plot_chroma

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Enhanced chroma and chroma variants

This notebook demonstrates a variety of techniques for enhancing chroma features and also, introduces chroma variants implemented in librosa.

Enhanced chroma ^^^^^^^^^^^^^^^ Beyond the default parameter settings of librosa's chroma functions, we apply the following enhancements:

1. Over-sampling the frequency axis to reduce sensitivity to tuning deviations
2. Harmonic-percussive-residual source separation to eliminate transients.
3. Nearest-neighbor smoothing to eliminate passing tones and sparse noise. This is inspired by the recurrence-based smoothing technique of `Cho and Bello, 2011 <http://ismir2011.ismir.net/papers/OS8-4.pdf>_`.
4. Local median filtering to suppress remaining discontinuities.

[ ]:
```
# Code source: Brian McFee
# License: ISC
# sphinx_gallery_thumbnail_number = 6

from __future__ import print_function
import numpy as np
import scipy
import matplotlib.pyplot as plt

import librosa
import librosa.display
```

We'll use a track that has harmonic, melodic, and percussive elements

[ ]:
```
y, sr = librosa.load('audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3')
```

First, let's plot the original chroma

[ ]:
```
chroma_orig = librosa.feature.chroma_cqt(y=y, sr=sr)

# For display purposes, let's zoom in on a 15-second chunk from the middle of␣
 ↪the song
```

```
idx = tuple([slice(None), slice(*list(librosa.time_to_frames([45, 60])))])

# And for comparison, we'll show the CQT matrix as well.
C = np.abs(librosa.cqt(y=y, sr=sr, bins_per_octave=12*3, n_bins=7*12*3))


plt.figure(figsize=(12, 4))
plt.subplot(2, 1, 1)
librosa.display.specshow(librosa.amplitude_to_db(C, ref=np.max)[idx],
                         y_axis='cqt_note', bins_per_octave=12*3)
plt.colorbar()
plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Original')
plt.tight_layout()
```

We can correct for minor tuning deviations by using 3 CQT bins per semi-tone, instead of one

[ ]:
```
chroma_os = librosa.feature.chroma_cqt(y=y, sr=sr, bins_per_octave=12*3)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Original')


plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_os[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('3x-over')
plt.tight_layout()
```

That cleaned up some rough edges, but we can do better by isolating the harmonic component. We'll use a large margin for separating harmonics from percussives

[ ]:
```
y_harm = librosa.effects.harmonic(y=y, margin=8)
chroma_os_harm = librosa.feature.chroma_cqt(y=y_harm, sr=sr,␣
 ↪bins_per_octave=12*3)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_os[idx], y_axis='chroma')
plt.colorbar()
```

```
plt.ylabel('3x-over')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_os_harm[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Harmonic')
plt.tight_layout()
```

There's still some noise in there though. We can clean it up using non-local filtering. This effectively removes any sparse additive noise from the features.

[ ]:
```
chroma_filter = np.minimum(chroma_os_harm,
                           librosa.decompose.nn_filter(chroma_os_harm,
                                                       aggregate=np.median,
                                                       metric='cosine'))


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_os_harm[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Harmonic')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_filter[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Non-local')
plt.tight_layout()
```

Local discontinuities and transients can be suppressed by using a horizontal median filter.

[ ]:
```
chroma_smooth = scipy.ndimage.median_filter(chroma_filter, size=(1, 9))


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_filter[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Non-local')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_smooth[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Median-filtered')
plt.tight_layout()
```

A final comparison between the CQT, original chromagram and the result of our filtering.

[ ]:
```
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
librosa.display.specshow(librosa.amplitude_to_db(C, ref=np.max)[idx],
                         y_axis='cqt_note', bins_per_octave=12*3)
plt.colorbar()
plt.ylabel('CQT')
plt.subplot(3, 1, 2)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.ylabel('Original')
plt.colorbar()
plt.subplot(3, 1, 3)
librosa.display.specshow(chroma_smooth[idx], y_axis='chroma', x_axis='time')
plt.ylabel('Processed')
plt.colorbar()
plt.tight_layout()
plt.show()
```

Chroma variants ^^^^^^^^^^^^^^^ There are three chroma variants implemented in librosa: `chroma_stft`, `chroma_cqt`, and `chroma_cens`. `chroma_stft` and `chroma_cqt` are two alternative ways of plotting chroma.

`chroma_stft` performs short-time fourier transform of an audio input and maps each STFT bin to chroma, while `chroma_cqt` uses constant-Q transform and maps each cq-bin to chroma.

A comparison between the STFT and the CQT methods for chromagram.

[ ]:
```
chromagram_stft = librosa.feature.chroma_stft(y=y, sr=sr)
chromagram_cqt = librosa.feature.chroma_cqt(y=y, sr=sr)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chromagram_stft[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('STFT')

plt.subplot(2, 1, 2)
librosa.display.specshow(chromagram_cqt[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('CQT')
plt.tight_layout()
```

CENS features (`chroma_cens`) are variants of chroma features introduced in `Müller and Ewart, 2011 <http://ismir2011.ismir.net/papers/PS2-8.pdf>_`, in which additional post processing steps are performed on the constant-Q chromagram to obtain features that are invariant to dynamics and timbre.

Thus, the CENS features are useful for applications, such as audio matching and retrieval.

Following steps are additional processing done on the chromagram, and are implemented in `chroma_cens`:

1. L1-Normalization across each chroma vector 2. Quantization of the amplitudes based on “log-

like" amplitude thresholds 3. Smoothing with sliding window (optional parameter) 4. Downsampling (not implemented)

A comparison between the original constant-Q chromagram and the CENS features.

[ ]:
```
chromagram_cens = librosa.feature.chroma_cens(y=y, sr=sr)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chromagram_cqt[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Orig')

plt.subplot(2, 1, 2)
librosa.display.specshow(chromagram_cens[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('CENS')
plt.tight_layout()
```

plot_hprss

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Harmonic-percussive source separation

This notebook illustrates how to separate an audio signal into its harmonic and percussive components.

