(*rec*, *pad=True*, *axis=-1*)[source]

Convert a recurrence matrix into a lag matrix.

lag[i, j] == rec[i+j, j]

Parameters: **rec** : np.ndarray, or scipy.sparse.spmatrix [shape=(n, n)]

A (binary) recurrence matrix, as returned by `recurrence_matrix`

pad : bool

If False, *lag* matrix is square, which is equivalent to assuming that the signal repeats itself indefinitely.

If True, *lag* is padded with *n* zeros, which eliminates the assumption of repetition.

axis : int

The axis to keep as the `time` axis. The alternate axis will be converted to lag coordinates.

Returns: **lag** : np.ndarray

The recurrence matrix in (lag, time) (if *axis=1*) or (time, lag) (if *axis=0*) coordinates

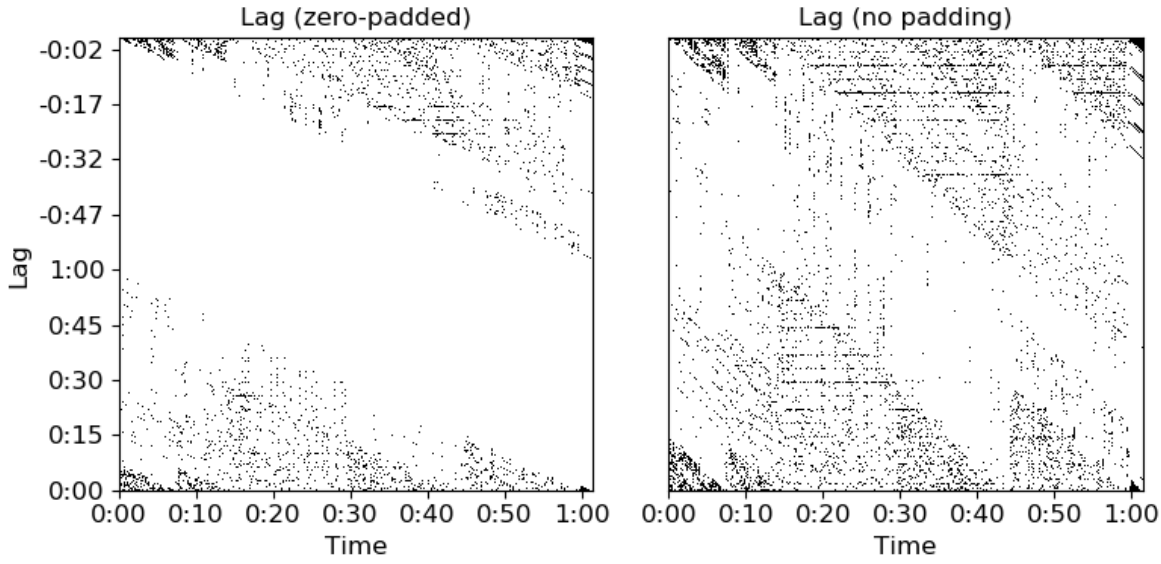
Raises: **ParameterError** : if *rec* is non-square

See also

recurrence_matrix
lag_to_recurrence
util.shear

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> hop_length = 1024
>>> mfccs = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length)
>>> recurrence = librosa.segment.recurrence_matrix(mfccs)
>>> lag_pad = librosa.segment.recurrence_to_lag(recurrence, pad=True)
>>> lag_nopad = librosa.segment.recurrence_to_lag(recurrence, pad=False)

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(1, 2, 1)
>>> librosa.display.specshow(lag_pad, x_axis='time', y_axis='lag',
...                          hop_length=hop_length)
>>> plt.title('Lag (zero-padded)')
>>> plt.subplot(1, 2, 2)
>>> librosa.display.specshow(lag_nopad, x_axis='time',
hop_length=hop_length)
>>> plt.title('Lag (no padding)')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.lag_to_recurrence

librosa.segment.lag_to_recurrence(*lag*, *axis=-1*)[source]

Convert a lag matrix into a recurrence matrix.

Parameters: lag : np.ndarray or scipy.sparse.spmatrix

A lag matrix, as produced by `recurrence_to_lag`

axis : int

The axis corresponding to the time dimension. The alternate axis will be interpreted in lag coordinates.

Returns: **rec** : np.ndarray or scipy.sparse.spmatrix [shape=(n, n)]

A recurrence matrix in (time, time) coordinates For sparse matrices, format will match that of *lag*.

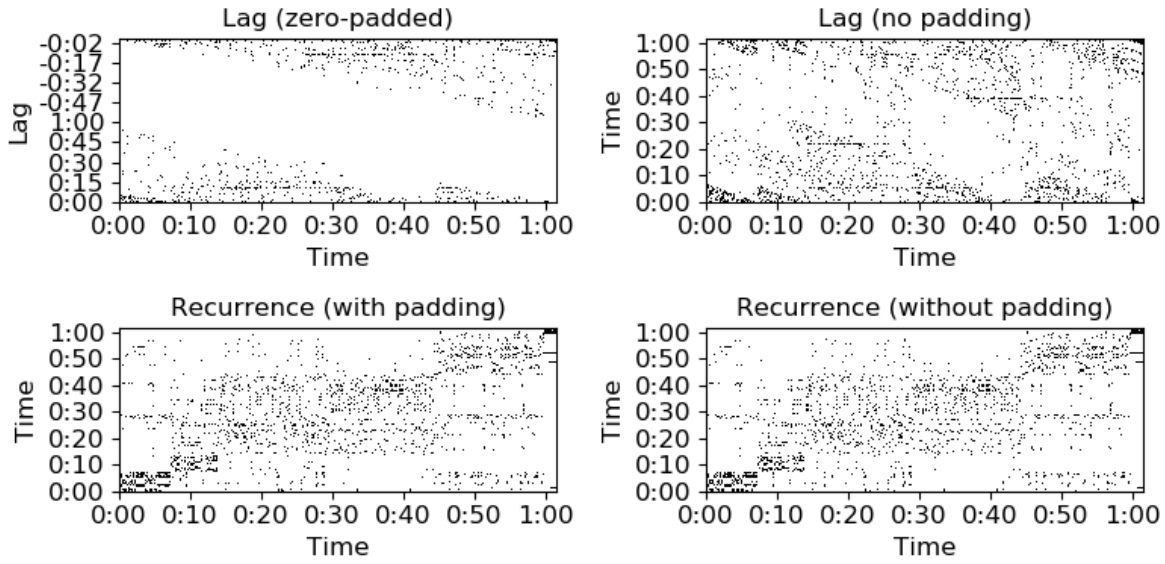
Raises: **ParameterError** : if *lag* does not have the correct shape

See also

`recurrence_to_lag`

Examples

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> hop_length = 1024
>>> mfccs = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length)
>>> recurrence = librosa.segment.recurrence_matrix(mfccs)
>>> lag_pad = librosa.segment.recurrence_to_lag(recurrence, pad=True)
>>> lag_nopad = librosa.segment.recurrence_to_lag(recurrence, pad=False)
>>> rec_pad = librosa.segment.lag_to_recurrence(lag_pad)
>>> rec_nopad = librosa.segment.lag_to_recurrence(lag_nopad)

>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(2, 2, 1)
>>> librosa.display.specshow(lag_pad, x_axis='time', y_axis='lag',
...                          hop_length=hop_length)
>>> plt.title('Lag (zero-padded)')
>>> plt.subplot(2, 2, 2)
>>> librosa.display.specshow(lag_nopad, x_axis='time', y_axis='time',
...                          hop_length=hop_length)
>>> plt.title('Lag (no padding)')
>>> plt.subplot(2, 2, 3)
>>> librosa.display.specshow(rec_pad, x_axis='time', y_axis='time',
...                          hop_length=hop_length)
>>> plt.title('Recurrence (with padding)')
>>> plt.subplot(2, 2, 4)
>>> librosa.display.specshow(rec_nopad, x_axis='time', y_axis='time',
...                          hop_length=hop_length)
>>> plt.title('Recurrence (without padding)')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.timelag_filter

`librosa.segment.timelag_filter`(*function*, *pad=True*, *index=0*)[source]

Filtering in the time-lag domain.

This is primarily useful for adapting image filters to operate on `recurrence_to_lag` output.

Using `timelag_filter` is equivalent to the following sequence of operations:

```
>>> data_tl = librosa.segment.recurrence_to_lag(data)
>>> data_filtered_tl = function(data_tl)
>>> data_filtered = librosa.segment.lag_to_recurrence(data_filtered_tl)
```

Parameters: **function** : callable

The filtering function to wrap, e.g., `scipy.ndimage.median_filter`

pad : bool

Whether to zero-pad the structure feature matrix

index : int >= 0

If *function* accepts input data as a positional argument, it should be indexed by *index*

Returns: **wrapped_function** : callable

A new filter function which applies in time-lag space rather than time-time space.

