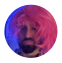
librosa: Audio and Music Signal Analysis in Python

Conference Paper · January 2015

DOI: 10.25080/Majora-7b98e3ed-003

Dawen Liang
Columbia University
10 PUBLICATIONS **3,227** CITATIONS
SEE PROFILE

Matt Mcvicar
University of Bristol
24 PUBLICATIONS **3,064** CITATIONS
SEE PROFILE


Oriol Nieto
Adobe Research
41 PUBLICATIONS **4,039** CITATIONS
SEE PROFILE

librosa: Audio and Music Signal Analysis in Python

Brian McFee[¶][||][*], Colin Raffel[§], Dawen Liang[§], Daniel P.W. Ellis[§], Matt McVicar[‡], Eric Battenberg[**], Oriol Nieto[||]

Abstract—This document describes version 0.4.0 of librosa: a Python package for audio and music signal processing. At a high level, librosa provides implementations of a variety of common functions used throughout the field of music information retrieval. In this document, a brief overview of the library's functionality is provided, along with explanations of the design goals, software development practices, and notational conventions.

Index Terms—audio, music, signal processing

Introduction

The emerging research field of music information retrieval (MIR) broadly covers topics at the intersection of musicology, digital signal processing, machine learning, information retrieval, and library science. Although the field is relatively young—the first international symposium on music information retrieval (ISMIR)[1] was held in October of 2000—it is rapidly developing, thanks in part to the proliferation and practical scientific needs of digital music services, such as iTunes, Pandora, and Spotify. While the preponderance of MIR research has been conducted with custom tools and scripts developed by researchers in a variety of languages such as MATLAB or C++, the stability, scalability, and ease of use these tools has often left much to be desired.

In recent years, interest has grown within the MIR community in using (scientific) Python as a viable alternative. This has been driven by a confluence of several factors, including the availability of high-quality machine learning libraries such as `scikit-learn` [Pedregosa11] and tools based on `Theano` [Bergstra11], as well as Python's vast catalog of packages for dealing with text data and web services. However, the adoption of Python has been slowed by the absence of a stable core library that provides the basic routines upon which many MIR applications are built. To remedy this situation, we have developed librosa:[2] a Python package for audio and music signal processing.[3] In doing so, we hope to both ease the transition of MIR researchers into Python (and modern software development practices), and also to make core MIR techniques readily available to the broader community of scientists and Python programmers.

* *Corresponding author: brian.mcfee@nyu.edu*
¶ *Center for Data Science, New York University*
|| *Music and Audio Research Laboratory, New York University*
§ *LabROSA, Columbia University*
‡ *Department of Engineering Mathematics, University of Bristol*
** *Silicon Valley AI Lab, Baidu, Inc.*

Design principles

In designing librosa, we have prioritized a few key concepts. First, we strive for a low barrier to entry for researchers familiar with MATLAB. In particular, we opted for a relatively flat package layout, and following `scipy` [Jones01] rely upon `numpy` data types and functions [VanDerWalt11], rather than abstract class hierarchies.

Second, we expended considerable effort in standardizing interfaces, variable names, and (default) parameter settings across the various analysis functions. This task was complicated by the fact that reference implementations from which our implementations are derived come from various authors, and are often designed as one-off scripts rather than proper library functions with well-defined interfaces.

Third, wherever possible, we retain backwards compatibility against existing reference implementations. This is achieved via regression testing for numerical equivalence of outputs. All tests are implemented in the `nose` framework.[4]

Fourth, because MIR is a rapidly evolving field, we recognize that the exact implementations provided by librosa may not represent the state of the art for any particular task. Consequently, functions are designed to be *modular*, allowing practitioners to provide their own functions when appropriate, e.g., a custom onset strength estimate may be provided to the beat tracker as a function argument. This allows researchers to leverage existing library functions while experimenting with improvements to specific components. Although this seems simple and obvious, from a practical standpoint the monolithic designs and lack of interoperability between different research codebases have historically made this difficult.

Finally, we strive for readable code, thorough documentation and exhaustive testing. All development is conducted on GitHub. We apply modern software development practices, such as continuous integration testing (via Travis[5]) and coverage (via Coveralls[6]). All functions are implemented in pure Python, thoroughly documented using Sphinx, and include example code demonstrating usage. The implementation mostly complies with PEP-8 recommendations, with a small

1. http://ismir.net
2. https://github.com/bmcfee/librosa
3. The name *librosa* is borrowed from *LabROSA*: the LABoratory for the Recognition and Organization of Speech and Audio at Columbia University, where the initial development of librosa took place.
4. https://nose.readthedocs.org/en/latest/

set of exceptions for variable names that make the code more concise without sacrificing clarity: e.g., `y` and `sr` are preferred over more verbose names such as `audio_buffer` and `sampling_rate`.