We'll compare the original median-filtering based approach of Fitzgerald, 2010 <http://arrow.dit.ie/cgi/viewcontent.cgi?article=1078&context=argcon> *and its margin-based extension due to Dreidger, Mueller and Disch, 2014 <http://www.terasoft.com.tw/conf/ismir2014/proceedings/T110_127_Paper.pdf>*.

[ ]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

import librosa
import librosa.display
```

Load the example clip.

[ ]:
```
y, sr = librosa.load('audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3',␣
↪offset=40, duration=10)
```

Compute the short-time Fourier transform of y

[ ]:
```
D = librosa.stft(y)
```

Decompose D into harmonic and percussive components

$D = D_{\text{harmonic}} + D_{\text{percussive}}$

[ ]:
```
D_harmonic, D_percussive = librosa.decompose.hpss(D)
```

We can plot the two components along with the original spectrogram

[ ]:
```
# Pre-compute a global reference power from the input spectrum
rp = np.max(np.abs(D))

plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
```

```
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D), ref=rp),␣
 ↪y_axis='log')
plt.colorbar()
plt.title('Full spectrogram')

plt.subplot(3, 1, 2)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic), ref=rp),␣
 ↪y_axis='log')
plt.colorbar()
plt.title('Harmonic spectrogram')

plt.subplot(3, 1, 3)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive), ref=rp),␣
 ↪y_axis='log', x_axis='time')
plt.colorbar()
plt.title('Percussive spectrogram')
plt.tight_layout()
```

The default HPSS above assigns energy to each time-frequency bin according to whether a horizontal (harmonic) or vertical (percussive) filter responds higher at that position.

This assumes that all energy belongs to either a harmonic or percussive source, but does not handle "noise" well. Noise energy ends up getting spread between D_harmonic and D_percussive.

If we instead require that the horizontal filter responds more than the vertical filter *by at least some margin*, and vice versa, then noise can be removed from both components.

Note: the default (above) corresponds to margin=1

[ ]:
```
# Let's compute separations for a few different margins and compare the results␣
 ↪below
D_harmonic2, D_percussive2 = librosa.decompose.hpss(D, margin=2)
D_harmonic4, D_percussive4 = librosa.decompose.hpss(D, margin=4)
D_harmonic8, D_percussive8 = librosa.decompose.hpss(D, margin=8)
D_harmonic16, D_percussive16 = librosa.decompose.hpss(D, margin=16)
```

In the plots below, note that vibrato has been suppressed from the harmonic components, and vocals have been suppressed in the percussive components.

[ ]:
```
plt.figure(figsize=(10, 10))

plt.subplot(5, 2, 1)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic), ref=rp),␣
 ↪y_axis='log')
plt.title('Harmonic')
plt.yticks([])
plt.ylabel('margin=1')

plt.subplot(5, 2, 2)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive), ref=rp),␣
 ↪y_axis='log')
```

```
plt.title('Percussive')
plt.yticks([]), plt.ylabel('')

plt.subplot(5, 2, 3)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic2), ref=rp),␣
 ↪y_axis='log')
plt.yticks([])
plt.ylabel('margin=2')

plt.subplot(5, 2, 4)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive2),␣
 ↪ref=rp), y_axis='log')
plt.yticks([]) ,plt.ylabel('')

plt.subplot(5, 2, 5)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic4), ref=rp),␣
 ↪y_axis='log')
plt.yticks([])
plt.ylabel('margin=4')

plt.subplot(5, 2, 6)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive4),␣
 ↪ref=rp), y_axis='log')
plt.yticks([]), plt.ylabel('')

plt.subplot(5, 2, 7)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic8), ref=rp),␣
 ↪y_axis='log')
plt.yticks([])
plt.ylabel('margin=8')

plt.subplot(5, 2, 8)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive8),␣
 ↪ref=rp), y_axis='log')
plt.yticks([]), plt.ylabel('')

plt.subplot(5, 2, 9)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_harmonic16), ref=rp),␣
 ↪y_axis='log')
plt.yticks([])
plt.ylabel('margin=16')

plt.subplot(5, 2, 10)
librosa.display.specshow(librosa.amplitude_to_db(np.abs(D_percussive16),␣
 ↪ref=rp), y_axis='log')
plt.yticks([]), plt.ylabel('')
```

```
plt.tight_layout()
plt.show()
```

plot_music_sync

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Music Synchronization with Dynamic Time Warping

In this short tutorial, we demonstrate the use of dynamic time warping (DTW) for music synchronization which is implemented in `librosa`.

We assume that you are familiar with the algorithm and focus on the application. Further information about the algorithm can be found in the literature, e. g. [1].

Our example consists of two recordings of the first bars of the famous brass section lick in Stevie Wonder's rendition of "Sir Duke". Due to differences in tempo, the first recording lasts for ca. 7 seconds and the second recording for ca. 5 seconds. Our objective is now to find an alignment between these two recordings by using DTW.

[ ]:
```
# Code source: Stefan Balke
# License: ISC
# sphinx_gallery_thumbnail_number = 4

from __future__ import print_function
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

import librosa
import librosa.display
```

Load Audio Recordings

First, let's load a first version of our audio recordings.

[ ]:
```
x_1, fs = librosa.load('audio/sir_duke_slow.mp3')
plt.figure(figsize=(16, 4))
librosa.display.waveplot(x_1, sr=fs)
plt.title('Slower Version $X_1$')
plt.tight_layout()
```

And a second version, slightly faster.

[ ]:
```
x_2, fs = librosa.load('audio/sir_duke_fast.mp3')
plt.figure(figsize=(16, 4))
librosa.display.waveplot(x_2, sr=fs)
plt.title('Faster Version $X_2$')
plt.tight_layout()
```

Extract Chroma Features

[ ]:
```
n_fft = 4410
hop_size = 2205