Examples

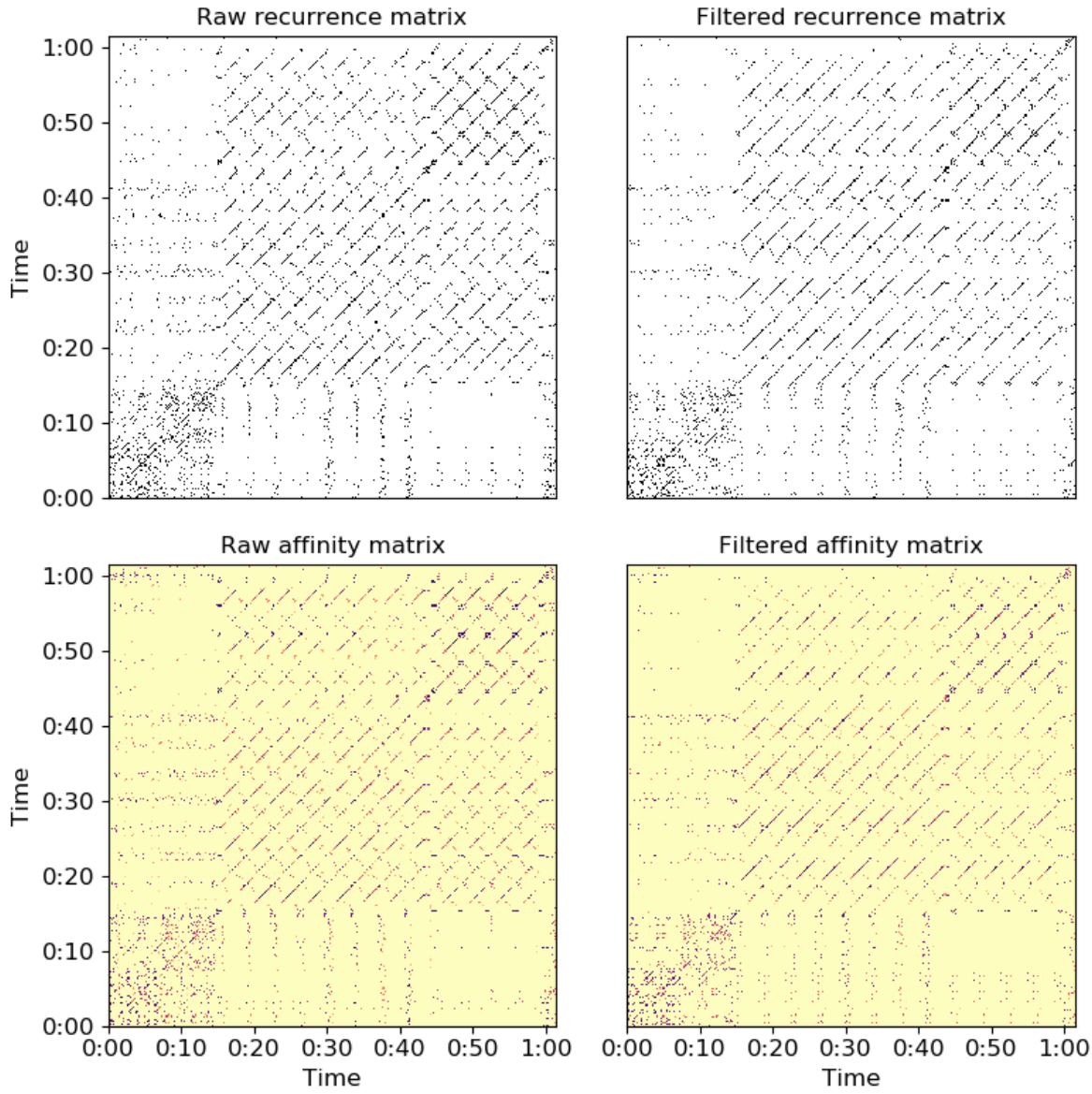
Apply a 5-bin median filter to the diagonal of a recurrence matrix

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr)
>>> rec = librosa.segment.recurrence_matrix(chroma)
>>> from scipy.ndimage import median_filter
>>> diagonal_median = librosa.segment.timelag_filter(median_filter)
>>> rec_filtered = diagonal_median(rec, size=(1, 3), mode='mirror')
```

Or with affinity weights

```
>>> rec_aff = librosa.segment.recurrence_matrix(chroma, mode='affinity')
>>> rec_aff_fil = diagonal_median(rec_aff, size=(1, 3), mode='mirror')
```

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8,8))
>>> plt.subplot(2, 2, 1)
>>> librosa.display.specshow(rec, y_axis='time')
>>> plt.title('Raw recurrence matrix')
>>> plt.subplot(2, 2, 2)
>>> librosa.display.specshow(rec_filtered)
>>> plt.title('Filtered recurrence matrix')
>>> plt.subplot(2, 2, 3)
>>> librosa.display.specshow(rec_aff, x_axis='time', y_axis='time',
...                          cmap='magma_r')
>>> plt.title('Raw affinity matrix')
>>> plt.subplot(2, 2, 4)
>>> librosa.display.specshow(rec_aff_fil, x_axis='time',
...                          cmap='magma_r')
>>> plt.title('Filtered affinity matrix')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.path_enhance

`librosa.segment.path_enhance`(*R*, *n*, *window='hann'*, *max_ratio=2.0*, *min_ratio=None*, *n_filters=7*, *zero_mean=False*, *clip=True*, ***kwargs*)[source]

Multi-angle path enhancement for self- and cross-similarity matrices.

This function convolves multiple diagonal smoothing filters with a self-similarity (or recurrence) matrix R, and aggregates the result by an element-wise maximum.

Technically, the output is a matrix R_smooth such that

*R_smooth[i, j] = max_theta (R * filter_theta)[i, j]*

where * denotes 2-dimensional convolution, and *filter_theta* is a smoothing filter at orientation theta.

This is intended to provide coherent temporal smoothing of self-similarity matrices when there are changes in tempo.

Smoothing filters are generated at evenly spaced orientations between min_ratio and max_ratio.

This function is inspired by the multi-angle path enhancement of [1], but differs by modeling tempo differences in the space of similarity matrices rather than re-sampling the underlying features prior to generating the self-similarity matrix.

[1] Müller, Meinard and Frank Kurth. “Enhancing similarity matrices for music audio analysis.” 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings. Vol. 5. IEEE, 2006.

Note

if using recurrence_matrix to construct the input similarity matrix, be sure to include the main diagonal by setting *self=True*. Otherwise, the diagonal will be suppressed, and this is likely to produce discontinuities which will pollute the smoothing filter response.

Parameters: R : np.ndarray

The self- or cross-similarity matrix to be smoothed. Note: sparse inputs are not supported.

n : int > 0

The length of the smoothing filter

window : window specification

The type of smoothing filter to use. See *filters.get_window* for more information on window specification formats.

max_ratio : float > 0

The maximum tempo ratio to support

min_ratio : float > 0

The minimum tempo ratio to support. If not provided, it will default to *1/max_ratio*

n_filters : int >= 1

The number of different smoothing filters to use, evenly spaced between *min_ratio* and *max_ratio*.

If *min_ratio = 1/max_ratio* (the default), using an odd number of filters will ensure that the main diagonal (ratio=1) is included.

zero_mean : bool

By default, the smoothing filters are non-negative and sum to one (i.e. are averaging filters).

If *zero_mean=True*, then the smoothing filters are made to sum to zero by subtracting a constant value from the non-diagonal coordinates of the filter. This is primarily useful for suppressing blocks while enhancing diagonals.

clip : bool

If True, the smoothed similarity matrix will be thresholded at 0, and will not contain negative entries.

kwargs : additional keyword arguments

Additional arguments to pass to `scipy.ndimage.convolve`

Returns: **R_smooth** : np.ndarray, shape=R.shape

The smoothed self- or cross-similarity matrix

See also

`filters.diagonal_filter`
`recurrence_matrix`

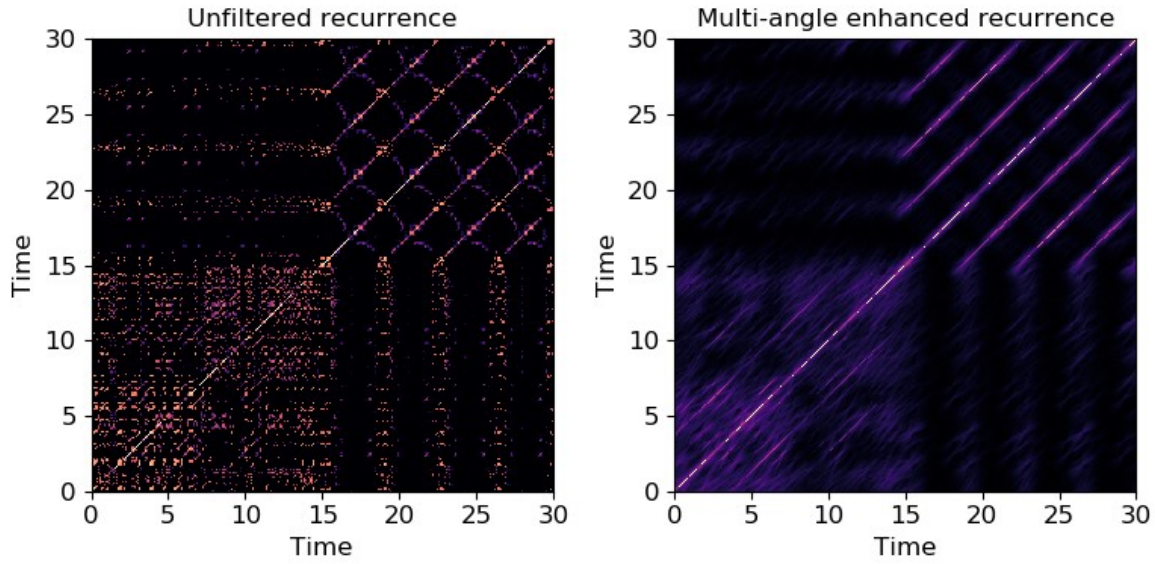
Examples

Use a 51-frame diagonal smoothing filter to enhance paths in a recurrence matrix

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=30)
>>> hop_length = 1024
>>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr, hop_length=hop_length)
>>> rec = librosa.segment.recurrence_matrix(chroma, mode='affinity',
self=True)
>>> rec_smooth = librosa.segment.path_enhance(rec, 51, window='hann',
n_filters=7)
```

Plot the recurrence matrix before and after smoothing

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8, 4))
>>> plt.subplot(1,2,1)
>>> librosa.display.specshow(rec, x_axis='time', y_axis='time',
...                          hop_length=hop_length)
>>> plt.title('Unfiltered recurrence')
>>> plt.subplot(1,2,2)
>>> librosa.display.specshow(rec_smooth, x_axis='time', y_axis='time',
...                          hop_length=hop_length)
>>> plt.title('Multi-angle enhanced recurrence')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.agglomerative

`librosa.segment.agglomerative`(*data*, *k*, *clusterer=None*, *axis=-1*)[source]

Bottom-up temporal segmentation.

Use a temporally-constrained agglomerative clustering routine to partition *data* into *k* contiguous segments.

Parameters: **data** : np.ndarray

data to cluster

k : int > 0 [scalar]

number of segments to produce

clusterer : sklearn.cluster.AgglomerativeClustering, optional

An optional AgglomerativeClustering object. If *None*, a constrained Ward object is instantiated.

axis : int

axis along which to cluster. By default, the last axis (-1) is chosen.

Returns: **boundaries** : np.ndarray [shape=(k,)]

left-boundaries (frame numbers) of detected segments. This will always include *0* as the first left-boundary.

