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

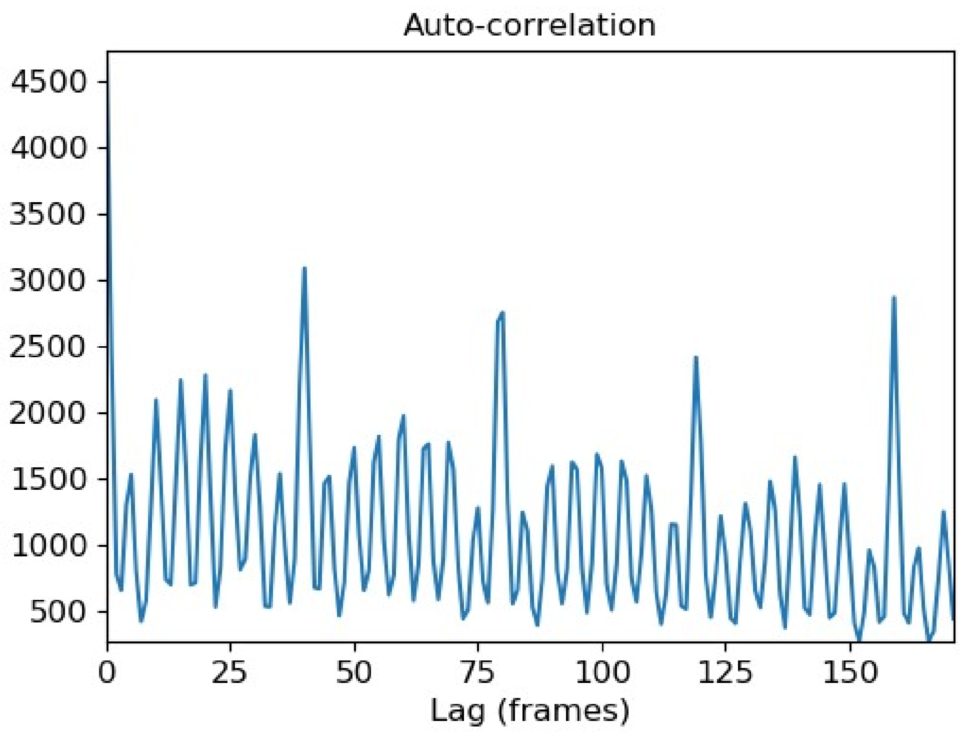
Compute full autocorrelation of y

```
>>> 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

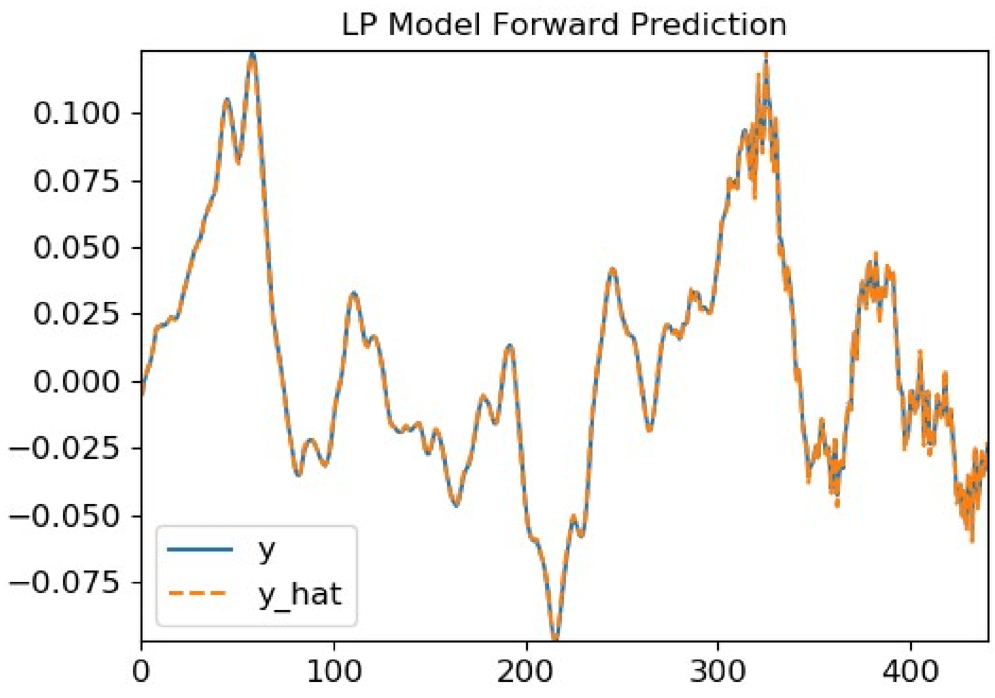
Compute LP coefficients of y at order 16 on entire series

```
>>> 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)
```

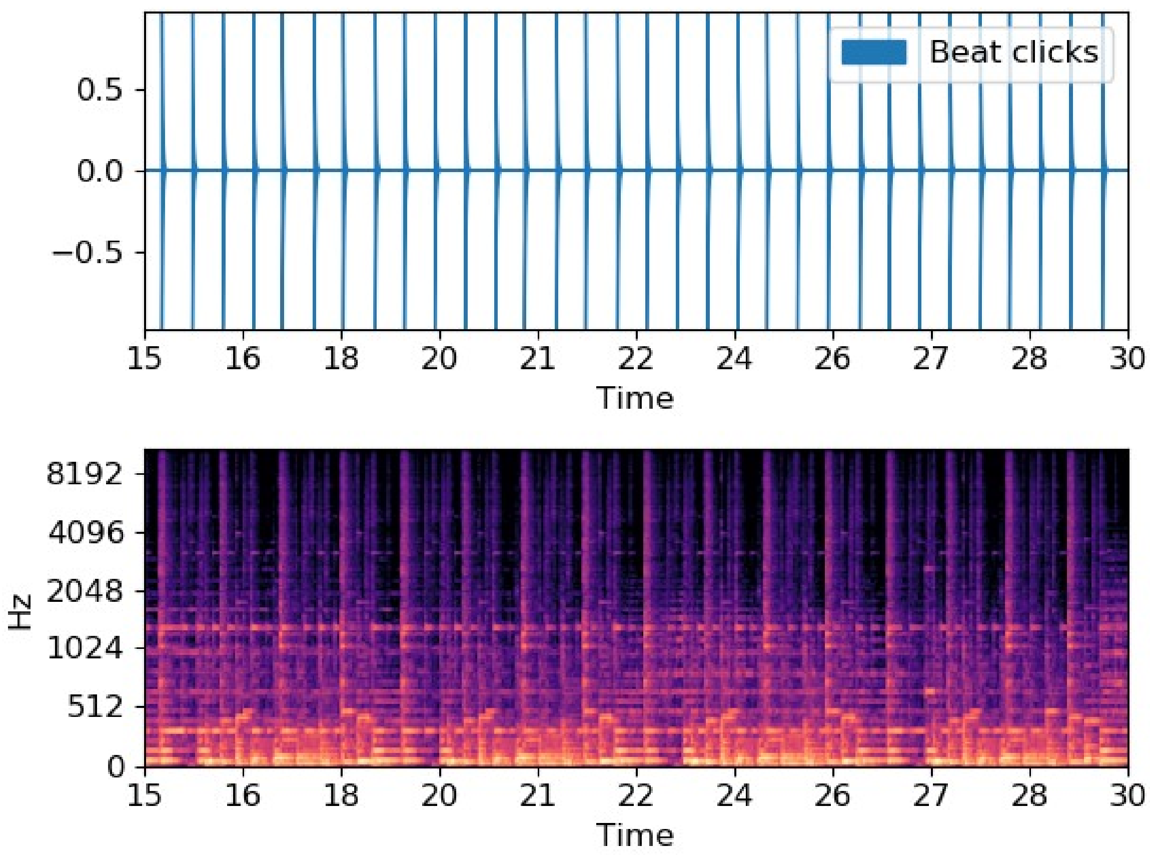
```
>>> # Or generate a signal of the same length as y
>>> y_beats = librosa.clicks(frames=beats, sr=sr, length=len(y))
```

```
>>> # 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.