x_1_chroma = librosa.feature.chroma_stft(y=x_1, sr=fs, tuning=0, norm=2,
                                         hop_length=hop_size, n_fft=n_fft)
x_2_chroma = librosa.feature.chroma_stft(y=x_2, sr=fs, tuning=0, norm=2,
                                         hop_length=hop_size, n_fft=n_fft)

plt.figure(figsize=(16, 8))
plt.subplot(2, 1, 1)
plt.title('Chroma Representation of $X_1$')
librosa.display.specshow(x_1_chroma, x_axis='time',
                         y_axis='chroma', cmap='gray_r', hop_length=hop_size)
plt.colorbar()
plt.subplot(2, 1, 2)
plt.title('Chroma Representation of $X_2$')
librosa.display.specshow(x_2_chroma, x_axis='time',
                         y_axis='chroma', cmap='gray_r', hop_length=hop_size)
plt.colorbar()
plt.tight_layout()
```

Align Chroma Sequences

[ ]:
```
D, wp = librosa.sequence.dtw(X=x_1_chroma, Y=x_2_chroma, metric='cosine')
wp_s = np.asarray(wp) * hop_size / fs

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111)
librosa.display.specshow(D, x_axis='time', y_axis='time',
                         cmap='gray_r', hop_length=hop_size)
imax = ax.imshow(D, cmap=plt.get_cmap('gray_r'),
                 origin='lower', interpolation='nearest', aspect='auto')
ax.plot(wp_s[:, 1], wp_s[:, 0], marker='o', color='r')
plt.title('Warping Path on Acc. Cost Matrix $D$')
plt.colorbar()
```

Alternative Visualization in the Time Domain

We can also visualize the warping path directly on our time domain signals. Red lines connect corresponding time positions in the input signals. (Thanks to F. Zalkow for the nice visualization.)

[ ]:
```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(16, 8))

# Plot x_1
librosa.display.waveplot(x_1, sr=fs, ax=ax1)
ax1.set(title='Slower Version $X_1$')

# Plot x_2
librosa.display.waveplot(x_2, sr=fs, ax=ax2)
ax2.set(title='Slower Version $X_2$')

plt.tight_layout()

trans_figure = fig.transFigure.inverted()
lines = []
arrows = 30
points_idx = np.int16(np.round(np.linspace(0, wp.shape[0] - 1, arrows)))

# for tp1, tp2 in zip((wp[points_idx, 0]) * hop_size, (wp[points_idx, 1]) *␣
 ↪hop_size):
for tp1, tp2 in wp[points_idx] * hop_size / fs:
    # get position on axis for a given index-pair
    coord1 = trans_figure.transform(ax1.transData.transform([tp1, 0]))
    coord2 = trans_figure.transform(ax2.transData.transform([tp2, 0]))

    # draw a line
    line = matplotlib.lines.Line2D((coord1[0], coord2[0]),
                                   (coord1[1], coord2[1]),
                                   transform=fig.transFigure,
                                   color='r')
    lines.append(line)

fig.lines = lines
plt.tight_layout()
```

Next steps...

Alright, you might ask where to go from here. Once we have the warping path between our two signals, we could realize different applications. One example is a player which enables you to navigate between different recordings of the same piece of music, e.g. one of Wagner's symphonies played by an orchestra or in a piano-reduced version.

Another example is that you could apply time scale modification algorithms, e.g. speed up the slower signal to the tempo of the faster one.

Literature

[1] Meinard Müller, Fundamentals of Music Processing — Audio, Analysis, Algorithms, Applications. Springer Verlag, 2015.

plot_pcen_stream

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 PCEN Streaming

This notebook demonstrates how to use streaming IO with `librosa.pcen` to do dynamic per-channel energy normalization on a spectrogram incrementally.

This is useful when processing long audio files that are too large to load all at once, or when streaming data from a recording device.

We'll need numpy and matplotlib for this example

[ ]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

import soundfile as sf

import librosa as librosa
import librosa.display as display
```

First, we'll start with an audio file that we want to stream

[ ]:
```
filename = librosa.util.example_audio_file()
```

Next, we'll set up the block reader to work on short segments of audio at a time.

[ ]:
```
# We'll generate 16 frames at a time, each frame having 4096 samples
# and 50% overlap.
#

n_fft = 4096
hop_length = n_fft // 2

# fill_value pads out the last frame with zeros so that we have a
# full frame at the end of the signal, even if the signal doesn't
# divide evenly into full frames.
sr = librosa.get_samplerate(filename)

stream = librosa.stream(filename, block_length=16,
                        frame_length=n_fft,
```

```
                        hop_length=hop_length,
                        mono=True,
                        fill_value=0)
```

For this example, we'll compute PCEN on each block, average over frequency, and store the results in a list.

[ ]:
```
# Make an array to store the frequency-averaged PCEN values
pcen_blocks = []

# Initialize the PCEN filter delays to steady state
zi = None

for y_block in stream:
    # Compute the STFT (without padding, so center=False)
    D = librosa.stft(y_block, n_fft=n_fft, hop_length=hop_length,
                     center=False)

    # Compute PCEN on the magnitude spectrum, using initial delays
    # returned from our previous call (if any)
    # store the final delays for use as zi in the next iteration
    P, zi = librosa.pcen(np.abs(D), sr=sr, hop_length=hop_length,
                         zi=zi, return_zf=True)

    # Compute the average PCEN over frequency, and append it to our list
    pcen_blocks.extend(np.mean(P, axis=0))

# Cast to a numpy array for use downstream
pcen_blocks = np.asarray(pcen_blocks)
```

For the sake of comparison, let's see how it would look had we run PCEN on the entire spectrum without block-wise processing

[ ]:
```
y, sr = librosa.load(filename, sr=44100)

# Keep the same parameters as before
D = librosa.stft(y, n_fft=n_fft, hop_length=hop_length, center=False)

# Compute pcen on the magnitude spectrum.
# We don't need to worry about initial and final filter delays if
# we're doing everything in one go.
P = librosa.pcen(np.abs(D), sr=sr, hop_length=hop_length)

pcen_full = np.mean(P, axis=0)
```