Conventions

In general, librosa's functions tend to expose all relevant parameters to the caller. While this provides a great deal of flexibility to expert users, it can be overwhelming to novice users who simply need a consistent interface to process audio files. To satisfy both needs, we define a set of general conventions and standardized default parameter values shared across many functions.

An audio signal is represented as a one-dimensional `numpy` array, denoted as `y` throughout librosa. Typically the signal `y` is accompanied by the *sampling rate* (denoted `sr`) which denotes the frequency (in Hz) at which values of `y` are sampled. The duration of a signal can then be computed by dividing the number of samples by the sampling rate:

```
>>> duration_seconds = float(len(y)) / sr
```

By default, when loading stereo audio files, the `librosa.load()` function downmixes to mono by averaging left- and right-channels, and then resamples the monophonic signal to the default rate `sr=22050` Hz.

Most audio analysis methods operate not at the native sampling rate of the signal, but over small *frames* of the signal which are spaced by a *hop length* (in samples). The default frame and hop lengths are set to 2048 and 512 samples, respectively. At the default sampling rate of 22050 Hz, this corresponds to overlapping frames of approximately 93ms spaced by 23ms. Frames are centered by default, so frame index `t` corresponds to the slice:

```
y[(t * hop_length - frame_length / 2):
  (t * hop_length + frame_length / 2)],
```

where boundary conditions are handled by reflection-padding the input signal `y`. Unless otherwise specified, all sliding-window analyses use Hann windows by default. For analyses that do not use fixed-width frames (such as the constant-Q transform), the default hop length of 512 is retained to facilitate alignment of results.

The majority of feature analyses implemented by librosa produce two-dimensional outputs stored as `numpy.ndarray`, e.g., `S[f, t]` might contain the energy within a particular frequency band `f` at frame index `t`. We follow the convention that the final dimension provides the index over time, e.g., `S[:, 0]`, `S[:, 1]` access features at the first and second frames. Feature arrays are organized column-major (Fortran style) in memory, so that common access patterns benefit from cache locality.

By default, all pitch-based analyses are assumed to be relative to a 12-bin equal-tempered chromatic scale with a reference tuning of `A440 = 440.0 Hz`. Pitch and pitch-class analyses are arranged such that the 0th bin corresponds to `C` for pitch class or `C1` (32.7 Hz) for absolute pitch measurements.

5. https://travis-ci.org
6. https://coveralls.io

Package organization

In this section, we give a brief overview of the structure of the librosa software package. This overview is intended to be superficial and cover only the most commonly used functionality. A complete API reference can be found at https://bmcfee.github.io/librosa.

Core functionality

The `librosa.core` submodule includes a range of commonly used functions. Broadly, `core` functionality falls into four categories: audio and time-series operations, spectrogram calculation, time and frequency conversion, and pitch operations. For convenience, all functions within the `core` submodule are aliased at the top level of the package hierarchy, e.g., `librosa.core.load` is aliased to `librosa.load`.

Audio and time-series operations include functions such as: reading audio from disk via the `audioread` package[7] (`core.load`), resampling a signal at a desired rate (`core.resample`), stereo to mono conversion (`core.to_mono`), time-domain bounded auto-correlation (`core.autocorrelate`), and zero-crossing detection (`core.zero_crossings`).

Spectrogram operations include the short-time Fourier transform (`stft`), inverse STFT (`istft`), and instantaneous frequency spectrogram (`ifgram`) [Abe95], which provide much of the core functionality for down-stream feature analysis. Additionally, an efficient constant-Q transform (`cqt`) implementation based upon the recursive down-sampling method of Schoerkhuber and Klapuri [Schoerkhuber10] is provided, which produces logarithmically-spaced frequency representations suitable for pitch-based signal analysis. Finally, `logamplitude` provides a flexible and robust implementation of log-amplitude scaling, which can be used to avoid numerical underflow and set an adaptive noise floor when converting from linear amplitude.

Because data may be represented in a variety of time or frequency units, we provide a comprehensive set of convenience functions to map between different time representations: seconds, frames, or samples; and frequency representations: hertz, constant-Q basis index, Fourier basis index, Mel basis index, MIDI note number, or note in scientific pitch notation.

Finally, the core submodule provides functionality to estimate the dominant frequency of STFT bins via parabolic interpolation (`piptrack`) [Smith11], and estimation of tuning deviation (in cents) from the reference `A440`. These functions allow pitch-based analyses (e.g., `cqt`) to dynamically adapt filter banks to match the global tuning offset of a particular audio signal.