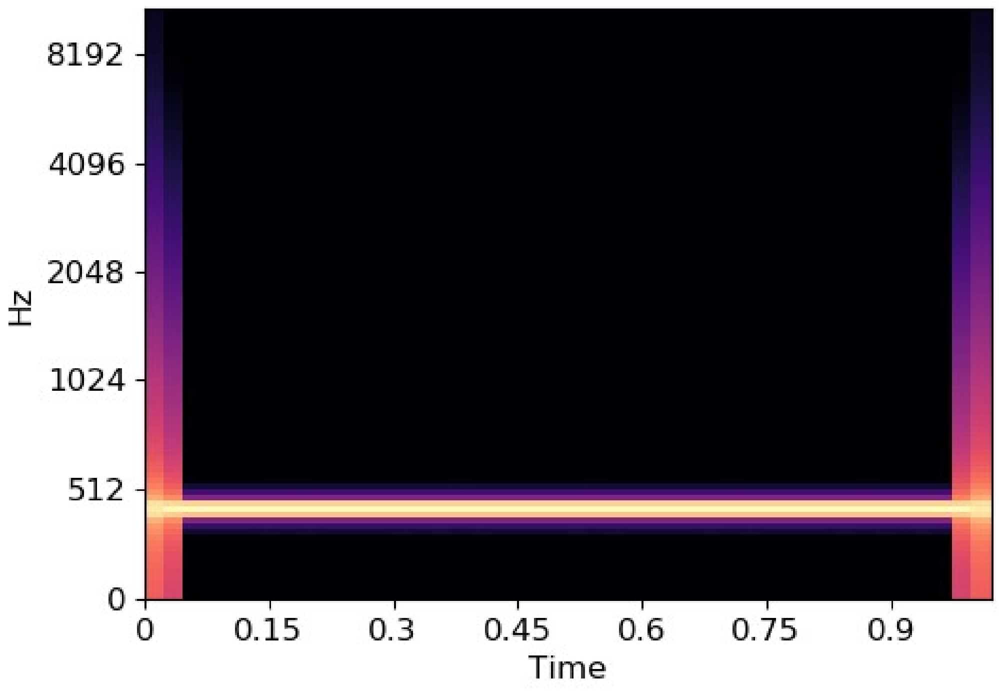
Examples

```
>>> # 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

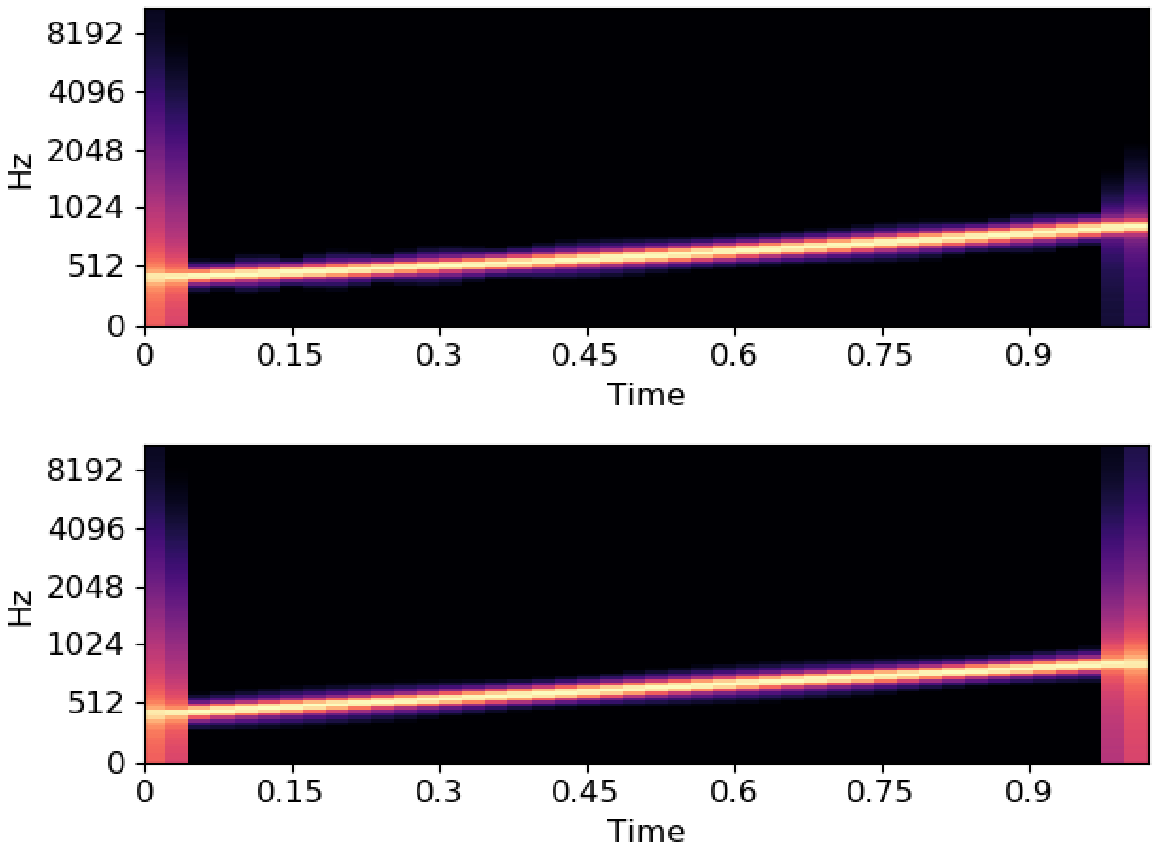
scipy.signal.chirp

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))
```

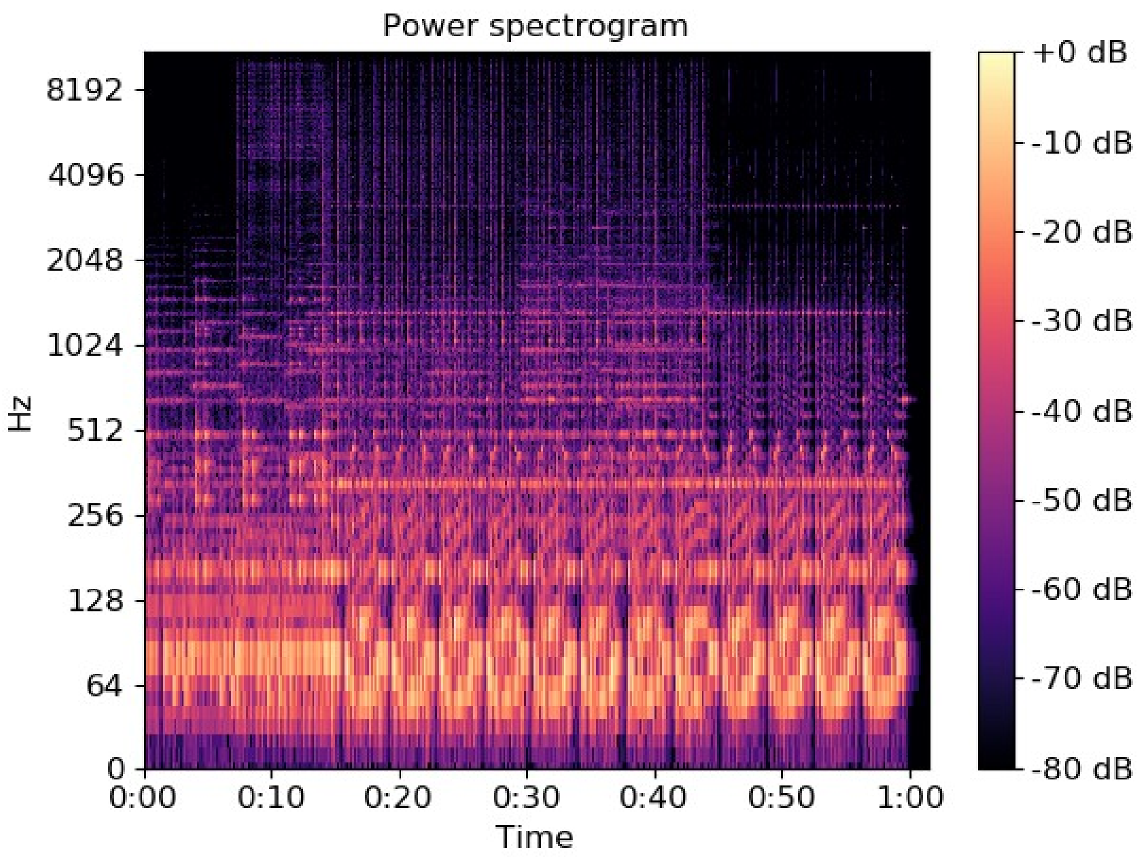