See also

`sklearn.cluster.AgglomerativeClustering`

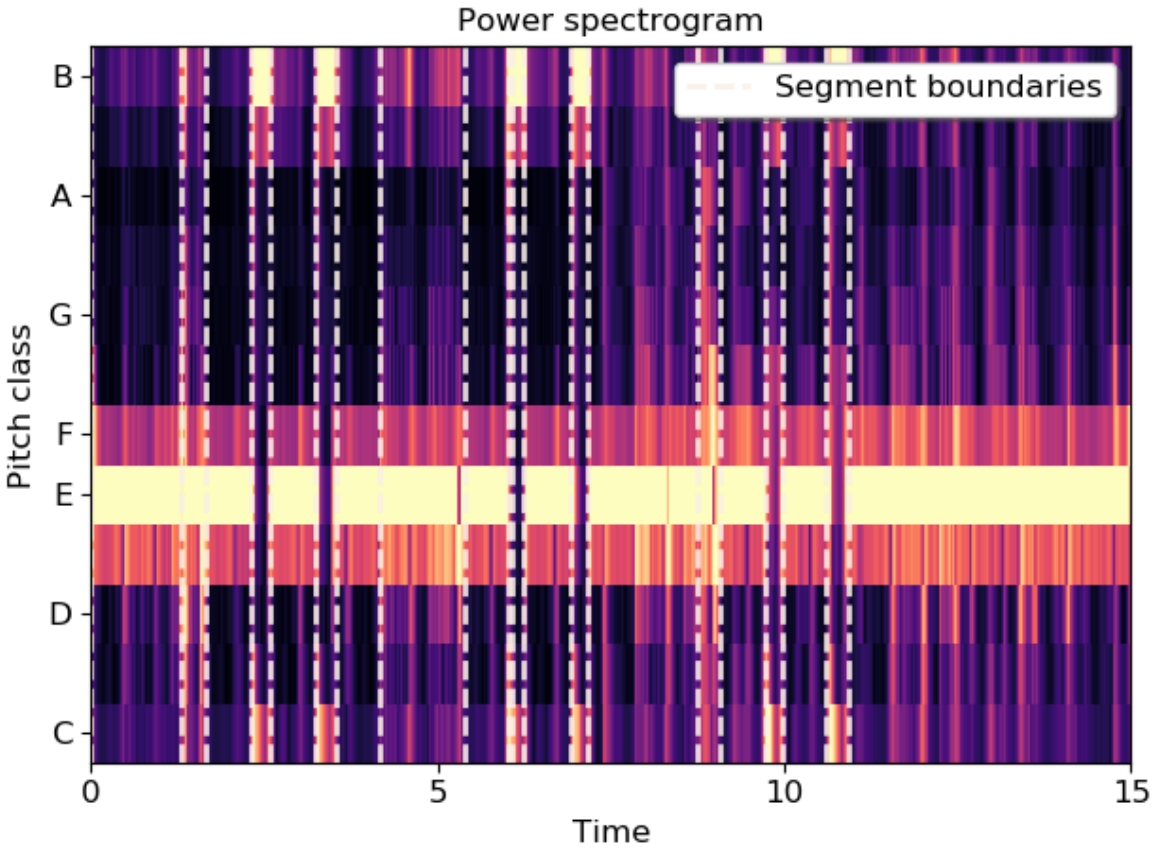
Examples

Cluster by chroma similarity, break into 20 segments

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=15)
>>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr)
>>> bounds = librosa.segment.agglomerative(chroma, 20)
>>> bound_times = librosa.frames_to_time(bounds, sr=sr)
>>> bound_times
array([  0.   ,   1.672,   2.322,   2.624,   3.251,   3.506,
         4.18 ,   5.387,   6.014,   6.293,   6.943,   7.198,
         7.848,   9.033,   9.706,   9.961,  10.635,  10.89 ,
        11.54 ,  12.539])
```

Plot the segmentation over the chromagram

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> librosa.display.specshow(chroma, y_axis='chroma', x_axis='time')
>>> plt.vlines(bound_times, 0, chroma.shape[0], color='linen',
linestyle='--',
...            linewidth=2, alpha=0.9, label='Segment boundaries')
>>> plt.axis('tight')
>>> plt.legend(frameon=True, shadow=True)
>>> plt.title('Power spectrogram')
>>> plt.tight_layout()
>>> plt.show()
```



librosa.segment.subsegment

`librosa.segment.subsegment`(*data, frames, n_segments=4, axis=-1*)[source]

Sub-divide a segmentation by feature clustering.

Given a set of frame boundaries (*frames*), and a data matrix (*data*), each successive interval defined by *frames* is partitioned into *n_segments* by constrained agglomerative clustering.

Note

If an interval spans fewer than *n_segments* frames, then each frame becomes a sub-segment.

Parameters: data : np.ndarray

Data matrix to use in clustering

frames : np.ndarray [shape=(n_boundaries,)], dtype=int, non-negative]

Array of beat or segment boundaries, as provided by `librosa.beat.beat_track`, `librosa.onset.onset_detect`, or `agglomerative`.

n_segments : int > 0

Maximum number of frames to sub-divide each interval.

axis : int

Axis along which to apply the segmentation. By default, the last index (-1) is taken.

Returns: **boundaries** : np.ndarray [shape=(n_subboundaries,)]

List of sub-divided segment boundaries

See also

`agglomerative`
Temporal segmentation

`librosa.onset.onset_detect`
Onset detection

`librosa.beat.beat_track`
Beat tracking

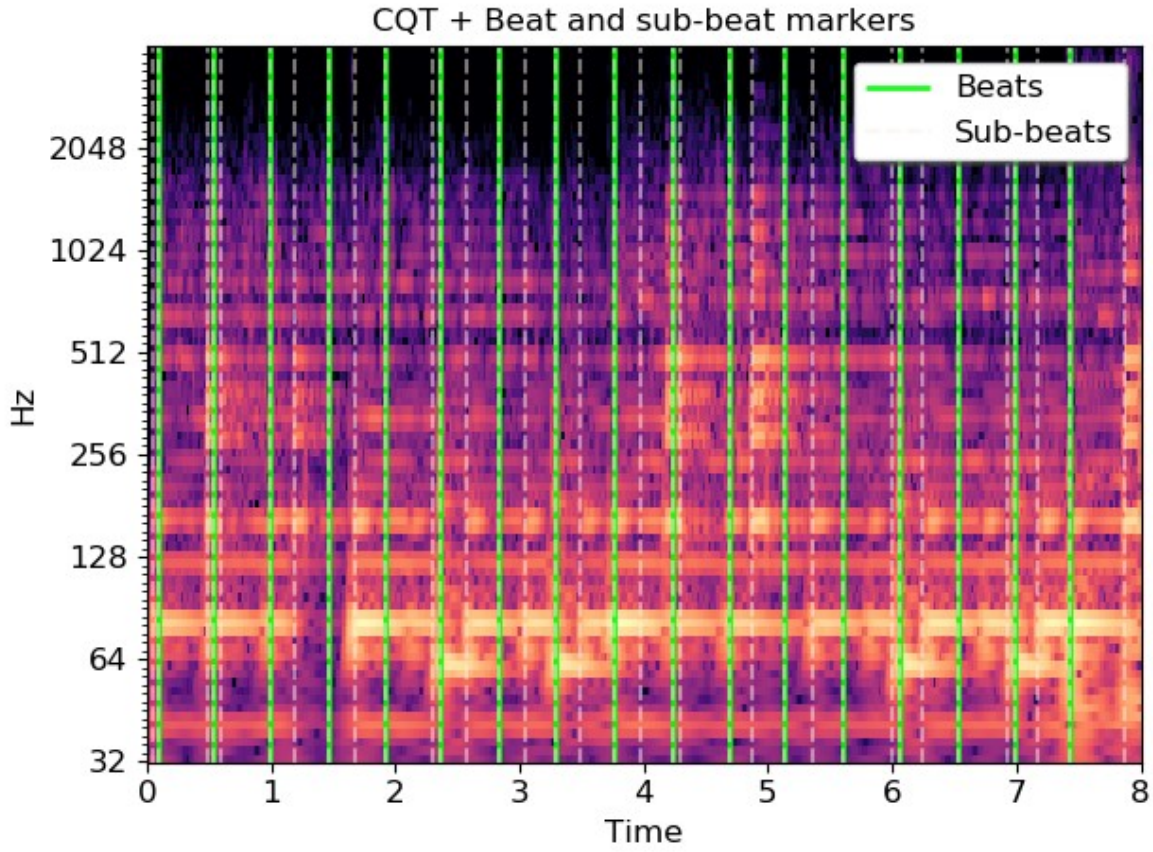
Notes

This function caches at level 30.

Examples

Load audio, detect beat frames, and subdivide in twos by CQT

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), duration=8)
>>> tempo, beats = librosa.beat.beat_track(y=y, sr=sr, hop_length=512)
>>> beat_times = librosa.frames_to_time(beats, sr=sr, hop_length=512)
>>> cqt = np.abs(librosa.cqt(y, sr=sr, hop_length=512))
>>> subseg = librosa.segment.subsegment(cqt, beats, n_segments=2)
>>> subseg_t = librosa.frames_to_time(subseg, sr=sr, hop_length=512)
>>> subseg
array([  0,   2,   4,  21,  23,  26,  43,  55,  63,  72,  83,
        97, 102, 111, 122, 137, 142, 153, 162, 180, 182, 185,
       202, 210, 221, 231, 241, 256, 261, 271, 281, 296, 301,
       310, 320, 339, 341, 344, 361, 368, 382, 389, 401, 416,
       420, 430, 436, 451, 456, 465, 476, 489, 496, 503, 515,
       527, 535, 544, 553, 558, 571, 578, 590, 607, 609, 638])