Plot the PCEN spectrum and the resulting magnitudes

[ ]:
```
plt.figure()
# First, plot the spectrum
ax = plt.subplot(2,1,1)
```

```
librosa.display.specshow(P, sr=sr, hop_length=hop_length, x_axis='time',␣
 ↪y_axis='log')
plt.title('PCEN spectrum')

# Now we'll plot the pcen curves
plt.subplot(2,1,2, sharex=ax)
times = librosa.times_like(pcen_full, sr=sr, hop_length=hop_length)
plt.plot(times, pcen_full, linewidth=3, alpha=0.25, label='Full signal PCEN')
times = librosa.times_like(pcen_blocks, sr=sr, hop_length=hop_length)
plt.plot(times, pcen_blocks, linestyle=':', label='Block-wise PCEN')
plt.legend()

# Zoom in to a short patch to see the fine details
plt.xlim([30, 40])

# render the plot
plt.tight_layout()
plt.show()
```

plot_presets

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Presets

This notebook demonstrates how to use the `presets` package to change the default parameters for librosa.

[ ]:
```
# Code source: Brian McFee
# License: ISC
```

We'll need numpy and matplotlib for this example

[ ]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

# Import the Preset class
from presets import Preset


# To use presets, we'll make a dummy import of librosa
# and the display submodule here.
import librosa as _librosa
import librosa.display as _display
# The assignment here is to circumvent python's inability
# to rename submodule imports within the package
_librosa.display = _display
```

By default, librosa uses the following parameters across all functions: - `sr=22050` (sampling rate) - `hop_length=512` (number of samples between frames) - `n_fft=2048` (number of samples per frame in STFT-like analyses)

You may want to change these values to suit your application, but doing so consistently in every function call can be somewhat cumbersome.

Presets makes it easy to do this all at once by wrapping the module and all function calls, and overriding default arguments.

[ ]:
```
# First, we need to set up the preset-wrapped librosa import

librosa = Preset(_librosa)
```

```
# To change the default sampling rate, we can set the `sr` entry:
librosa['sr'] = 44100

# and similarly for hop_length and n_fft
librosa['hop_length'] = 1024
librosa['n_fft'] = 4096

# In general, when you set `librosa['X']` for any string `X`, anywhere within
# librosa where the parameter `X` occurs as a keyword-argument,
# its default value will be replaced by whatever value you provide.
```

Now we can load in a file and do some analysis with the new defaults

[ ]:
```
filename = 'audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3'

y, sr = librosa.load(filename, duration=5, offset=35)

# Generate a Mel spectrogram:

M = librosa.feature.melspectrogram(y=y)

# Of course, you can still override the new default manually, e.g.:

M_highres = librosa.feature.melspectrogram(y=y, hop_length=512)


# And plot the results
plt.figure(figsize=(6, 6))
ax = plt.subplot(3, 1, 1)

librosa.display.specshow(librosa.power_to_db(M, ref=np.max),
                         y_axis='mel', x_axis='time')

plt.title('44100/1024/4096')

plt.subplot(3, 1, 2, sharex=ax, sharey=ax)
librosa.display.specshow(librosa.power_to_db(M_highres, ref=np.max),
                         hop_length=512,
                         y_axis='mel', x_axis='time')
plt.title('44100/512/4096')

# We can repeat the whole process with different defaults, just by
# updating the parameter entries
librosa['sr'] = 11025

y2, sr2 = librosa.load(filename, duration=5, offset=35)
M2 = librosa.feature.melspectrogram(y=y2, sr=sr2)
```

```
plt.subplot(3, 1, 3, sharex=ax, sharey=ax)
librosa.display.specshow(librosa.power_to_db(M2, ref=np.max),
                         y_axis='mel', x_axis='time')

plt.title('11025/1024/4096')

plt.tight_layout()
plt.show()
```

plot_segmentation

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Laplacian segmentation

This notebook implements the laplacian segmentation method of `McFee and Ellis, 2014 <http://bmcfee.github.io/papers/ismir2014_spectral.pdf>_`, with a couple of minor stability improvements.

Throughout the example, we will refer to equations in the paper by number, so it will be helpful to read along.

[ ]:
```
# Code source: Brian McFee
# License: ISC
```

Imports - numpy for basic functionality - scipy for graph Laplacian - matplotlib for visualization - sklearn.cluster for K-Means

[ ]:
```
from __future__ import print_function

import numpy as np
import scipy
import matplotlib.pyplot as plt

import sklearn.cluster

import librosa
import librosa.display
```

First, we'll load in a song

[ ]:
```
y, sr = librosa.load('audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3')
```

Next, we'll compute and plot a log-power CQT

[ ]:
```
BINS_PER_OCTAVE = 12 * 3
N_OCTAVES = 7
C = librosa.amplitude_to_db(np.abs(librosa.cqt(y=y, sr=sr,
                                               bins_per_octave=BINS_PER_OCTAVE,
                                               n_bins=N_OCTAVES * BINS_PER_OCTAVE)),
                            ref=np.max)
```

```
plt.figure(figsize=(12, 4))
librosa.display.specshow(C, y_axis='cqt_hz', sr=sr,
                         bins_per_octave=BINS_PER_OCTAVE,
                         x_axis='time')
plt.tight_layout()
```

To reduce dimensionality, we'll beat-synchronous the CQT

[ ]:
```
tempo, beats = librosa.beat.beat_track(y=y, sr=sr, trim=False)
Csync = librosa.util.sync(C, beats, aggregate=np.median)

# For plotting purposes, we'll need the timing of the beats
# we fix_frames to include non-beat frames 0 and C.shape[1] (final frame)
beat_times = librosa.frames_to_time(librosa.util.fix_frames(beats,
                                                            x_min=0,
                                                            x_max=C.shape[1]),
                                    sr=sr)

plt.figure(figsize=(12, 4))
librosa.display.specshow(Csync, bins_per_octave=12*3,
                         y_axis='cqt_hz', x_axis='time',
                         x_coords=beat_times)
plt.tight_layout()
```

Let's build a weighted recurrence matrix using beat-synchronous CQT (Equation 1) width=3 prevents links within the same bar mode='affinity' here implements S_rep (after Eq. 8)

[ ]:
```
R = librosa.segment.recurrence_matrix(Csync, width=3, mode='affinity',
                                      sym=True)