Use a shorter hop length

```
>>> 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

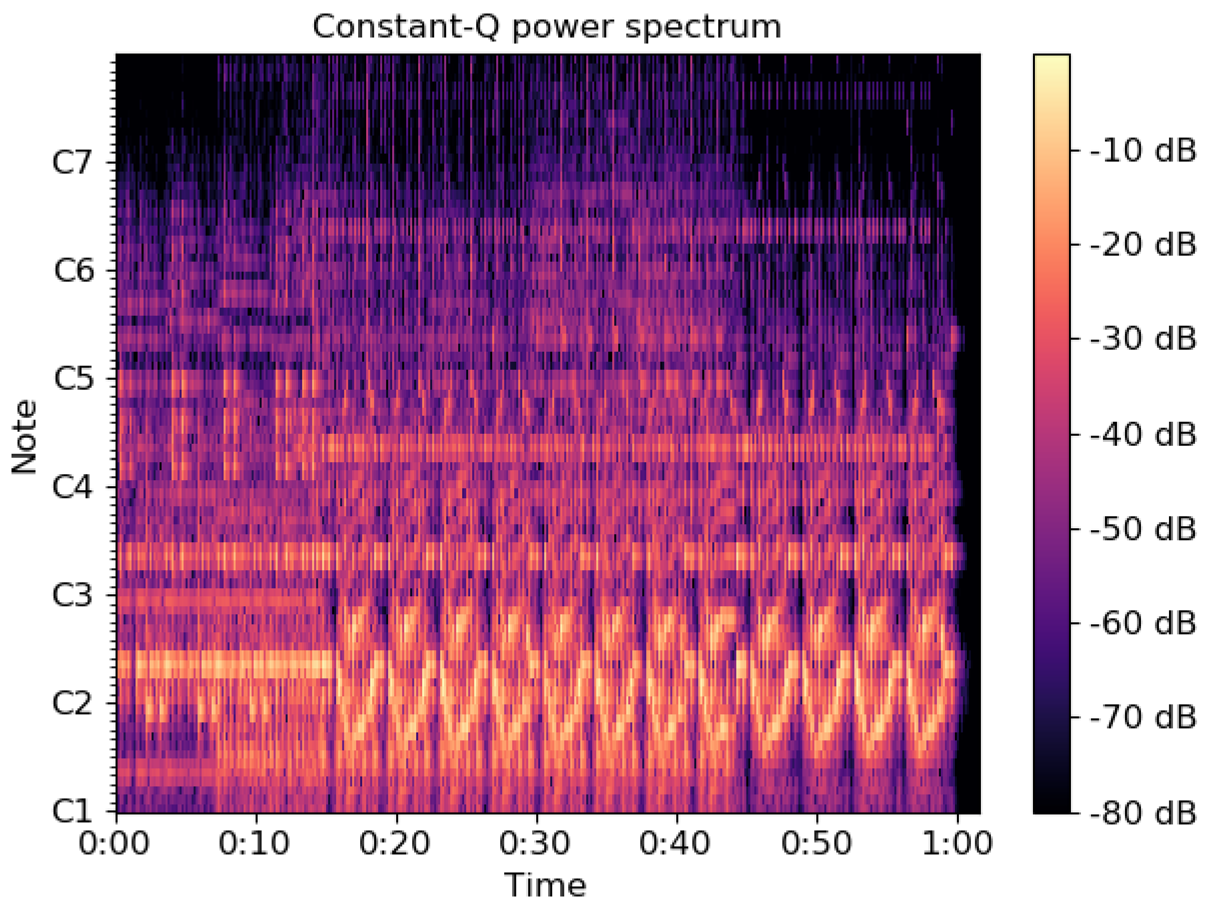
This function caches at level 20.

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

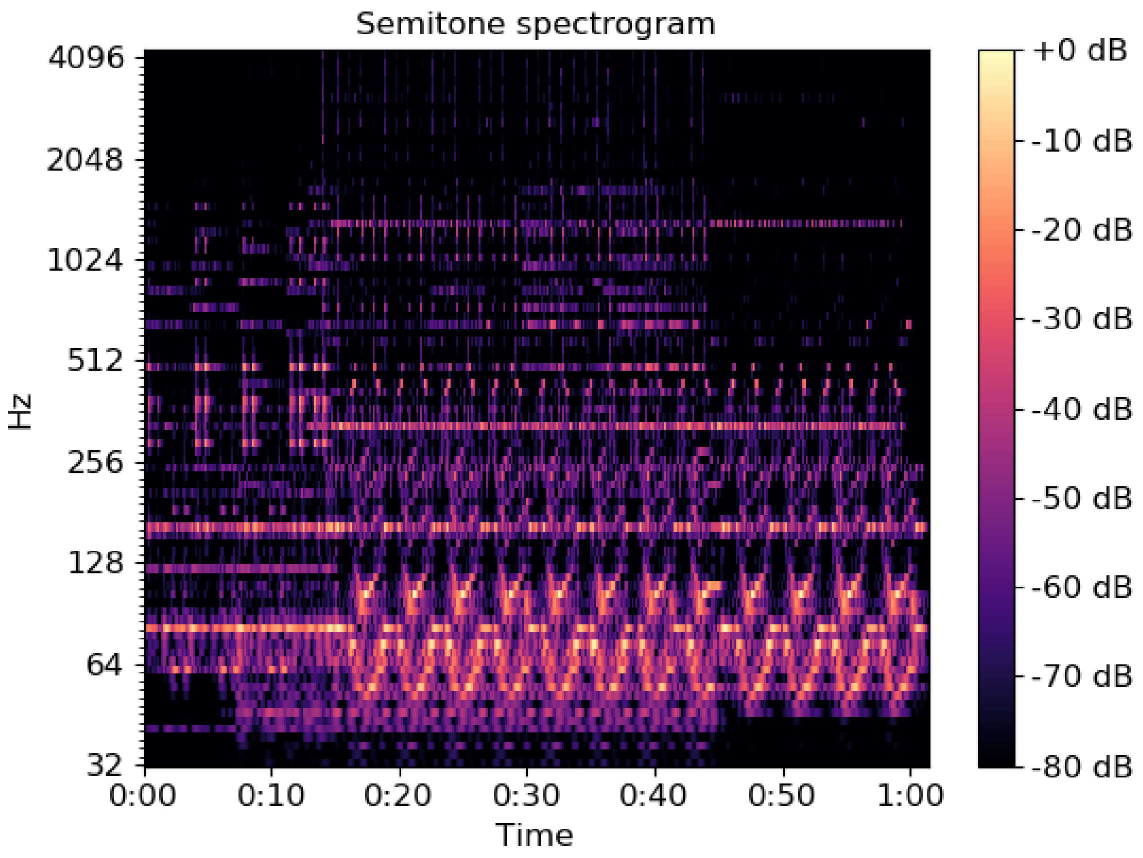
If *flayout* is not None, *ba*, or *sos*.