>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> librosa.display.specshow(librosa.amplitude_to_db(cqt,
...                                                  ref=np.max),
...                          y_axis='cqt_hz', x_axis='time')
>>> lims = plt.gca().get_ylim()
>>> plt.vlines(beat_times, lims[0], lims[1], color='lime', alpha=0.9,
...            linewidth=2, label='Beats')
>>> plt.vlines(subseg_t, lims[0], lims[1], color='linen', linestyle='--',
...            linewidth=1.5, alpha=0.5, label='Sub-beats')
>>> plt.legend(frameon=True, shadow=True)
>>> plt.title('CQT + Beat and sub-beat markers')
>>> plt.tight_layout()
>>> plt.show()
```



Sequential modeling

Sequence alignment

Viterbi decoding

viterbi(prob, transition[, p_init, return_logp])	Viterbi decoding from observation likelihoods.
viterbi_discriminative(prob, transition[, …])	Viterbi decoding from discriminative state predictions.
viterbi_binary(prob, transition[, p_state, …])	Viterbi decoding from binary (multi-label), discriminative state predictions.

Transition matrices

transition_uniform(n_states)	Construct a uniform transition matrix over *n_states*.
transition_loop(n_states, prob)	Construct a self-loop transition matrix over *n_states*.

transition_cycle(n_states, prob)	Construct a cyclic transition matrix over *n_states*.
transition_local(n_states, width[, window, wrap])	Construct a localized transition matrix.

librosa.sequence.dtw

`librosa.sequence.dtw`(*X=None, Y=None, C=None, metric='euclidean', step_sizes_sigma=None, weights_add=None, weights_mul=None, subseq=False, backtrack=True, global_constraints=False, band_rad=0.25*)[source]

Dynamic time warping (DTW).

This function performs a DTW and path backtracking on two sequences. We follow the nomenclature and algorithmic approach as described in [1].

[1] Meinard Mueller Fundamentals of Music Processing — Audio, Analysis, Algorithms, Applications Springer Verlag, ISBN: 978-3-319-21944-8, 2015.

Parameters: **X** : np.ndarray [shape=(K, N)]

audio feature matrix (e.g., chroma features)

Y : np.ndarray [shape=(K, M)]

audio feature matrix (e.g., chroma features)

C : np.ndarray [shape=(N, M)]

Precomputed distance matrix. If supplied, X and Y must not be supplied and `metric` will be ignored.

metric : str

Identifier for the cost-function as documented in *scipy.spatial.cdist()*

step_sizes_sigma : np.ndarray [shape=[n, 2]]

Specifies allowed step sizes as used by the dtw.

weights_add : np.ndarray [shape=[n,]]

Additive weights to penalize certain step sizes.

weights_mul : np.ndarray [shape=[n,]]

Multiplicative weights to penalize certain step sizes.

subseq : binary

Enable subsequence DTW, e.g., for retrieval tasks.

backtrack : binary

Enable backtracking in accumulated cost matrix.

global_constraints : binary

Applies global constraints to the cost matrix `C` (Sakoe-Chiba band).

band_rad : float

The Sakoe-Chiba band radius (1/2 of the width) will be `int(radius*min(C.shape))`.

Returns: **D** : np.ndarray [shape=(N,M)]

accumulated cost matrix. D[N,M] is the total alignment cost. When doing subsequence DTW, D[N,:] indicates a matching function.

wp : np.ndarray [shape=(N,2)]

Warping path with index pairs. Each row of the array contains an index pair n,m). Only returned when `backtrack` is True.

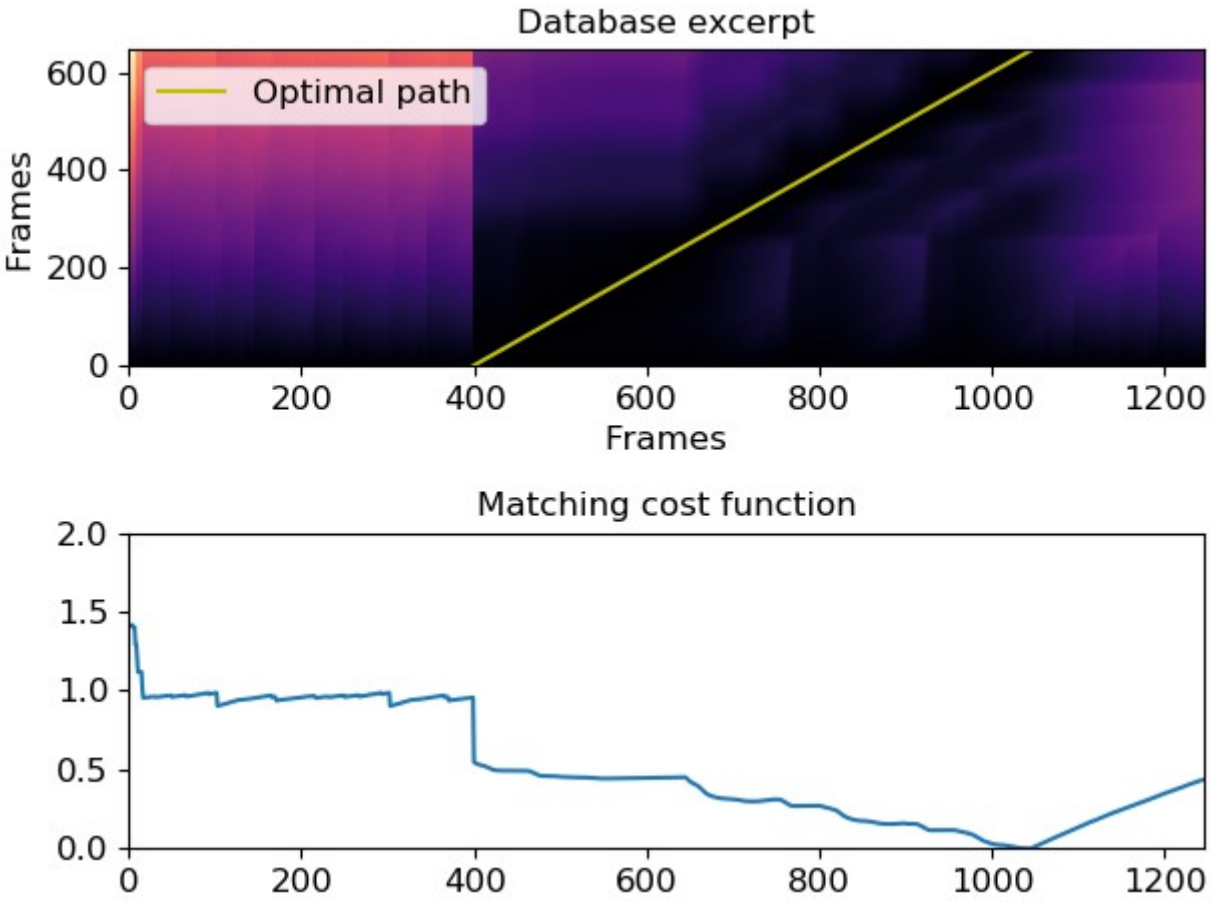
Raises: ParameterError

If you are doing diagonal matching and Y is shorter than X or if an incompatible combination of X, Y, and C are supplied. If your input dimensions are incompatible.

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(), offset=10,
duration=15)
>>> X = librosa.feature.chroma_cens(y=y, sr=sr)
>>> noise = np.random.rand(X.shape[0], 200)
>>> Y = np.concatenate((noise, noise, X, noise), axis=1)
>>> D, wp = librosa.sequence.dtw(X, Y, subseq=True)
>>> plt.subplot(2, 1, 1)
>>> librosa.display.specshow(D, x_axis='frames', y_axis='frames')
>>> plt.title('Database excerpt')
>>> plt.plot(wp[:, 1], wp[:, 0], label='Optimal path', color='y')
>>> plt.legend()
>>> plt.subplot(2, 1, 2)
>>> plt.plot(D[-1, :] / wp.shape[0])
>>> plt.xlim([0, Y.shape[1]])
>>> plt.ylim([0, 2])
>>> plt.title('Matching cost function')
```

```
>>> plt.tight_layout()
>>> plt.show()
```



librosa.sequence.rqa

`librosa.sequence.rqa`(*sim*, *gap_onset=1*, *gap_extend=1*, *knight_moves=True*, *backtrack=True*)[source]

Recurrence quantification analysis (RQA)

This function implements different forms of RQA as described by Serra, Serra, and Andrzejak [1]. These methods take as input a self- or cross-similarity matrix *sim*, and calculate the value of path alignments by dynamic programming.

Note that unlike dynamic time warping (`dtw`), alignment paths here are maximized, not minimized, so the input should measure similarity rather than distance.

The simplest RQA method, denoted as *L* [1] (equation 3) and equivalent to the method described by Eckman, Kamphorst, and Ruelle [2], accumulates the length of diagonal paths with positive values in the input:

- *score[i, j] = score[i-1, j-1] + 1* if *sim[i, j] > 0*
- *score[i, j] = 0* otherwise.

The second method, denoted as *S* [1] (equation 4), is similar to the first, but allows for “knight moves” (as in the chess piece) in addition to strict diagonal moves:

- *score[i, j] = max(score[i-1, j-1], score[i-2, j-1], score[i-1, j-2]) + 1* if *sim[i, j] > 0*
- *score[i, j] = 0* otherwise.

The third method, denoted as *Q* [1] (equations 5 and 6) extends this by allowing gaps in the alignment that incur some cost, rather than a hard reset to 0 whenever *sim[i, j] == 0*. Gaps are penalized by two additional parameters, *gap_onset* and *gap_extend*, which are subtracted from the value of the alignment path every time a gap is introduced or extended (respectively).

Note that setting *gap_onset* and *gap_extend* to `np.inf` recovers the second method, and disabling knight moves recovers the first.

[1] *(1, 2, 3, 4)* Serrà, Joan, Xavier Serra, and Ralph G. Andrzejak. “Cross recurrence quantification for cover song identification.” New Journal of Physics 11, no. 9 (2009): 093017.

[2] Eckmann, J. P., S. Oliffson Kamphorst, and D. Ruelle. “Recurrence plots of dynamical systems.” World Scientific Series on Nonlinear Science Series A 16 (1995): 441-446.

Parameters: **sim** : np.ndarray [shape=(N, M), non-negative]

The similarity matrix to use as input.

This can either be a recurrence matrix (self-similarity) or a cross-similarity matrix between two sequences.

gap_onset : float > 0

Penalty for introducing a gap to an alignment sequence

gap_extend : float > 0

Penalty for extending a gap in an alignment sequence

knight_moves : bool

If *True* (default), allow for “knight moves” in the alignment, e.g., *(n, m) => (n + 1, m + 2)* or *(n + 2, m + 1)*.

If *False*, only allow for diagonal moves *(n, m) => (n + 1, m + 1)*.

backtrack : bool

If *True*, return the alignment path.

If *False*, only return the score matrix.

Returns: **score** : np.ndarray [shape=(N, M)]

The alignment score matrix. *score[n, m]* is the cumulative value of the best alignment sequence ending in frames *n* and *m*.

path : np.ndarray [shape=(k, 2)] (optional)

If *backtrack=True*, *path* contains a list of pairs of aligned frames in the best alignment sequence.

path[i] = [n, m] indicates that row *n* aligns to column *m*.