# Enhance diagonals with a median filter (Equation 2)
df = librosa.segment.timelag_filter(scipy.ndimage.median_filter)
Rf = df(R, size=(1, 7))
```

Now let's build the sequence matrix (S_loc) using mfcc-similarity
$R_{\text{path}}[i, i \pm 1] = \exp(-\|C_i - C_{i\pm1}\|^2/\sigma^2)$
Here, we take σ to be the median distance between successive beats.

[ ]:
```
mfcc = librosa.feature.mfcc(y=y, sr=sr)
Msync = librosa.util.sync(mfcc, beats)

path_distance = np.sum(np.diff(Msync, axis=1)**2, axis=0)
sigma = np.median(path_distance)
path_sim = np.exp(-path_distance / sigma)

R_path = np.diag(path_sim, k=1) + np.diag(path_sim, k=-1)
```

And compute the balanced combination (Equations 6, 7, 9)

[ ]:
```
deg_path = np.sum(R_path, axis=1)
deg_rec = np.sum(Rf, axis=1)
```

```
mu = deg_path.dot(deg_path + deg_rec) / np.sum((deg_path + deg_rec)**2)

A = mu * Rf + (1 - mu) * R_path
```

Plot the resulting graphs (Figure 1, left and center)

[ ]:
```
plt.figure(figsize=(8, 4))
plt.subplot(1, 3, 1)
librosa.display.specshow(Rf, cmap='inferno_r', y_axis='time',
                         y_coords=beat_times)
plt.title('Recurrence similarity')
plt.subplot(1, 3, 2)
librosa.display.specshow(R_path, cmap='inferno_r')
plt.title('Path similarity')
plt.subplot(1, 3, 3)
librosa.display.specshow(A, cmap='inferno_r')
plt.title('Combined graph')
plt.tight_layout()
```

Now let's compute the normalized Laplacian (Eq. 10)

[ ]:
```
L = scipy.sparse.csgraph.laplacian(A, normed=True)


# and its spectral decomposition
evals, evecs = scipy.linalg.eigh(L)


# We can clean this up further with a median filter.
# This can help smooth over small discontinuities
evecs = scipy.ndimage.median_filter(evecs, size=(9, 1))


# cumulative normalization is needed for symmetric normalize laplacian␣
 ↪eigenvectors
Cnorm = np.cumsum(evecs**2, axis=1)**0.5

# If we want k clusters, use the first k normalized eigenvectors.
# Fun exercise: see how the segmentation changes as you vary k

k = 5

X = evecs[:, :k] / Cnorm[:, k-1:k]


# Plot the resulting representation (Figure 1, center and right)

plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 2)
```

```
librosa.display.specshow(Rf, cmap='inferno_r')
plt.title('Recurrence matrix')

plt.subplot(1, 2, 1)
librosa.display.specshow(X,
                         y_axis='time',
                         y_coords=beat_times)
plt.title('Structure components')
plt.tight_layout()
```

Let's use these k components to cluster beats into segments (Algorithm 1)

[ ]:
```
KM = sklearn.cluster.KMeans(n_clusters=k)

seg_ids = KM.fit_predict(X)


# and plot the results
plt.figure(figsize=(12, 4))
colors = plt.get_cmap('Paired', k)

plt.subplot(1, 3, 2)
librosa.display.specshow(Rf, cmap='inferno_r')
plt.title('Recurrence matrix')
plt.subplot(1, 3, 1)
librosa.display.specshow(X,
                         y_axis='time',
                         y_coords=beat_times)
plt.title('Structure components')
plt.subplot(1, 3, 3)
librosa.display.specshow(np.atleast_2d(seg_ids).T, cmap=colors)
plt.title('Estimated segments')
plt.colorbar(ticks=range(k))
plt.tight_layout()
```

Locate segment boundaries from the label sequence

[ ]:
```
bound_beats = 1 + np.flatnonzero(seg_ids[:-1] != seg_ids[1:])

# Count beat 0 as a boundary
bound_beats = librosa.util.fix_frames(bound_beats, x_min=0)

# Compute the segment label for each boundary
bound_segs = list(seg_ids[bound_beats])

# Convert beat indices to frames
bound_frames = beats[bound_beats]

# Make sure we cover to the end of the track
```

```
bound_frames = librosa.util.fix_frames(bound_frames,
                                       x_min=None,
                                       x_max=C.shape[1]-1)
```

And plot the final segmentation over original CQT

[ ]:
```
# sphinx_gallery_thumbnail_number = 5

import matplotlib.patches as patches
plt.figure(figsize=(12, 4))

bound_times = librosa.frames_to_time(bound_frames)
freqs = librosa.cqt_frequencies(n_bins=C.shape[0],
                                fmin=librosa.note_to_hz('C1'),
                                bins_per_octave=BINS_PER_OCTAVE)

librosa.display.specshow(C, y_axis='cqt_hz', sr=sr,
                         bins_per_octave=BINS_PER_OCTAVE,
                         x_axis='time')
ax = plt.gca()

for interval, label in zip(zip(bound_times, bound_times[1:]), bound_segs):
    ax.add_patch(patches.Rectangle((interval[0], freqs[0]),
                                   interval[1] - interval[0],
                                   freqs[-1],
                                   facecolor=colors(label),
                                   alpha=0.50))

plt.tight_layout()
plt.show()
```

plot_superflux

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Superflux onsets

This notebook demonstrates how to recover the Superflux onset detection algorithm of `Boeck and Widmer, 2013 <http://dafx13.nuim.ie/papers/09.dafx2013_submission_12.pdf>_` from librosa.

This algorithm improves onset detection accuracy in the presence of vibrato.

[ ]:
```
# Code source: Brian McFee
# License: ISC
```

We'll need numpy and matplotlib for this example

[ ]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

import librosa
import librosa.display
```

We'll load in a five-second clip of a track that has noticeable vocal vibrato. The method works fine for longer signals, but the results are harder to visualize.