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

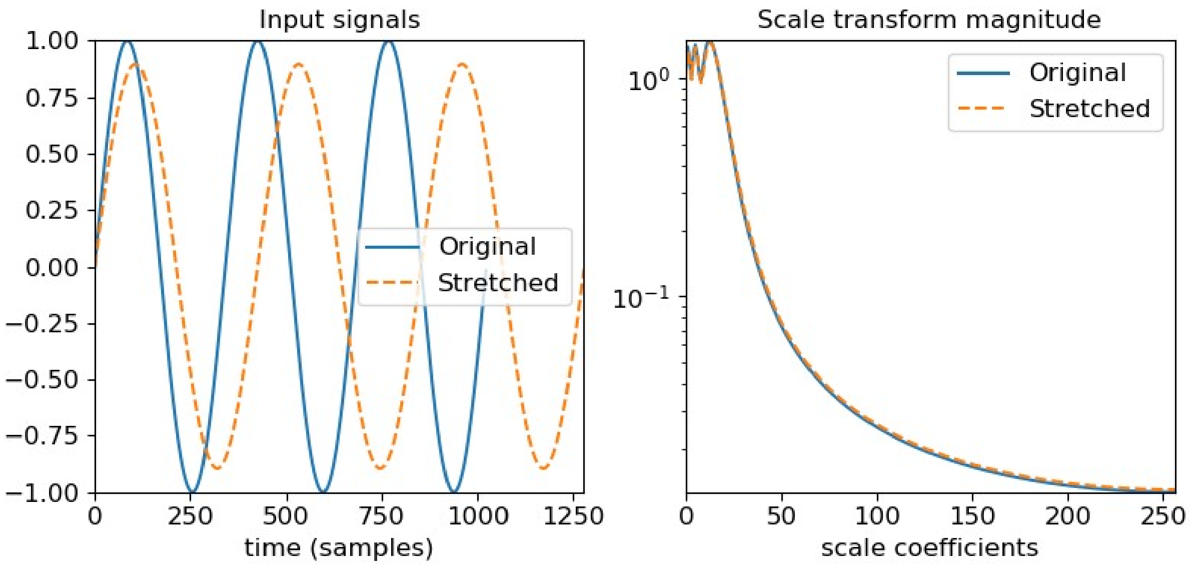
if *n_fmt < 2* or *t_min <= 0* or if *y* is not finite or if *y.shape[axis] < 3*.

Notes

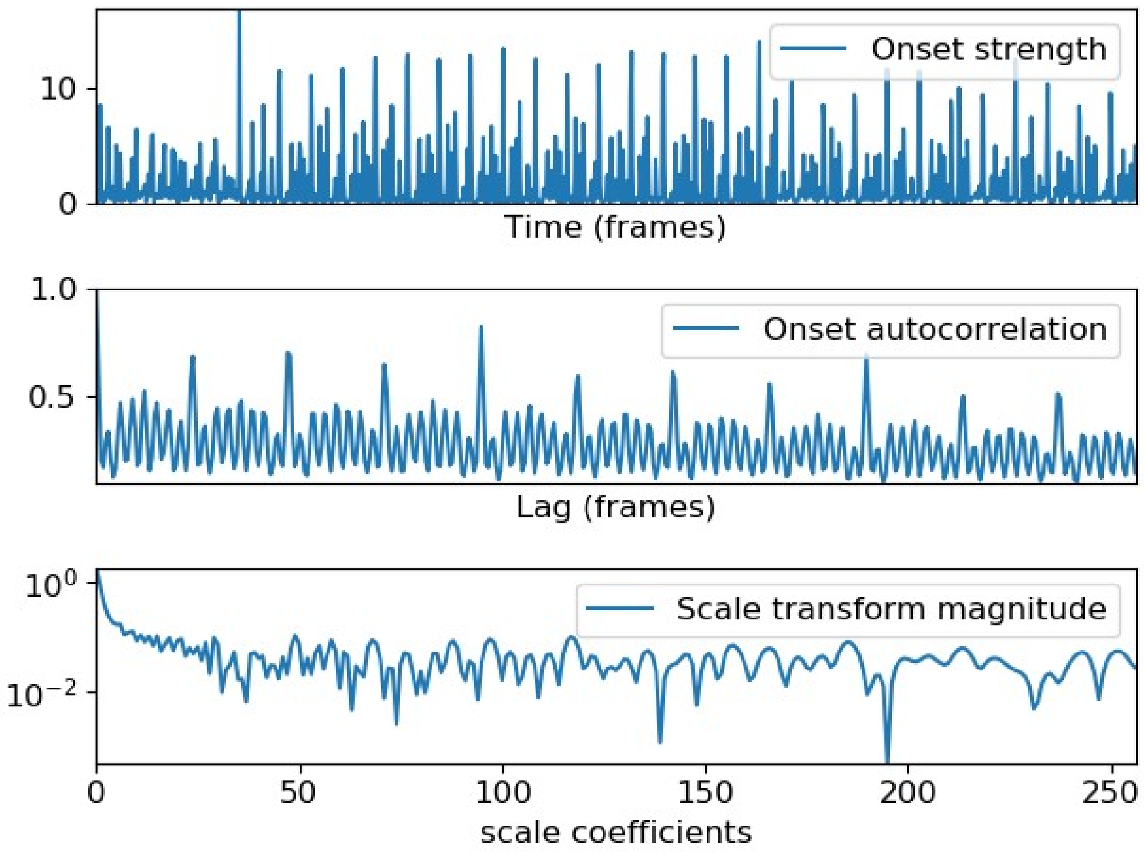
This function caches at level 30.