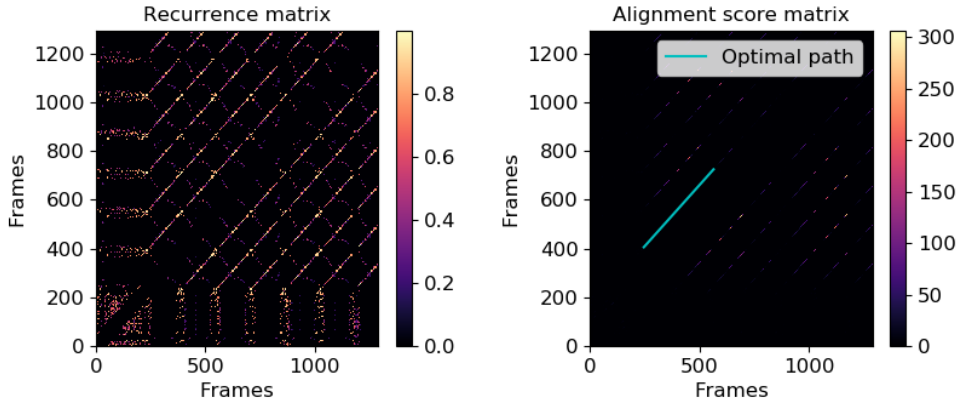
See also

`segment.recurrence_matrix`
`segment.cross_similarity`
`dtw`

Examples

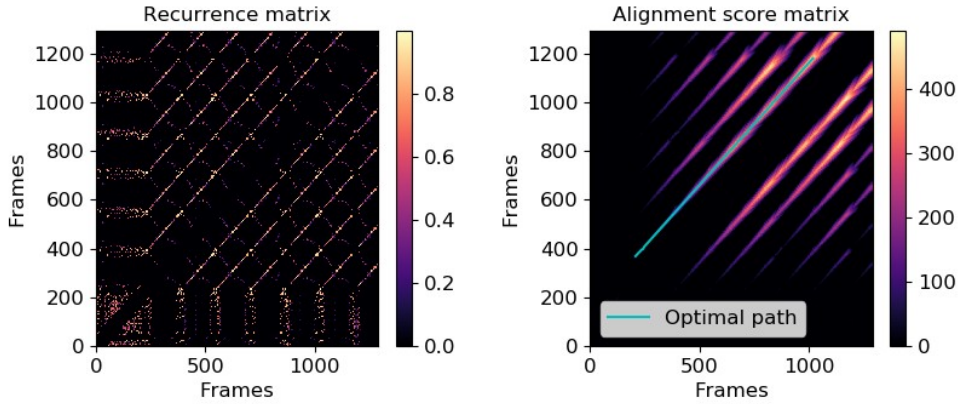
Simple diagonal path enhancement (L-mode)

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10, duration=30)
>>> chroma = librosa.feature.chroma_cqt(y=y, sr=sr)
>>> # Use time-delay embedding to reduce noise
>>> chroma_stack = librosa.feature.stack_memory(chroma, n_steps=3)
>>> # Build recurrence, suppress self-loops within 1 second
>>> rec = librosa.segment.recurrence_matrix(chroma_stack, width=43,
...                                         mode='affinity',
...                                         metric='cosine')
>>> # using infinite cost for gaps enforces strict path continuation
>>> L_score, L_path = librosa.sequence.rqa(rec, np.inf, np.inf,
...                                        knight_moves=False)
>>> plt.figure(figsize=(10, 4))
>>> plt.subplot(1,2,1)
>>> librosa.display.specshow(rec, x_axis='frames', y_axis='frames')
>>> plt.title('Recurrence matrix')
>>> plt.colorbar()
>>> plt.subplot(1,2,2)
>>> librosa.display.specshow(L_score, x_axis='frames', y_axis='frames')
>>> plt.title('Alignment score matrix')
>>> plt.colorbar()
>>> plt.plot(L_path[:, 1], L_path[:, 0], label='Optimal path', color='c')
>>> plt.legend()
>>> plt.show()
```



Full alignment using gaps and knight moves

```
>>> # New gaps cost 5, extending old gaps cost 10 for each step
>>> score, path = librosa.sequence.rqa(rec, 5, 10)
>>> plt.figure(figsize=(10, 4))
>>> plt.subplot(1,2,1)
>>> librosa.display.specshow(rec, x_axis='frames', y_axis='frames')
>>> plt.title('Recurrence matrix')
>>> plt.colorbar()
>>> plt.subplot(1,2,2)
>>> librosa.display.specshow(score, x_axis='frames', y_axis='frames')
>>> plt.title('Alignment score matrix')
>>> plt.plot(path[:, 1], path[:, 0], label='Optimal path', color='c')
>>> plt.colorbar()
>>> plt.legend()
>>> plt.show()
```



librosa.sequence.viterbi

`librosa.sequence.viterbi`(*prob*, *transition*, *p_init=None*, *return_logp=False*)[source]

Viterbi decoding from observation likelihoods.

Given a sequence of observation likelihoods *prob[s, t]*, indicating the conditional likelihood of seeing the observation at time *t* from state *s*, and a transition matrix *transition[i, j]* which encodes the conditional probability of moving from state *i* to state *j*, the Viterbi algorithm [1] computes the most likely sequence of states from the observations.

[1] Viterbi, Andrew. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm.” IEEE transactions on Information Theory 13.2 (1967): 260-269.

Parameters: **prob** : np.ndarray [shape=(n_states, n_steps), non-negative]

prob[s, t] is the probability of observation at time *t* being generated by state *s*.

transition : np.ndarray [shape=(n_states, n_states), non-negative]

transition[i, j] is the probability of a transition from i->j. Each row must sum to 1.

p_init : np.ndarray [shape=(n_states,)]

Optional: initial state distribution. If not provided, a uniform distribution is assumed.

return_logp : bool

If *True*, return the log-likelihood of the state sequence.

Returns: Either *states* or *(states, logp)*:
states : np.ndarray [shape=(n_steps,)]

The most likely state sequence.

logp : scalar [float]

If *return_logp=True*, the log probability of *states* given the observations.

See also

`viterbi_discriminative`
Viterbi decoding from state likelihoods

Examples

Example from https://en.wikipedia.org/wiki/Viterbi_algorithm#Example

In this example, we have two states `healthy` and `fever`, with initial probabilities 60% and 40%.

We have three observation possibilities: `normal`, `cold`, and `dizzy`, whose probabilities given each state are:

`healthy => {normal: 50%, cold: 40%, dizzy: 10%}` and `fever => {normal: 10%, cold: 30%, dizzy: 60%}`

Finally, we have transition probabilities:

`healthy => healthy (70%)` and `fever => fever (60%)`.

Over three days, we observe the sequence `[normal, cold, dizzy]`, and wish to know the maximum likelihood assignment of states for the corresponding days, which we compute with the Viterbi algorithm below.

```
>>> p_init = np.array([0.6, 0.4])
>>> p_emit = np.array([[0.5, 0.4, 0.1],
...                    [0.1, 0.3, 0.6]])
>>> p_trans = np.array([[0.7, 0.3], [0.4, 0.6]])
>>> path, logp = librosa.sequence.viterbi(p_emit, p_trans, p_init,
...                                       return_logp=True)
>>> print(logp, path)
-4.19173690823075 [0 0 1]
```

librosa.sequence.viterbi_discriminative

`librosa.sequence.viterbi_discriminative`(*prob*, *transition*, *p_state=None*, *p_init=None*, *return_logp=False*)[source]

Viterbi decoding from discriminative state predictions.

Given a sequence of conditional state predictions *prob[s, t]*, indicating the conditional likelihood of state *s* given the observation at time *t*, and a transition matrix *transition[i, j]* which encodes the conditional probability of moving from state *i* to state *j*, the Viterbi algorithm computes the most likely sequence of states from the observations.

This implementation uses the standard Viterbi decoding algorithm for observation likelihood sequences, under the assumption that *P[Obs(t) | State(t) = s]* is proportional to *P[State(t) = s | Obs(t)] / P[State(t) = s]*, where the denominator is the marginal probability of state *s* occurring as given by *p_state*.

Parameters: **prob** : np.ndarray [shape=(n_states, n_steps), non-negative]

prob[s, t] is the probability of state *s* conditional on the observation at time *t*. Must be non-negative and sum to 1 along each column.

transition : np.ndarray [shape=(n_states, n_states), non-negative]

transition[i, j] is the probability of a transition from i->j. Each row must sum to 1.

p_state : np.ndarray [shape=(n_states,)]

Optional: marginal probability distribution over states, must be non-negative and sum to 1. If not provided, a uniform distribution is assumed.

p_init : np.ndarray [shape=(n_states,)]

Optional: initial state distribution. If not provided, it is assumed to be uniform.

return_logp : bool

If *True*, return the log-likelihood of the state sequence.

Returns: Either *states* or *(states, logp)*:
states : np.ndarray [shape=(n_steps,)]

The most likely state sequence.

logp : scalar [float]

If *return_logp=True*, the log probability of *states* given the observations.

See also

`viterbi`
Viterbi decoding from observation likelihoods
`viterbi_binary`
Viterbi decoding for multi-label, conditional state likelihoods

Examples

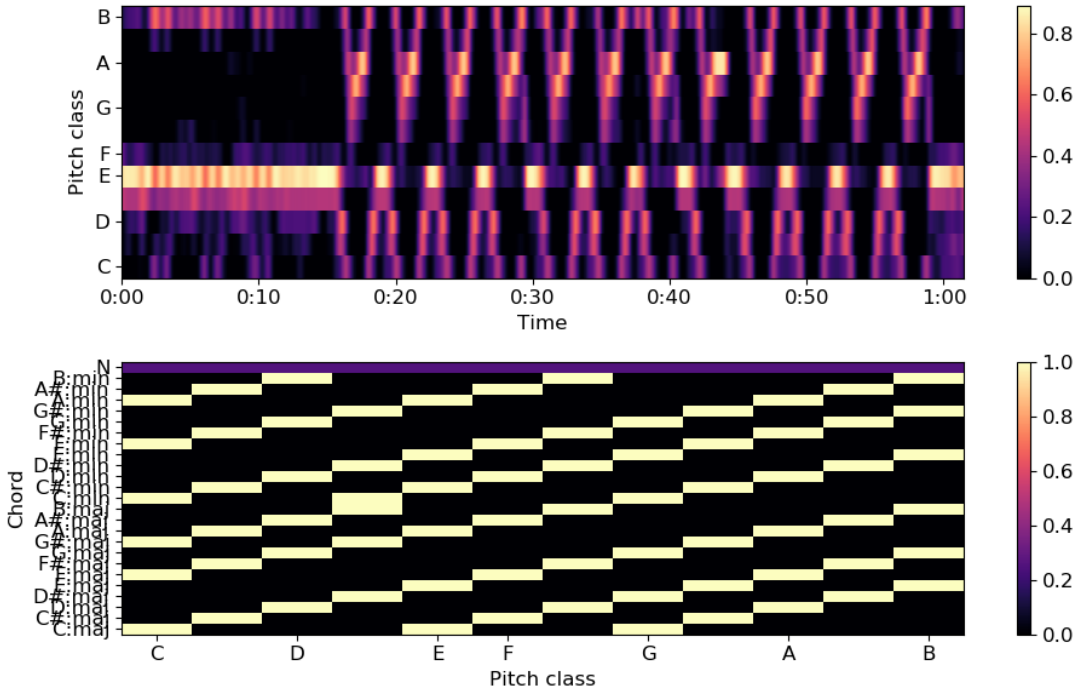
This example constructs a simple, template-based discriminative chord estimator, using CENS chroma as input features.

Note

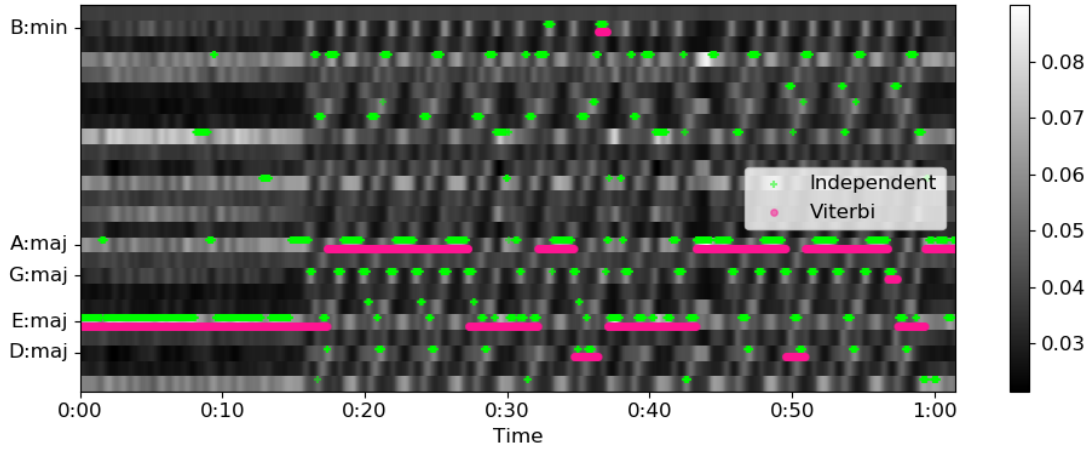
this chord model is not accurate enough to use in practice. It is only intended to demonstrate how to use discriminative Viterbi decoding.