[ ]:
```
y, sr = librosa.load('audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3',
                     sr=44100,
                     duration=5,
                     offset=35)
```

These parameters are taken directly from the paper

[ ]:
```
n_fft = 1024
hop_length = int(librosa.time_to_samples(1./200, sr=sr))
lag = 2
n_mels = 138
fmin = 27.5
fmax = 16000.
max_size = 3
```

The paper uses a log-frequency representation, but for simplicity, we'll use a Mel spectrogram instead.

[ ]:
```
S = librosa.feature.melspectrogram(y, sr=sr, n_fft=n_fft,
                                   hop_length=hop_length,
                                   fmin=fmin,
                                   fmax=fmax,
                                   n_mels=n_mels)


plt.figure(figsize=(6, 4))
librosa.display.specshow(librosa.power_to_db(S, ref=np.max),
                         y_axis='mel', x_axis='time', sr=sr,
                         hop_length=hop_length, fmin=fmin, fmax=fmax)
plt.tight_layout()
```

Now we'll compute the onset strength envelope and onset events using the librosa defaults.

[ ]:
```
odf_default = librosa.onset.onset_strength(y=y, sr=sr, hop_length=hop_length)
onset_default = librosa.onset.onset_detect(y=y, sr=sr, hop_length=hop_length,
                                           units='time')
```

And similarly with the superflux method

[ ]:
```
odf_sf = librosa.onset.onset_strength(S=librosa.power_to_db(S, ref=np.max),
                                      sr=sr,
                                      hop_length=hop_length,
                                      lag=lag, max_size=max_size)

onset_sf = librosa.onset.onset_detect(onset_envelope=odf_sf,
                                      sr=sr,
                                      hop_length=hop_length,
                                      units='time')
```

If you look carefully, the default onset detector (top sub-plot) has several false positives in high-vibrato regions, eg around 0.62s or 1.80s.

The superflux method (middle plot) is less susceptible to vibrato, and does not detect onset events at those points.

[ ]:
```
# sphinx_gallery_thumbnail_number = 2
plt.figure(figsize=(6, 6))

frame_time = librosa.frames_to_time(np.arange(len(odf_default)),
                                    sr=sr,
                                    hop_length=hop_length)

ax = plt.subplot(2, 1, 2)
librosa.display.specshow(librosa.power_to_db(S, ref=np.max),
                         y_axis='mel', x_axis='time', sr=sr,
                         hop_length=hop_length, fmin=fmin, fmax=fmax)
plt.xlim([0, 5.0])
plt.axis('tight')
```

```
plt.subplot(4, 1, 1, sharex=ax)
plt.plot(frame_time, odf_default, label='Spectral flux')
plt.vlines(onset_default, 0, odf_default.max(), label='Onsets')
plt.xlim([0, 5.0])
plt.legend()


plt.subplot(4, 1, 2, sharex=ax)
plt.plot(frame_time, odf_sf, color='g', label='Superflux')
plt.vlines(onset_sf, 0, odf_sf.max(), label='Onsets')
plt.xlim([0, 5.0])
plt.legend()

plt.tight_layout()
plt.show()
```

plot_viterbi

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Viterbi decoding

This notebook demonstrates how to use Viterbi decoding to impose temporal smoothing on frame-wise state predictions.

Our working example will be the problem of silence/non-silence detection.

[ ]:
```
# Code source: Brian McFee
# License: ISC

###################
# Standard imports
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import librosa

import librosa.display
```

Load an example signal

[ ]:
```
y, sr = librosa.load('audio/sir_duke_slow.mp3')


# And compute the spectrogram magnitude and phase
S_full, phase = librosa.magphase(librosa.stft(y))


####################
# Plot the spectrum
plt.figure(figsize=(12, 4))
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.colorbar()
plt.tight_layout()
```

As you can see, there are periods of silence and non-silence throughout this recording.

[ ]:
```
# As a first step, we can plot the root-mean-square (RMS) curve
rms = librosa.feature.rms(y=y)[0]

times = librosa.frames_to_time(np.arange(len(rms)))

plt.figure(figsize=(12, 4))
plt.plot(times, rms)
plt.axhline(0.02, color='r', alpha=0.5)
plt.xlabel('Time')
plt.ylabel('RMS')
plt.axis('tight')
plt.tight_layout()

# The red line at 0.02 indicates a reasonable threshold for silence detection.
# However, the RMS curve occasionally dips below the threshold momentarily,
# and we would prefer the detector to not count these brief dips as silence.
# This is where the Viterbi algorithm comes in handy!
```

As a first step, we will convert the raw RMS score into a likelihood (probability) by logistic mapping

$P[V = 1|x] = \frac{\exp(x-\tau)}{1+\exp(x-\tau)}$

where x denotes the RMS value and $\tau = 0.02$ is our threshold. The variable V indicates whether the signal is non-silent (1) or silent (0).

We'll normalize the RMS by its standard deviation to expand the range of the probability vector

[ ]:
```
r_normalized = (rms - 0.02) / np.std(rms)
p = np.exp(r_normalized) / (1 + np.exp(r_normalized))

# We can plot the probability curve over time:

plt.figure(figsize=(12, 4))
plt.plot(times, p, label='P[V=1|x]')
plt.axhline(0.5, color='r', alpha=0.5, label='Descision threshold')
plt.xlabel('Time')
plt.axis('tight')
plt.legend()
plt.tight_layout()
```

which looks much like the first plot, but with the decision threshold shifted to 0.5. A simple silence detector would classify each frame independently of its neighbors, which would result in the following plot:

[ ]:
```
plt.figure(figsize=(12, 6))
ax = plt.subplot(2,1,1)
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.subplot(2,1,2, sharex=ax)
plt.step(times, p>=0.5, label='Non-silent')
plt.xlabel('Time')
```

```
plt.axis('tight')
plt.ylim([0, 1.05])
plt.legend()
plt.tight_layout()
```

We can do better using the Viterbi algorithm. We'll use state 0 to indicate silent, and 1 to indicate non-silent. We'll assume that a silent frame is equally likely to be followed by silence or non-silence, but that non-silence is slightly more likely to be followed by non-silence. This is accomplished by building a self-loop transition matrix, where `transition[i, j]` is the probability of moving from state `i` to state `j` in the next frame.