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`

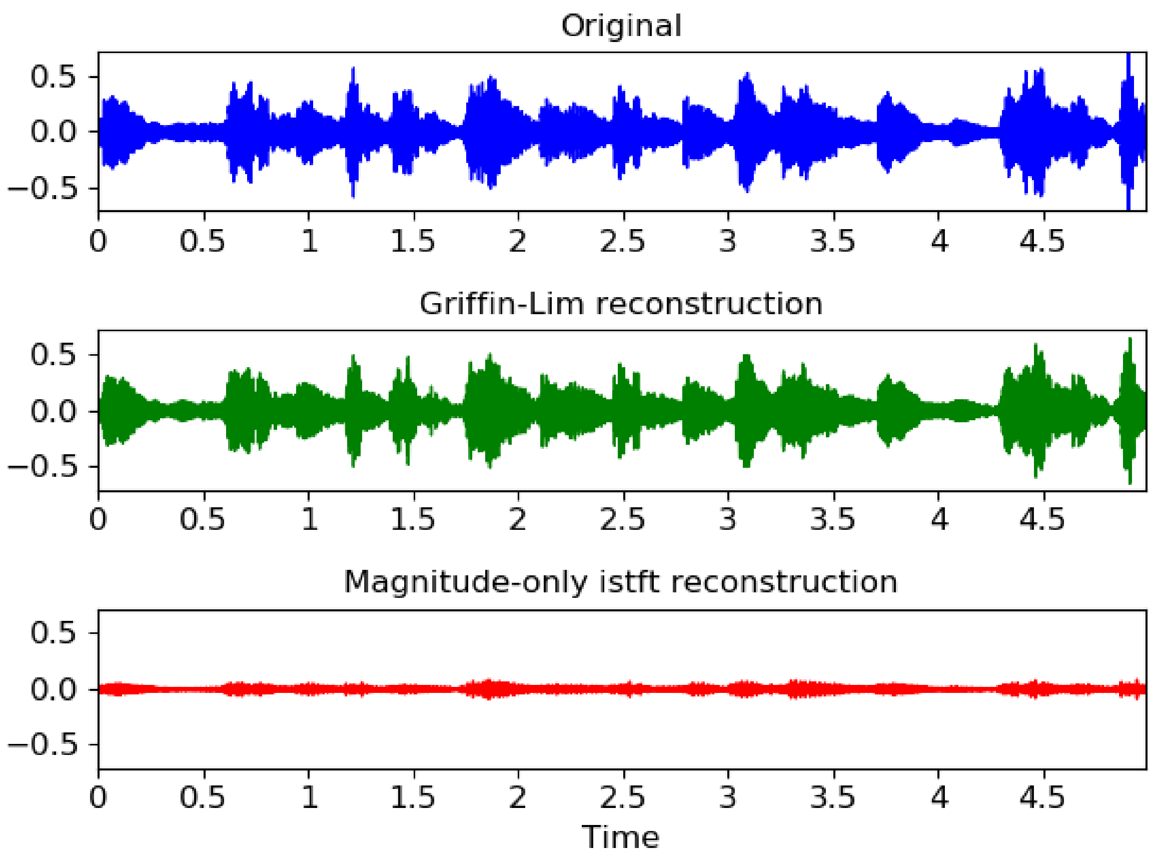
Examples

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

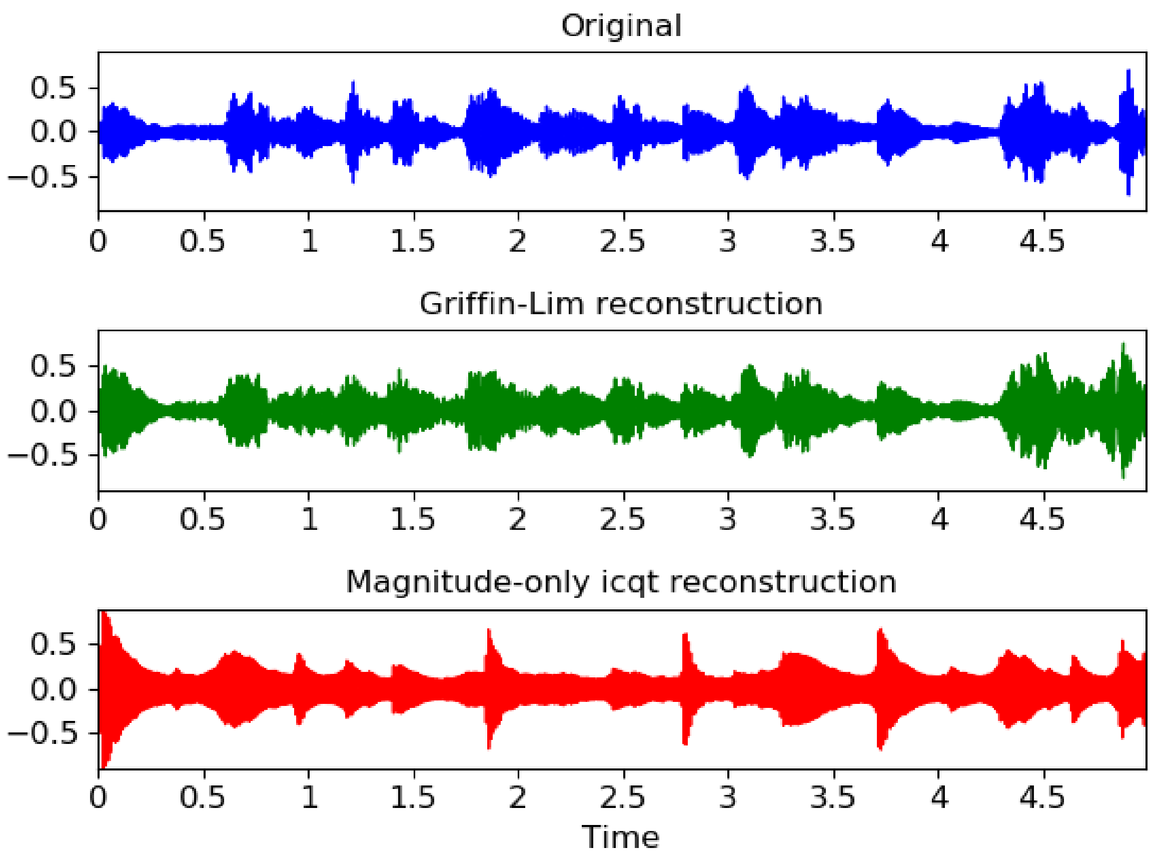
A basis CQT inverse example

```
>>> 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

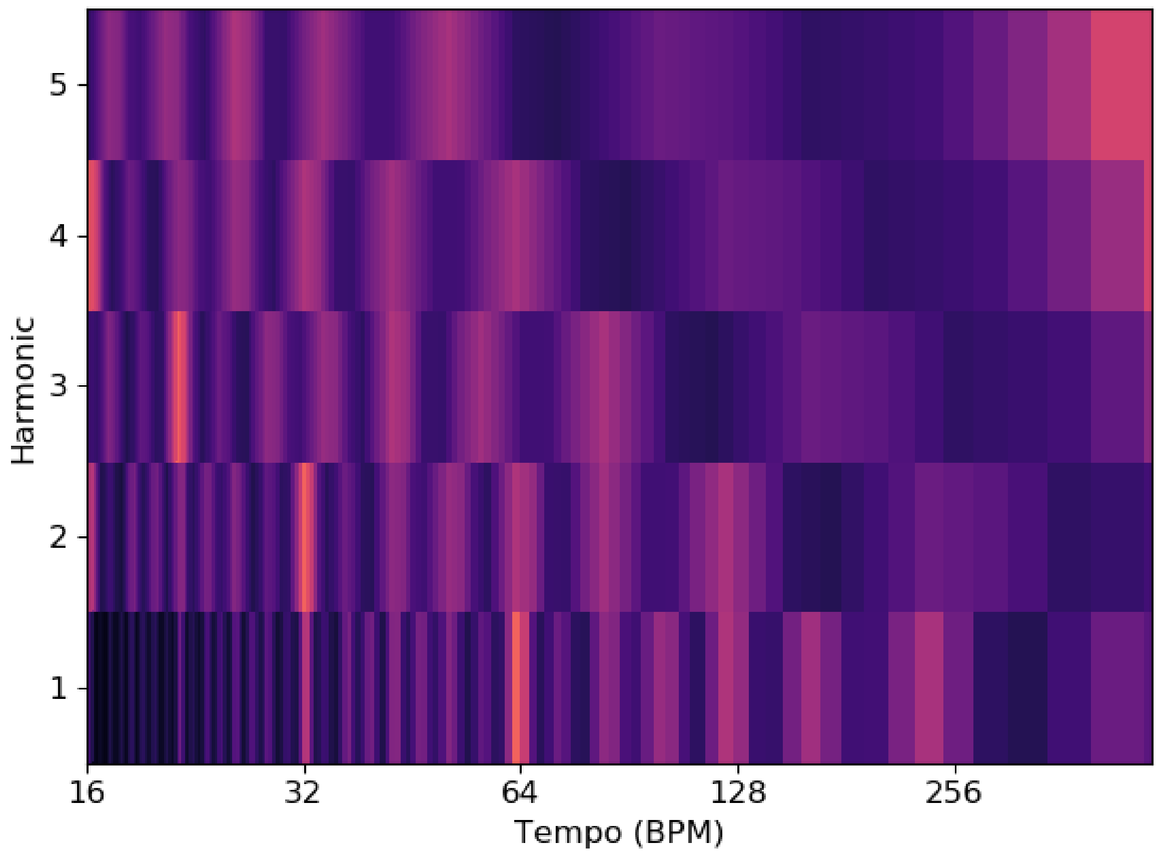
`scipy.interpolate.interp1d`

Examples

Estimate the harmonics of a time-averaged tempogram

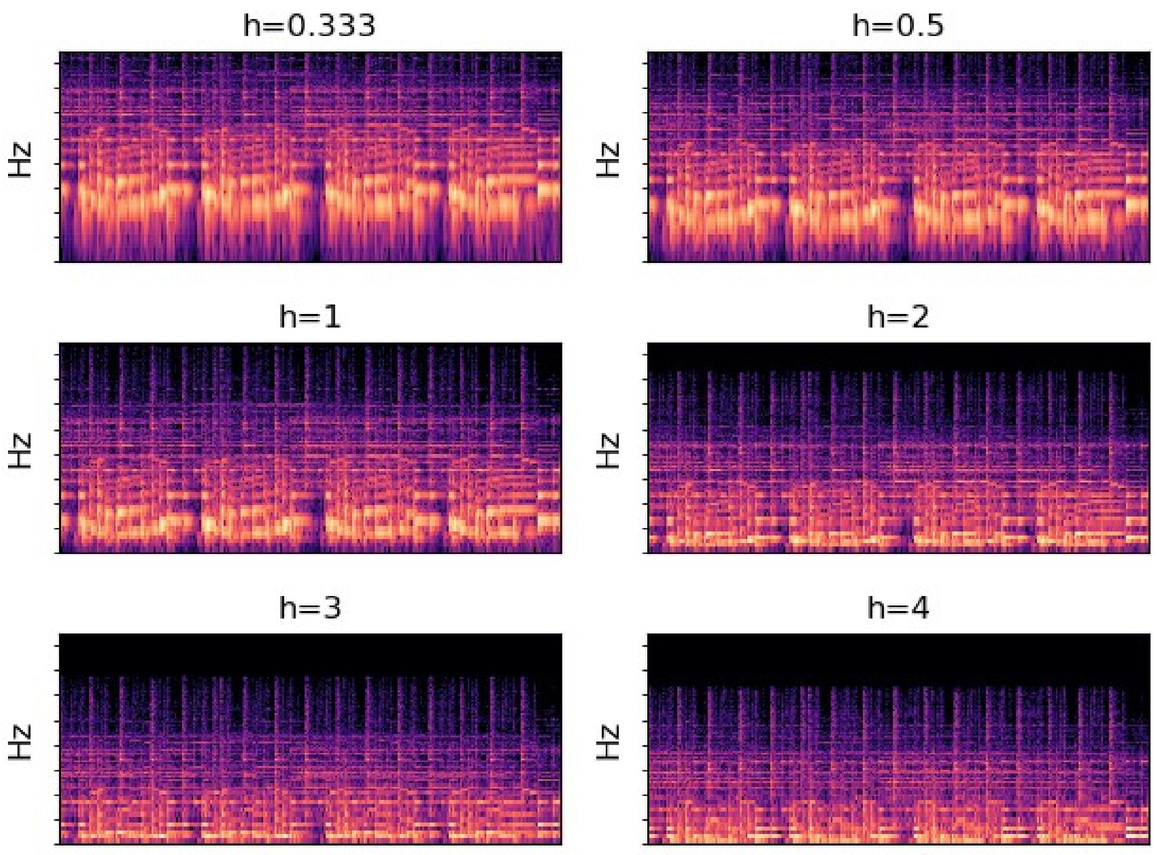