```
>>> # Create templates for major, minor, and no-chord qualities
>>> maj_template = np.array([1,0,0, 0,1,0, 0,1,0, 0,0,0])
>>> min_template = np.array([1,0,0, 1,0,0, 0,1,0, 0,0,0])
>>> N_template   = np.array([1,1,1, 1,1,1, 1,1,1, 1,1,1.]) / 4.
>>> # Generate the weighting matrix that maps chroma to labels
>>> weights = np.zeros((25, 12), dtype=float)
>>> labels = ['C:maj', 'C#:maj', 'D:maj', 'D#:maj', 'E:maj', 'F:maj',
...           'F#:maj', 'G:maj', 'G#:maj', 'A:maj', 'A#:maj', 'B:maj',
...           'C:min', 'C#:min', 'D:min', 'D#:min', 'E:min', 'F:min',
...           'F#:min', 'G:min', 'G#:min', 'A:min', 'A#:min', 'B:min',
...           'N']
>>> for c in range(12):
...     weights[c, :] = np.roll(maj_template, c) # c:maj
...     weights[c + 12, :] = np.roll(min_template, c)  # c:min
>>> weights[-1] = N_template  # the last row is the no-chord class
>>> # Make a self-loop transition matrix over 25 states
>>> trans = librosa.sequence.transition_loop(25, 0.9)

>>> # Load in audio and make features
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> # Suppress percussive elements
>>> y = librosa.effects.harmonic(y, margin=4)
>>> chroma = librosa.feature.chroma_cens(y=y, sr=sr, bins_per_octave=36)
>>> # Map chroma (observations) to class (state) likelihoods
>>> probs = np.exp(weights.dot(chroma))  # P[class | chroma] ~=
exp(template' chroma)
>>> probs /= probs.sum(axis=0, keepdims=True)  # probabilities must sum to
1 in each column
>>> # Compute independent frame-wise estimates
>>> chords_ind = np.argmax(probs, axis=0)
```

```
>>> # And viterbi estimates
>>> chords_vit = librosa.sequence.viterbi_discriminative(probs, trans)

>>> # Plot the features and prediction map
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(10, 6))
>>> plt.subplot(2,1,1)
>>> librosa.display.specshow(chroma, x_axis='time', y_axis='chroma')
>>> plt.colorbar()
>>> plt.subplot(2,1,2)
>>> librosa.display.specshow(weights, x_axis='chroma')
>>> plt.yticks(np.arange(25) + 0.5, labels)
>>> plt.ylabel('Chord')
>>> plt.colorbar()
>>> plt.tight_layout()
>>> plt.show()
```




```
>>> # And plot the results

>>> plt.figure(figsize=(10, 4))
>>> librosa.display.specshow(probs, x_axis='time', cmap='gray')
>>> plt.colorbar()
>>> times = librosa.times_like(chords_vit)
>>> plt.scatter(times, chords_ind + 0.75, color='lime', alpha=0.5, marker='+',
...             s=15, label='Independent')
>>> plt.scatter(times, chords_vit + 0.25, color='deeppink', alpha=0.5,
marker='o',
...             s=15, label='Viterbi')
>>> plt.yticks(0.5 + np.unique(chords_vit),
...            [labels[i] for i in np.unique(chords_vit)], va='center')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



librosa.sequence.viterbi_binary

`librosa.sequence.viterbi_binary`(*prob*, *transition*, *p_state=None*, *p_init=None*, *return_logp=False*)[source]

Viterbi decoding from binary (multi-label), discriminative state predictions.

Given a sequence of conditional state predictions *prob[s, t]*, indicating the conditional likelihood of state *s* being active conditional on observation at time *t*, and a 2*2 transition matrix *transition* which encodes the conditional probability of moving from state *s* to state *s* (not-*s*), the Viterbi algorithm computes the most likely sequence of states from the observations.

This function differs from `viterbi_discriminative` in that it does not assume the states to be mutually exclusive. `viterbi_binary` is implemented by transforming the multi-label decoding problem to a collection of binary Viterbi problems (one for each *state* or label).

The output is a binary matrix *states[s, t]* indicating whether each state *s* is active at time *t*.

Parameters: **prob** : np.ndarray [shape=(n_steps,) or (n_states, n_steps)], non-negative

prob[s, t] is the probability of state *s* being active conditional on the observation at time *t*. Must be non-negative and less than 1.

If *prob* is 1-dimensional, it is expanded to shape *(1, n_steps)*.

transition : np.ndarray [shape=(2, 2) or (n_states, 2, 2)], non-negative

If 2-dimensional, the same transition matrix is applied to each sub-problem. *transition[0, i]* is the probability of the state going from inactive to *i*, *transition[1, i]* is the probability of the state going from active to *i*. Each row must sum to 1.

If 3-dimensional, *transition[s]* is interpreted as the 2x2 transition matrix for state label *s*.

p_state : np.ndarray [shape=(n_states,)]

Optional: marginal probability for each state (between [0,1]). If not provided, a uniform distribution (0.5 for each state) is assumed.

p_init : np.ndarray [shape=(n_states,)]

Optional: initial state distribution. If not provided, it is assumed to be uniform.

return_logp : bool

If *True*, return the log-likelihood of the state sequence.

Returns: Either *states* or *(states, logp)*:
states : np.ndarray [shape=(n_states, n_steps)]

The most likely state sequence.

logp : np.ndarray [shape=(n_states,)]

If *return_logp=True*, the log probability of each state activation sequence *states*

See also

`viterbi`
Viterbi decoding from observation likelihoods
`viterbi_discriminative`
Viterbi decoding for discriminative (mutually exclusive) state predictions

Examples

In this example, we have a sequence of binary state likelihoods that we want to de-noise under the assumption that state changes are relatively uncommon. Positive predictions should only be retained if they persist for multiple steps, and any transient predictions should be considered as errors. This use case arises frequently in problems such as instrument recognition, where state activations tend to be stable over time, but subject to abrupt changes (e.g., when an instrument joins the mix).

We assume that the 0 state has a self-transition probability of 90%, and the 1 state has a self-transition probability of 70%. We assume the marginal and initial probability of either state is 50%.

```
>>> trans = np.array([[0.9, 0.1], [0.3, 0.7]])
>>> prob = np.array([0.1, 0.7, 0.4, 0.3, 0.8, 0.9, 0.8, 0.2, 0.6, 0.3])
>>> librosa.sequence.viterbi_binary(prob, trans, p_state=0.5, p_init=0.5)
array([[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]])
```

librosa.sequence.transition_uniform

`librosa.sequence.transition_uniform`(*n_states*)[source]

Construct a uniform transition matrix over *n_states*.

Parameters: **n_states** : int > 0

The number of states

Returns: **transition** : np.ndarray [shape=(n_states, n_states)]

transition[i, j] = 1./n_states

Examples

```
>>> librosa.sequence.transition_uniform(3)
array([[0.333, 0.333, 0.333],
       [0.333, 0.333, 0.333],
       [0.333, 0.333, 0.333]])
```

librosa.sequence.transition_loop

`librosa.sequence.transition_loop`(*n_states*, *prob*)[source]

Construct a self-loop transition matrix over *n_states*.

The transition matrix will have the following properties:

- *transition[i, i] = p* for all i
- *transition[i, j] = (1 - p) / (n_states - 1)* for all *j != i*

This type of transition matrix is appropriate when states tend to be locally stable, and there is no additional structure between different states. This is primarily useful for de-noising frame-wise predictions.

Parameters: **n_states** : int > 1

The number of states

prob : float in [0, 1] or iterable, length=n_states

If a scalar, this is the probability of a self-transition.

If a vector of length *n_states*, *p[i]* is the probability of state *i*'s self-transition.

Returns: **transition** : np.ndarray [shape=(n_states, n_states)]

The transition matrix

Examples

```
>>> librosa.sequence.transition_loop(3, 0.5)
array([[0.5 , 0.25, 0.25],
       [0.25, 0.5 , 0.25],
       [0.25, 0.25, 0.5 ]])

>>> librosa.sequence.transition_loop(3, [0.8, 0.5, 0.25])
array([[0.8  , 0.1  , 0.1  ],
       [0.25 , 0.5  , 0.25 ],
       [0.375, 0.375, 0.25 ]])
```

librosa.sequence.transition_cycle

`librosa.sequence.transition_cycle`(*n_states*, *prob*)[source]

Construct a cyclic transition matrix over *n_states*.