[ ]:
```
transition = librosa.sequence.transition_loop(2, [0.5, 0.6])
print(transition)
```

Our `p` variable only indicates the probability of non-silence, so we need to also compute the probability of silence as its complement.

[ ]:
```
full_p = np.vstack([1 - p, p])
print(full_p)
```

Now, we're ready to decode! We'll use `viterbi_discriminative` here, since the inputs are state likelihoods conditional on data (in our case, data is rms).

[ ]:
```
states = librosa.sequence.viterbi_discriminative(full_p, transition)

# sphinx_gallery_thumbnail_number = 5
plt.figure(figsize=(12, 6))
ax = plt.subplot(2,1,1)
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.xlabel('')
ax.tick_params(labelbottom=False)
plt.subplot(2, 1, 2, sharex=ax)
plt.step(times, p>=0.5, label='Frame-wise')
plt.step(times, states, linestyle='--', color='orange', label='Viterbi')
plt.xlabel('Time')
plt.axis('tight')
plt.ylim([0, 1.05])
plt.legend()
```

Note how the Viterbi output has fewer state changes than the frame-wise predictor, and it is less sensitive to momentary dips in energy. This is controlled directly by the transition matrix. A higher self-transition probability means that the decoder is less likely to change states.

plot_vocal_separation

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Vocal separation

This notebook demonstrates a simple technique for separating vocals (and other sporadic foreground signals) from accompanying instrumentation.

This is based on the "REPET-SIM" method of `Rafii and Pardo, 2012 <http://www.cs.northwestern.edu/~zra446/doc/Rafii-Pardo%20-%20Music-Voice%20Separation%20using%` but includes a couple of modifications and extensions:

```
- FFT windows overlap by 1/4, instead of 1/2
- Non-local filtering is converted into a soft mask by Wiener filtering.
  This is similar in spirit to the soft-masking method used by `Fitzgerald, 2012
  <http://arrow.dit.ie/cgi/viewcontent.cgi?article=1086&context=argcon>`_,
  but is a bit more numerically stable in practice.
```

[ ]:
```
# Code source: Brian McFee
# License: ISC

###################
# Standard imports
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import librosa

import librosa.display
```

Load an example with vocals.

[ ]:
```
y, sr = librosa.load('audio/
 ↪Cheese_N_Pot-C_-_16_-_The_Raps_Well_Clean_Album_Version.mp3', duration=120)


# And compute the spectrogram magnitude and phase
S_full, phase = librosa.magphase(librosa.stft(y))
```

Plot a 5-second slice of the spectrum

[ ]:
```
idx = slice(*librosa.time_to_frames([30, 35], sr=sr))
plt.figure(figsize=(12, 4))
librosa.display.specshow(librosa.amplitude_to_db(S_full[:, idx], ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.colorbar()
plt.tight_layout()
```

The wiggly lines above are due to the vocal component. Our goal is to separate them from the accompanying instrumentation.

[ ]:
```
# We'll compare frames using cosine similarity, and aggregate similar frames
# by taking their (per-frequency) median value.
#
# To avoid being biased by local continuity, we constrain similar frames to be
# separated by at least 2 seconds.
#
# This suppresses sparse/non-repetetitive deviations from the average spectrum,
# and works well to discard vocal elements.

S_filter = librosa.decompose.nn_filter(S_full,
                                       aggregate=np.median,
                                       metric='cosine',
                                       width=int(librosa.time_to_frames(2,␣
 ↪sr=sr)))

# The output of the filter shouldn't be greater than the input
# if we assume signals are additive.  Taking the pointwise minimium
# with the input spectrum forces this.
S_filter = np.minimum(S_full, S_filter)
```

The raw filter output can be used as a mask, but it sounds better if we use soft-masking.

[ ]:
```
# We can also use a margin to reduce bleed between the vocals and␣
 ↪instrumentation masks.
# Note: the margins need not be equal for foreground and background separation
margin_i, margin_v = 2, 10
power = 2

mask_i = librosa.util.softmask(S_filter,
                               margin_i * (S_full - S_filter),
                               power=power)

mask_v = librosa.util.softmask(S_full - S_filter,
                               margin_v * S_filter,
                               power=power)

# Once we have the masks, simply multiply them with the input spectrum
# to separate the components
```

```
S_foreground = mask_v * S_full
S_background = mask_i * S_full
```

Plot the same slice, but separated into its foreground and background

[ ]:
```
# sphinx_gallery_thumbnail_number = 2

plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
librosa.display.specshow(librosa.amplitude_to_db(S_full[:, idx], ref=np.max),
                         y_axis='log', sr=sr)
plt.title('Full spectrum')
plt.colorbar()

plt.subplot(3, 1, 2)
librosa.display.specshow(librosa.amplitude_to_db(S_background[:, idx], ref=np.
 ↪max),
                         y_axis='log', sr=sr)
plt.title('Background')
plt.colorbar()
plt.subplot(3, 1, 3)
librosa.display.specshow(librosa.amplitude_to_db(S_foreground[:, idx], ref=np.
 ↪max),
                         y_axis='log', x_axis='time', sr=sr)
plt.title('Foreground')
plt.colorbar()
plt.tight_layout()
plt.show()
```

In [ ]:

```
%matplotlib inline
```

Enhanced chroma and chroma variants

This notebook demonstrates a variety of techniques for enhancing chroma features and also, introduces chroma variants implemented in librosa.