```
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      duration=15, offset=30)
>>> # Compute the time-varying tempogram and average over time
>>> tempi = np.mean(librosa.feature.tempogram(y=y, sr=sr), axis=1)
>>> # We'll measure the first five harmonics
>>> h_range = [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])

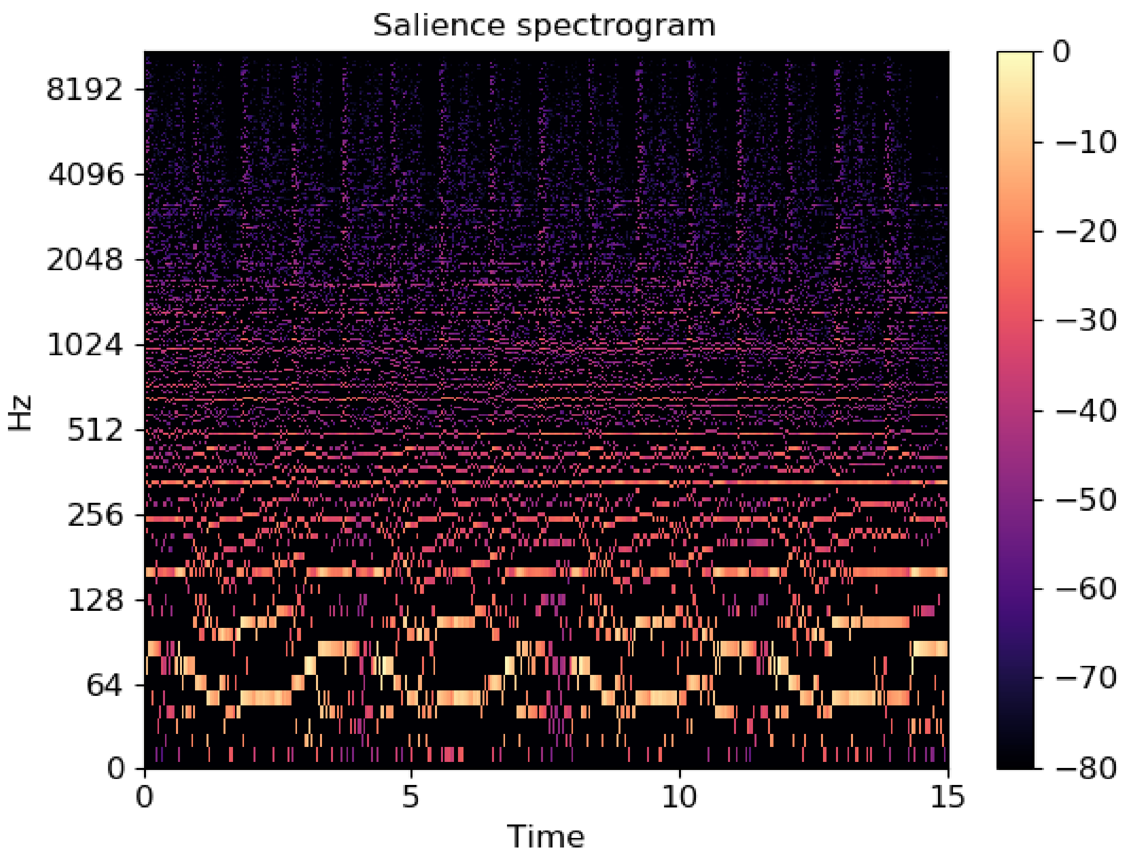
S_sal will have the same shape as *S*, and measure the overal harmonic energy at each frequency.