The transition matrix will have the following properties:

- *transition[i, i] = p*
- *transition[i, i + 1] = (1 - p)*

This type of transition matrix is appropriate for state spaces with cyclical structure, such as metrical position within a bar. For example, a song in 4/4 time has state transitions of the form

1->{1, 2}, 2->{2, 3}, 3->{3, 4}, 4->{4, 1}.

Parameters: **n_states** : int > 1

The number of states

prob : float in [0, 1] or iterable, length=n_states

If a scalar, this is the probability of a self-transition.

If a vector of length *n_states*, *p[i]* is the probability of state *i*'s self-transition.

Returns: **transition** : np.ndarray [shape=(n_states, n_states)]

The transition matrix

Examples

```
>>> librosa.sequence.transition_cycle(4, 0.9)
array([[0.9, 0.1, 0. , 0. ],
       [0. , 0.9, 0.1, 0. ],
       [0. , 0. , 0.9, 0.1],
       [0.1, 0. , 0. , 0.9]])
```

librosa.sequence.transition_local

`librosa.sequence.transition_local`(*n_states*, *width*, *window='triangle'*, *wrap=False*) [source]

Construct a localized transition matrix.

The transition matrix will have the following properties:

- *transition[i, j] = 0* if *|i - j| > width*
- *transition[i, i]* is maximal
- *transition[i, i - width//2 : i + width//2]* has shape *window*

This type of transition matrix is appropriate for state spaces that discretely approximate continuous variables, such as in fundamental frequency estimation.

Parameters: **n_states** : int > 1

The number of states

width : int >= 1 or iterable

The maximum number of states to treat as "local". If iterable, it should have length equal to *n_states*, and specify the width independently for each state.

window : str, callable, or window specification

The window function to determine the shape of the "local" distribution.

Any window specification supported by *filters.get_window* will work

here.

Note

Certain windows (e.g., ‘hann’) are identically 0 at the boundaries, so and effectively have *width-2* non-zero values. You may have to expand *width* to get the desired behavior.

wrap : bool

If *True*, then state locality $|i - j|$ is computed modulo *n_states*. If *False* (default), then locality is absolute.

Returns: **transition** : np.ndarray [shape=(n_states, n_states)]

The transition matrix

See also

`filters.get_window`

Examples

Triangular distributions with and without wrapping

```
>>> librosa.sequence.transition_local(5, 3, window='triangle', wrap=False)
array([[0.667, 0.333, 0.   , 0.   , 0.   ],
       [0.25 , 0.5  , 0.25 , 0.   , 0.   ],
       [0.   , 0.25 , 0.5  , 0.25 , 0.   ],
       [0.   , 0.   , 0.25 , 0.5  , 0.25 ],
       [0.   , 0.   , 0.   , 0.333, 0.667]])

>>> librosa.sequence.transition_local(5, 3, window='triangle', wrap=True)
array([[0.5 , 0.25, 0.  , 0.  , 0.25],
       [0.25, 0.5 , 0.25, 0.  , 0.  ],
       [0.  , 0.25, 0.5 , 0.25, 0.  ],
       [0.  , 0.  , 0.25, 0.5 , 0.25],
       [0.25, 0.  , 0.  , 0.25, 0.5 ]])
```

Uniform local distributions with variable widths and no wrapping

```
>>> librosa.sequence.transition_local(5, [1, 2, 3, 3, 1], window='ones',
wrap=False)
array([[1.   , 0.   , 0.   , 0.   , 0.   ],
       [0.5  , 0.5  , 0.   , 0.   , 0.   ],
       [0.   , 0.333, 0.333, 0.333, 0.   ],
       [0.   , 0.   , 0.333, 0.333, 0.333],
       [0.   , 0.   , 0.   , 0.   , 1.   ]])
```

Utilities

Array operations

frame(x[, frame_length, hop_length, axis])	Slice a data array into (overlapping) frames.
pad_center(data, size[, axis])	Wrapper for np.pad to automatically center an array prior to padding.
fix_length(data, size[, axis])	Fix the length an array *data* to exactly *size*.
fix_frames(frames[, x_min, x_max, pad])	Fix a list of frames to lie within [x_min, x_max]
index_to_slice(idx[, idx_min, idx_max, …])	Generate a slice array from an index array.
softmask(X, X_ref[, power, split_zeros])	Robustly compute a softmask operation.
stack(arrays[, axis])	Stack one or more arrays along a target axis.
sync(data, idx[, aggregate, pad, axis])	Synchronous aggregation of a multi-dimensional array between boundaries
axis_sort(S[, axis, index, value])	Sort an array along its rows or columns.
normalize(S[, norm, axis, threshold, fill])	Normalize an array along a chosen axis.
shear(X[, factor, axis])	Shear a matrix by a given factor.
sparsify_rows(x[, quantile])	Return a row-sparse matrix approximating the input *x*.
buf_to_float(x[, n_bytes, dtype])	Convert an integer buffer to floating point values.
tiny(x)	Compute the tiny-value corresponding to an input's data type.

Matching

match_intervals(intervals_from, intervals_to)	Match one set of time intervals to another.
match_events(events_from, events_to[, left, …])	Match one set of events to another.

Miscellaneous

localmax(x[, axis])	Find local maxima in an array *x*.
peak_pick(x, pre_max, post_max, pre_avg, …)	Uses a flexible heuristic to pick peaks in a signal.
nnls(A, B, **kwargs)	Non-negative least squares.
cyclic_gradient(data[, edge_order, axis])	Estimate the gradient of a function over a uniformly sampled, periodic domain.

validation

valid_audio(y[, mono])	Validate whether a variable contains valid, mono audio data.
valid_int(x[, cast])	Ensure that an input value is integer-typed.
valid_intervals(intervals)	Ensure that an array is a valid representation of time intervals:

File operations

example_audio_file()	Get the path to an included audio example file.
find_files(directory[, ext, recurse, …])	Get a sorted list of (audio) files in a directory or directory sub-tree.

Deprecated

roll_sparse(x, shift[, axis])	Sparse matrix roll

librosa.util.frame

librosa.util.frame(*x*, *frame_length=2048*, *hop_length=512*, *axis=-1*)[source]

Slice a data array into (overlapping) frames.

This implementation uses low-level stride manipulation to avoid making a copy of the data. The resulting frame representation is a new view of the input data.

For example, a one-dimensional input *x = [0, 1, 2, 3, 4, 5, 6]* can be framed with frame length 3 and hop length 2 in two ways. The first (*axis=-1*), results in the array *x_frames*:

[[0, 2, 4],
[1, 3, 5], [2, 4, 6]]

where each column *x_frames[:, i]* contains a contiguous slice of the input *x[i * hop_length : i * hop_length + frame_length]*.

The second way (*axis=0*) results in the array *x_frames*:

[[0, 1, 2],
[2, 3, 4], [4, 5, 6]]

where each row *x_frames[i]* contains a contiguous slice of the input.

This generalizes to higher dimensional inputs, as shown in the examples below. In general, the framing operation increments by 1 the number of dimensions, adding a new “frame axis” either to the end of the array (*axis=-1*) or the beginning of the array (*axis=0*).

Parameters: **x** : np.ndarray

Time series to frame. Must be contiguous in memory, see the “Raises” section below for more information.

frame_length : int > 0 [scalar]

Length of the frame

hop_length : int > 0 [scalar]

Number of steps to advance between frames

axis : 0 or -1

The axis along which to frame.

If *axis=-1* (the default), then *x* is framed along its last dimension. *x* must be “F-contiguous” in this case.

If *axis=0*, then *x* is framed along its first dimension. *x* must be “C-contiguous” in this case.

Returns: **x_frames** : np.ndarray [shape=(…, frame_length, N_FRAMES) or (N_FRAMES, frame_length, …)]

A framed view of *x*, for example with *axis=-1* (framing on the last dimension): *x_frames[…, j] == x[…, j * hop_length : j * hop_length + frame_length]*

If *axis=0* (framing on the first dimension), then: *x_frames[j] = x[j * hop_length : j * hop_length + frame_length]*

Raises: ParameterError

If *x* is not contiguous in memory or not an *np.ndarray*.

If *x.shape[axis] < frame_length*, there is not enough data to fill one frame.

If *hop_length < 1*, frames cannot advance.

If *axis* is not 0 or -1. Framing is only supported along the first or last axis.

If *axis=-1* (the default), then *x* must be “F-contiguous”. If *axis=0*, then *x* must be “C-contiguous”.

If the contiguity of *x* is incompatible with the framing axis.

See also

`np.asfortranarray`
: Convert data to F-contiguous representation

`np.ascontiguousarray`
: Convert data to C-contiguous representation

`np.ndarray.flags`
: information about the memory layout of a numpy *ndarray*.

Examples

Extract 2048-sample frames from monophonic *y* with a hop of 64 samples per frame

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> frames = librosa.util.frame(y, frame_length=2048, hop_length=64)
>>> frames
array([[ 0.000e+00,  0.000e+00, ..., -2.448e-06, -6.789e-07],
       [ 0.000e+00,  0.000e+00, ..., -1.399e-05,  1.004e-06],
       ...,
       [-7.352e-04,  5.162e-03, ...,  0.000e+00,  0.000e+00],
       [ 2.168e-03,  4.870e-03, ...,  0.000e+00,  0.000e+00]],
      dtype=float32)
>>> y.shape
(1355168,)
>>> frames.shape
(2048, 21143)
```

Or frame along the first axis instead of the last:

```
>>> frames = librosa.util.frame(y, frame_length=2048, hop_length=64,
axis=0)
>>> frames.shape
(21143, 2048)
```

Frame a stereo signal:

```
>>> y, sr = librosa.load(librosa.util.example_audio_file(), mono=False)
>>> y.shape
(2, 1355168)
>>> frames = librosa.util.frame(y, frame_length=2048, hop_length=64)
(2, 2048, 21143)
```

Carve an STFT into fixed-length patches of 32 frames with 50% overlap

```
>>> y, sr = librosa.load(librosa.util.example_audio_file())
>>> S = np.abs(librosa.stft(y))
>>> S.shape
(1025, 2647)
>>> S_patch = librosa.util.frame(S, frame_length=32, hop_length=16)
>>> S_patch.shape
(1025, 32, 82)
>>> # The first patch contains the first 32 frames of S
>>> np.allclose(S_patch[:, :, 0], S[:, :32])
True
>>> # The second patch contains frames 16 to 16+32=48, and so on
>>> np.allclose(S_patch[:, :, 1], S[:, 16:48])
True
```

librosa.util.pad_center

`librosa.util.pad_center`(*data*, *size*, *axis=-1*, ***kwargs*)[source]

Wrapper for np.pad to automatically center an array prior to padding. This is analogous to *str.center()*

Parameters: **data** : np.ndarray

Vector to be padded and centered

size : int >= len(data) [scalar]

Length to pad *data*

axis : int

Axis along which to pad and center the data

kwargs : additional keyword arguments

arguments passed to *np.pad()*

Returns: **data_padded** : np.ndarray

data centered and padded to length *size* along the specified axis

Raises: ParameterError

If *size* < *data.shape[axis]*

See also

`numpy.pad`

Examples

```
>>> # Generate a vector
>>> data = np.ones(5)
>>> librosa.util.pad_center(data, 10, mode='constant')
array([ 0.,  0.,  1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.])