Enhanced chroma ^^^^^^^^^^^^^^^^ Beyond the default parameter settings of librosa's chroma functions, we apply the following enhancements:

1. Over-sampling the frequency axis to reduce sensitivity to tuning deviations
2. Harmonic-percussive-residual source separation to eliminate transients.
3. Nearest-neighbor smoothing to eliminate passing tones and sparse noise. This is inspired by the recurrence-based smoothing technique of Cho and Bello, 2011 <http://ismir2011.ismir.net/papers/OS8-4.pdf> _.
4. Local median filtering to suppress remaining discontinuities.

In [ ]:

```
# Code source: Brian McFee
# License: ISC
# sphinx_gallery_thumbnail_number = 6

from __future__ import print_function
import numpy as np
import scipy
import matplotlib.pyplot as plt

import librosa
import librosa.display
```

We'll use a track that has harmonic, melodic, and percussive elements

In [ ]:

```
y, sr = librosa.load('audio/Karissa_Hobbs_-_09_-_Lets_Go_Fishin.mp3')
```

First, let's plot the original chroma

In [ ]:

```
chroma_orig = librosa.feature.chroma_cqt(y=y, sr=sr)

# For display purposes, let's zoom in on a 15-second chunk from the middle of the song
idx = tuple([slice(None), slice(*list(librosa.time_to_frames([45, 60])))])

# And for comparison, we'll show the CQT matrix as well.
C = np.abs(librosa.cqt(y=y, sr=sr, bins_per_octave=12*3, n_bins=7*12*3))


plt.figure(figsize=(12, 4))
plt.subplot(2, 1, 1)
librosa.display.specshow(librosa.amplitude_to_db(C, ref=np.max)[idx],
                         y_axis='cqt_note', bins_per_octave=12*3)
plt.colorbar()
plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Original')
plt.tight_layout()
```

We can correct for minor tuning deviations by using 3 CQT bins per semi-tone, instead of one

In [ ]:

```
chroma_os = librosa.feature.chroma_cqt(y=y, sr=sr, bins_per_octave=12*3)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Original')


plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_os[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('3x-over')
plt.tight_layout()
```

That cleaned up some rough edges, but we can do better by isolating the harmonic component. We'll use a large margin for separating harmonics from percussives

In [ ]:

```
y_harm = librosa.effects.harmonic(y=y, margin=8)
chroma_os_harm = librosa.feature.chroma_cqt(y=y_harm, sr=sr, bins_per_octave=12*3)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_os[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('3x-over')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_os_harm[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Harmonic')
plt.tight_layout()
```

There's still some noise in there though. We can clean it up using non-local filtering. This effectively removes any sparse additive noise from the features.

In [ ]:

```
chroma_filter = np.minimum(chroma_os_harm,
                           librosa.decompose.nn_filter(chroma_os_harm,
                                                       aggregate=np.median,
                                                       metric='cosine'))


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_os_harm[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Harmonic')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_filter[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Non-local')
plt.tight_layout()
```

Local discontinuities and transients can be suppressed by using a horizontal median filter.

In [ ]:

```
chroma_smooth = scipy.ndimage.median_filter(chroma_filter, size=(1, 9))
```

```
plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chroma_filter[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Non-local')

plt.subplot(2, 1, 2)
librosa.display.specshow(chroma_smooth[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('Median-filtered')
plt.tight_layout()
```

A final comparison between the CQT, original chromagram and the result of our filtering.

In [ ]:

```
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
librosa.display.specshow(librosa.amplitude_to_db(C, ref=np.max)[idx],
                         y_axis='cqt_note', bins_per_octave=12*3)
plt.colorbar()
plt.ylabel('CQT')
plt.subplot(3, 1, 2)
librosa.display.specshow(chroma_orig[idx], y_axis='chroma')
plt.ylabel('Original')
plt.colorbar()
plt.subplot(3, 1, 3)
librosa.display.specshow(chroma_smooth[idx], y_axis='chroma', x_axis='time')
plt.ylabel('Processed')
plt.colorbar()
plt.tight_layout()
plt.show()
```

Chroma variants ^^^^^^^^^^^^^^^ There are three chroma variants implemented in librosa: `chroma_stft`, `chroma_cqt`, and `chroma_cens`. `chroma_stft` and `chroma_cqt` are two alternative ways of plotting chroma.

`chroma_stft` performs short-time fourier transform of an audio input and maps each STFT bin to chroma, while `chroma_cqt` uses constant-Q transform and maps each cq-bin to chroma.

A comparison between the STFT and the CQT methods for chromagram.

In [ ]:

```
chromagram_stft = librosa.feature.chroma_stft(y=y, sr=sr)
chromagram_cqt = librosa.feature.chroma_cqt(y=y, sr=sr)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chromagram_stft[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('STFT')

plt.subplot(2, 1, 2)
librosa.display.specshow(chromagram_cqt[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('CQT')
plt.tight_layout()
```

CENS features (`chroma_cens`) are variants of chroma features introduced in `Müller and Ewart, 2011 <http://ismir2011.ismir.net/papers/PS2-8.pdf> _`, in which additional post processing steps are performed on the constant-Q chromagram to obtain features that are invariant to dynamics and timbre.

Thus, the CENS features are useful for applications, such as audio matching and retrieval.

Following steps are additional processing done on the chromagram, and are implemented in `chroma_cens`:

1. L1-Normalization across each chroma vector
2. Quantization of the amplitudes based on "log-like" amplitude thresholds

2. Quantization of the amplitudes based on "log-like" amplitude thresholds
3. Smoothing with sliding window (optional parameter)
4. Downsampling (not implemented)

A comparison between the original constant-Q chromagram and the CENS features.

In [ ]:

```
chromagram_cens = librosa.feature.chroma_cens(y=y, sr=sr)


plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
librosa.display.specshow(chromagram_cqt[idx], y_axis='chroma')
plt.colorbar()
plt.ylabel('Orig')

plt.subplot(2, 1, 2)
librosa.display.specshow(chromagram_cens[idx], y_axis='chroma', x_axis='time')
plt.colorbar()
plt.ylabel('CENS')
plt.tight_layout()
```