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

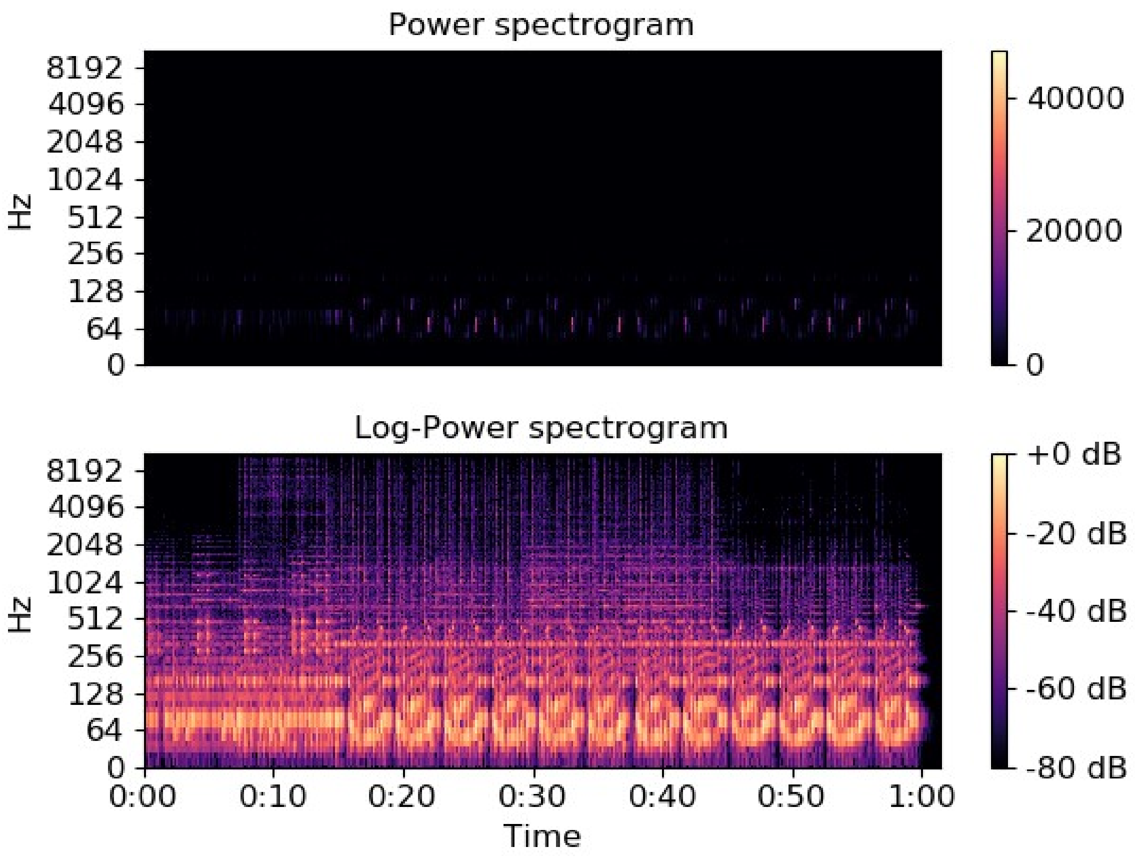
```
>>> librosa.power_to_db(S**2, ref=np.max)
array([[-80.   , -74.027, ..., -80.   , -80.   ],
       [-80.   , -72.431, ..., -80.   , -80.   ],
       ...,
       [-80.   , -80.   , ..., -80.   , -80.   ],
       [-80.   , -80.   , ..., -80.   , -80.   ]], dtype=float32)
```

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`

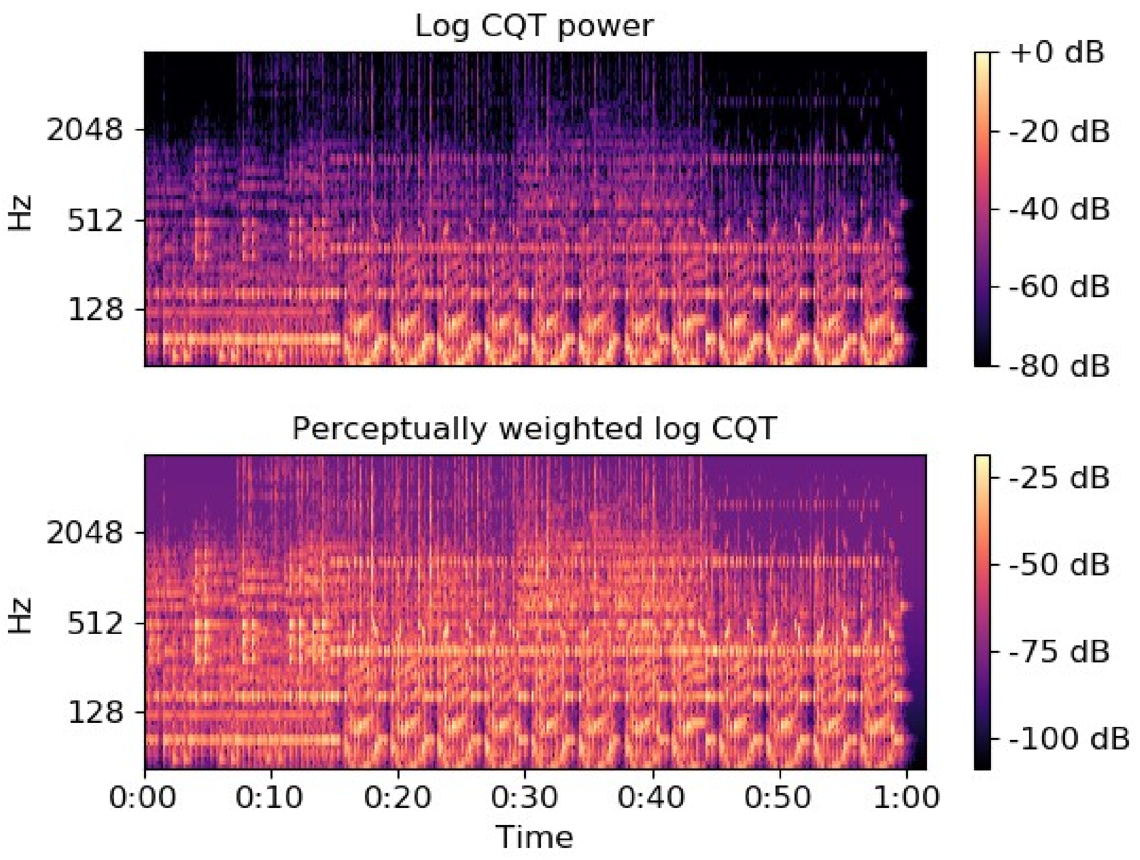
Notes

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]*

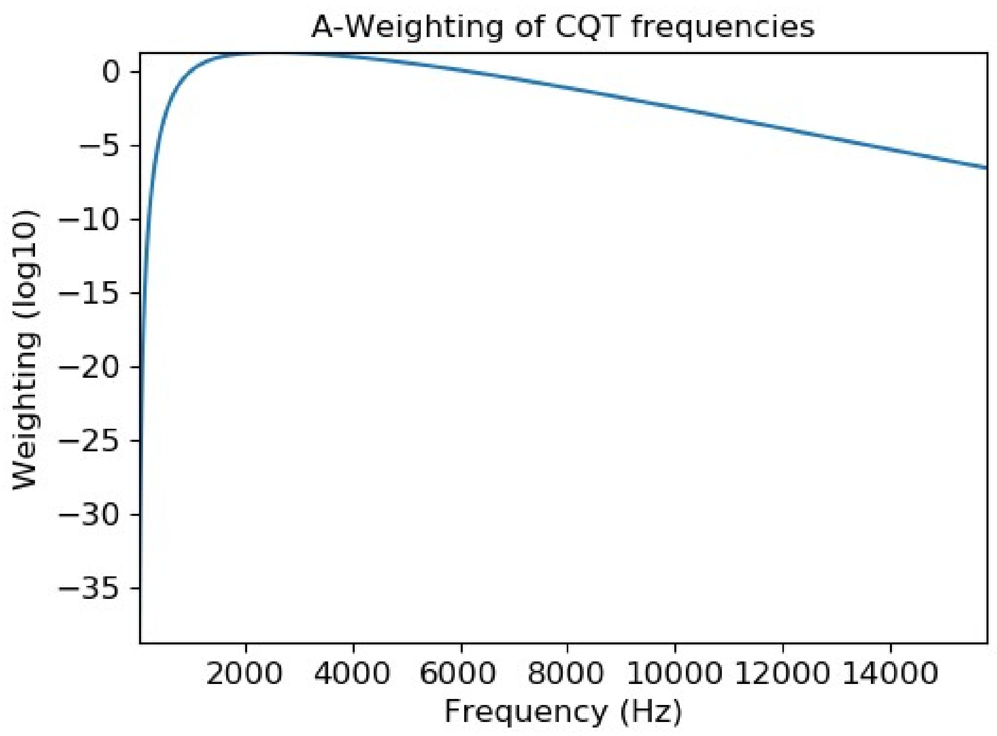
See also

`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*.