>>> # Pad a matrix along its first dimension
>>> data = np.ones((3, 5))
>>> librosa.util.pad_center(data, 7, axis=0)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> # Or its second dimension
```

```
>>> librosa.util.pad_center(data, 7, axis=1)
array([[ 0.,  1.,  1.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  1.,  1.,  0.]])
```

librosa.util.fix_length

`librosa.util.fix_length`(*data*, *size*, *axis=-1*, ***kwargs*)[source]

Fix the length an array *data* to exactly *size*.

If *data.shape[axis]* < *n*, pad according to the provided kwargs. By default, *data* is padded with trailing zeros.

Parameters:

data : np.ndarray

array to be length-adjusted

size : int >= 0 [scalar]

desired length of the array

axis : int, <= data.ndim

axis along which to fix length

kwargs : additional keyword arguments

Parameters to *np.pad()*

Returns:

data_fixed : np.ndarray [shape=data.shape]

data either trimmed or padded to length *size* along the specified axis.

See also

`numpy.pad`

Examples

```
>>> y = np.arange(7)
>>> # Default: pad with zeros
>>> librosa.util.fix_length(y, 10)
array([0, 1, 2, 3, 4, 5, 6, 0, 0, 0])
>>> # Trim to a desired length
>>> librosa.util.fix_length(y, 5)
array([0, 1, 2, 3, 4])
>>> # Use edge-padding instead of zeros
>>> librosa.util.fix_length(y, 10, mode='edge')
array([0, 1, 2, 3, 4, 5, 6, 6, 6, 6])
```

librosa.util.fix_frames

`librosa.util.fix_frames`(*frames*, *x_min=0*, *x_max=None*, *pad=True*)[source]

Fix a list of frames to lie within [x_min, x_max]

Parameters: **frames** : np.ndarray [shape=(n_frames,)]

List of non-negative frame indices

x_min : int >= 0 or None

Minimum allowed frame index

x_max : int >= 0 or None

Maximum allowed frame index

pad : boolean

If *True*, then *frames* is expanded to span the full range *[x_min, x_max]*

Returns: **fixed_frames** : np.ndarray [shape=(n_fixed_frames,), dtype=int]

Fixed frame indices, flattened and sorted

Raises: ParameterError

If *frames* contains negative values

Examples

```
>>> # Generate a list of frame indices
>>> frames = np.arange(0, 1000.0, 50)
>>> frames
array([   0.,   50.,  100.,  150.,  200.,  250.,  300.,  350.,
        400.,  450.,  500.,  550.,  600.,  650.,  700.,  750.,
        800.,  850.,  900.,  950.])
>>> # Clip to span at most 250
>>> librosa.util.fix_frames(frames, x_max=250)
array([  0,  50, 100, 150, 200, 250])
>>> # Or pad to span up to 2500
>>> librosa.util.fix_frames(frames, x_max=2500)
array([   0,   50,  100,  150,  200,  250,  300,  350,  400,
        450,  500,  550,  600,  650,  700,  750,  800,  850,
        900,  950, 2500])
>>> librosa.util.fix_frames(frames, x_max=2500, pad=False)
array([  0,  50, 100, 150, 200, 250, 300, 350, 400, 450, 500,
       550, 600, 650, 700, 750, 800, 850, 900, 950])

>>> # Or starting away from zero
>>> frames = np.arange(200, 500, 33)
>>> frames
```

```
array([200, 233, 266, 299, 332, 365, 398, 431, 464, 497])
>>> librosa.util.fix_frames(frames)
array([  0, 200, 233, 266, 299, 332, 365, 398, 431, 464, 497])
>>> librosa.util.fix_frames(frames, x_max=500)
array([  0, 200, 233, 266, 299, 332, 365, 398, 431, 464, 497,
       500])
```

librosa.util.index_to_slice

`librosa.util.index_to_slice`(*idx*, *idx_min=None*, *idx_max=None*, *step=None*, *pad=True*)[source]

Generate a slice array from an index array.

Parameters: **idx** : list-like

Array of index boundaries

idx_min : None or int
idx_max : None or int

Minimum and maximum allowed indices

step : None or int

Step size for each slice. If *None*, then the default step of 1 is used.

pad : boolean

If *True*, pad *idx* to span the range *idx_min:idx_max*.

Returns: **slices** : list of slice

`slices[i] = slice(idx[i], idx[i+1], step)` Additional slice objects may be added at the beginning or end, depending on whether `pad==True` and the supplied values for *idx_min* and *idx_max*.

See also

`fix_frames`

Examples

```
>>> # Generate slices from spaced indices
>>> librosa.util.index_to_slice(np.arange(20, 100, 15))
[slice(20, 35, None), slice(35, 50, None), slice(50, 65, None), slice(65,
80, None),
 slice(80, 95, None)]
>>> # Pad to span the range (0, 100)
>>> librosa.util.index_to_slice(np.arange(20, 100, 15),
...                             idx_min=0, idx_max=100)
[slice(0, 20, None), slice(20, 35, None), slice(35, 50, None), slice(50,
65, None),
 slice(65, 80, None), slice(80, 95, None), slice(95, 100, None)]
```

```
>>> # Use a step of 5 for each slice
>>> librosa.util.index_to_slice(np.arange(20, 100, 15),
...                             idx_min=0, idx_max=100, step=5)
[slice(0, 20, 5), slice(20, 35, 5), slice(35, 50, 5), slice(50, 65, 5),
slice(65, 80, 5),
 slice(80, 95, 5), slice(95, 100, 5)]
```

librosa.util.softmask

`librosa.util.softmask`(*X*, *X_ref*, *power=1*, *split_zeros=False*)[source]

Robustly compute a softmask operation.

*M = X**power / (X**power + X_ref**power)*

Parameters: **X** : np.ndarray

The (non-negative) input array corresponding to the positive mask elements

X_ref : np.ndarray

The (non-negative) array of reference or background elements. Must have the same shape as *X*.

power : number > 0 or np.inf

If finite, returns the soft mask computed in a numerically stable way

If infinite, returns a hard (binary) mask equivalent to *X* > *X_ref*. Note: for hard masks, ties are always broken in favor of *X_ref* (*mask=0*).

split_zeros : bool

If *True*, entries where *X* and X`_ref` are both small (close to 0) will receive mask values of 0.5.

Otherwise, the mask is set to 0 for these entries.

Returns: **mask** : np.ndarray, shape=`X.shape`

The output mask array

Raises: ParameterError

If *X* and *X_ref* have different shapes.

If *X* or *X_ref* are negative anywhere

If *power* <= *0*

Examples

```
>>> X = 2 * np.ones((3, 3))
>>> X_ref = np.vander(np.arange(3.0))
>>> X
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
>>> X_ref
array([[ 0.,  0.,  1.],
       [ 1.,  1.,  1.],
       [ 4.,  2.,  1.]])
>>> librosa.util.softmask(X, X_ref, power=1)
array([[ 1.   ,  1.   ,  0.667],
       [ 0.667,  0.667,  0.667],
       [ 0.333,  0.5  ,  0.667]])
>>> librosa.util.softmask(X_ref, X, power=1)
array([[ 0.   ,  0.   ,  0.333],
       [ 0.333,  0.333,  0.333],
       [ 0.667,  0.5  ,  0.333]])
>>> librosa.util.softmask(X, X_ref, power=2)
array([[ 1. ,  1. ,  0.8],
       [ 0.8,  0.8,  0.8],
       [ 0.2,  0.5,  0.8]])
>>> librosa.util.softmask(X, X_ref, power=4)
array([[ 1.   ,  1.   ,  0.941],
       [ 0.941,  0.941,  0.941],
       [ 0.059,  0.5  ,  0.941]])
>>> librosa.util.softmask(X, X_ref, power=100)
array([[  1.000e+00,   1.000e+00,   1.000e+00],
       [  1.000e+00,   1.000e+00,   1.000e+00],
       [  7.889e-31,   5.000e-01,   1.000e+00]])
>>> librosa.util.softmask(X, X_ref, power=np.inf)
array([[ True,  True,  True],
       [ True,  True,  True],
       [False, False,  True]], dtype=bool)
```

librosa.util.stack

`librosa.util.stack`(*arrays*, *axis=0*)[source]

Stack one or more arrays along a target axis.

This function is similar to *np.stack*, except that memory contiguity is retained when stacking along the first dimension.

This is useful when combining multiple monophonic audio signals into a multi-channel signal, or when stacking multiple feature representations to form a multi-dimensional array.

Parameters: arrays : list

one or more *np.ndarray*

axis : integer

The target axis along which to stack. *axis=0* creates a new first axis, and

axis=-1 creates a new last axis.

Returns: **arr_stack** : np.ndarray [shape=(len(arrays), array_shape) or shape=(array_shape, len(arrays))]

The input arrays, stacked along the target dimension.

If *axis=0*, then *arr_stack* will be F-contiguous. Otherwise, *arr_stack* will be C-contiguous by default, as computed by *np.stack*.

Raises: ParameterError

- If *arrays* do not all have the same shape
- If no *arrays* are given

See also

`np.stack`
`np.ndarray.flags`
`frame`

Examples

Combine two buffers into a contiguous arrays

```
>>> y_left = np.ones(5)
>>> y_right = -np.ones(5)
>>> y_stereo = librosa.util.stack([y_left, y_right], axis=0)
>>> y_stereo
array([[ 1.,  1.,  1.,  1.,  1.],
       [-1., -1., -1., -1., -1.]])
>>> y_stereo.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

Or along the trailing axis

```
>>> y_stereo = librosa.util.stack([y_left, y_right], axis=-1)
>>> y_stereo
array([[ 1., -1.],
       [ 1., -1.],
       [ 1., -1.],
       [ 1., -1.],
       [ 1., -1.]])
>>> y_stereo.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```