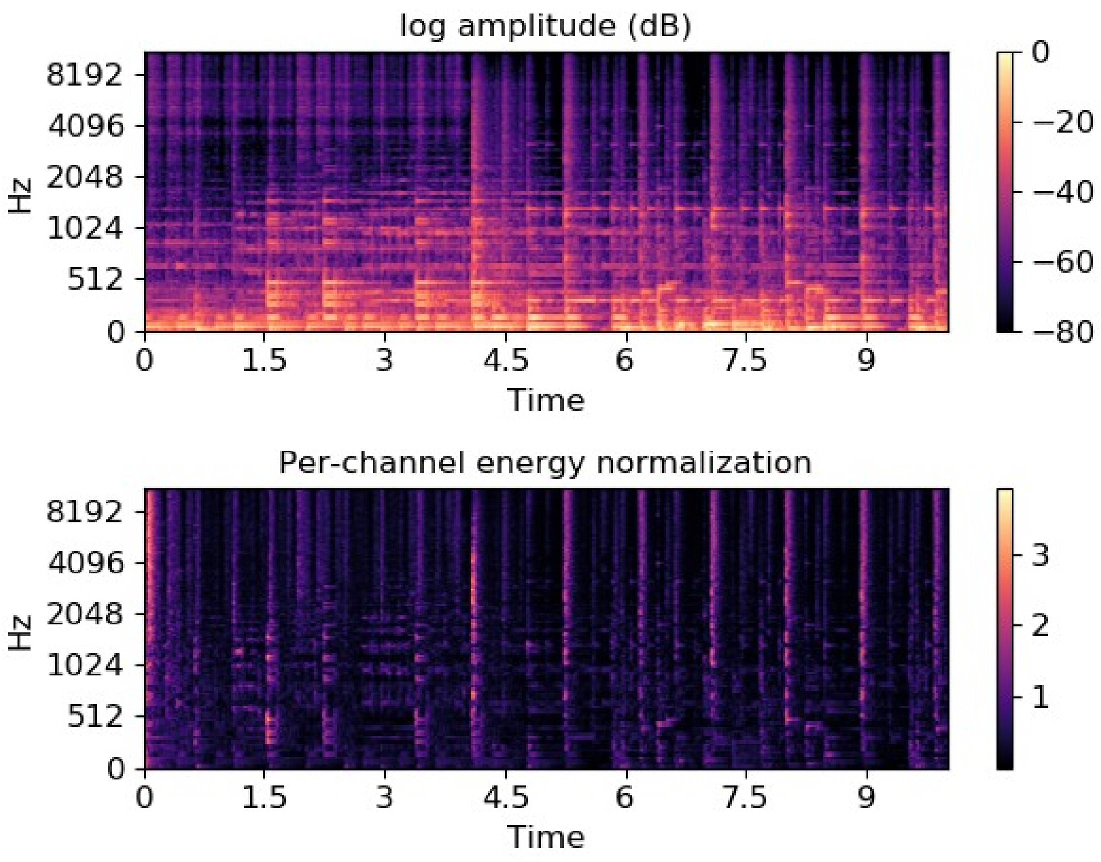
See also

`amplitude_to_db`
`librosa.onset.onset_strength`

Examples

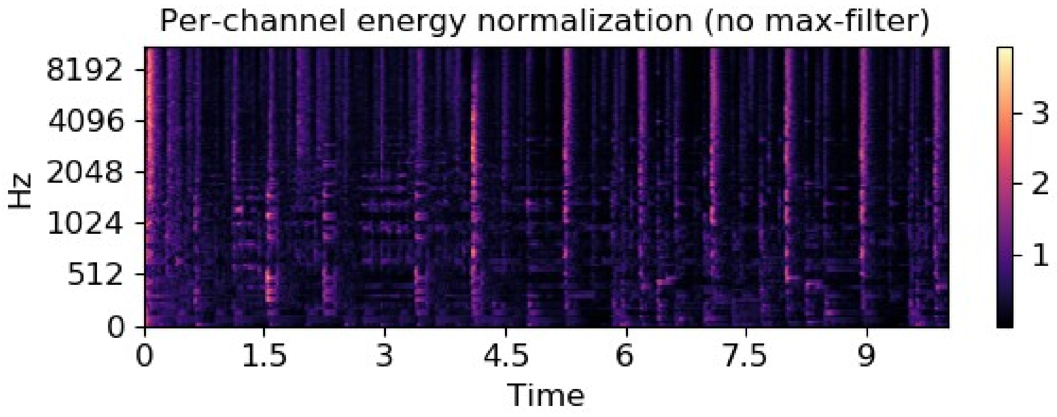
Compare PCEN to log amplitude (dB) scaling on Mel spectra

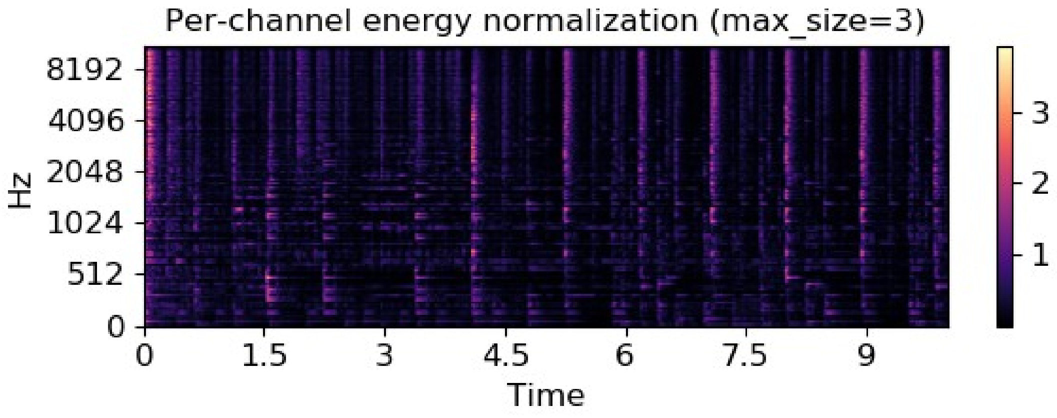
```
>>> import matplotlib.pyplot as plt
>>> y, sr = librosa.load(librosa.util.example_audio_file(),
...                      offset=10, duration=10)

>>> # 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])
```