Spectral features

Spectral representations—the distributions of energy over a set of frequencies—form the basis of many analysis techniques in MIR and digital signal processing in general. The `librosa.feature` module implements a variety of spectral representations, most of which are based upon the short-time Fourier transform.

7. https://github.com/sampsyo/audioread

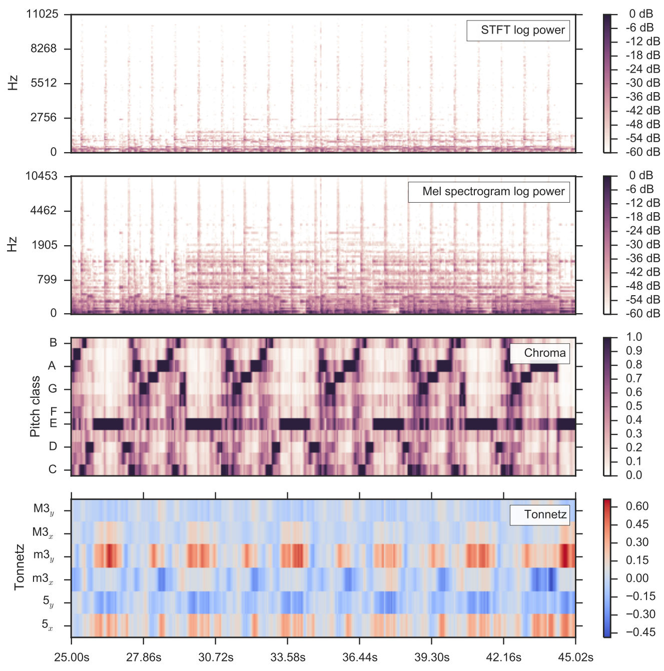


Fig. 1: First: the short-time Fourier transform of a 20-second audio clip (`librosa.stft`). Second: the corresponding Mel spectrogram, using 128 Mel bands (`librosa.feature.melspectrogram`). Third: the corresponding chromagram (`librosa.feature.chroma_cqt`). Fourth: the Tonnetz features (`librosa.feature.tonnetz`).

The Mel frequency scale is commonly used to represent audio signals, as it provides a rough model of human frequency perception [Stevens37]. Both a Mel-scale spectrogram (`librosa.feature.melspectrogram`) and the commonly used Mel-frequency Cepstral Coefficients (MFCC) (`librosa.feature.mfcc`) are provided. By default, Mel scales are defined to match the implementation provided by Slaney's auditory toolbox [Slaney98], but they can be made to match the Hidden Markov Model Toolkit (HTK) by setting the flag `htk=True` [Young97].

While Mel-scaled representations are commonly used to capture timbral aspects of music, they provide poor resolution of pitches and pitch classes. Pitch class (or *chroma*) representations are often used to encode harmony while suppressing variations in octave height, loudness, or timbre. Two flexible chroma implementations are provided: one uses a fixed-window STFT analysis (`chroma_stft`)[8] and the other uses variable-window constant-Q transform analysis (`chroma_cqt`). An alternative representation of pitch and harmony can be obtained by the `tonnetz` function, which estimates tonal centroids as coordinates in a six-dimensional interval space using the method of Harte et al. [Harte06]. Figure 1 illustrates the difference between STFT, Mel spectrogram, chromagram, and Tonnetz representations, as constructed by the following code fragment:[9]

```
>>> filename = librosa.util.example_audio_file()
>>> y, sr = librosa.load(filename,
...                      offset=25.0,
...                      duration=20.0)
>>> spectrogram = np.abs(librosa.stft(y))
>>> melspec = librosa.feature.melspectrogram(y=y,
...                                          sr=sr)
>>> chroma = librosa.feature.chroma_cqt(y=y,
...                                     sr=sr)
>>> tonnetz = librosa.feature.tonnetz(y=y, sr=sr)
```

In addition to Mel and chroma features, the `feature` submodule provides a number of spectral statistic representations, including `spectral_centroid`, `spectral_bandwidth`, `spectral_rolloff` [Klapuri07], and `spectral_contrast` [Jiang02].[10]

Finally, the `feature` submodule provides a few functions to implement common transformations of time-series features in MIR. This includes `delta`, which provides a smoothed estimate of the time derivative; `stack_memory`, which concatenates an input feature array with time-lagged copies of itself (effectively simulating feature n-grams); and `sync`, which applies a user-supplied aggregation function (e.g., `numpy.mean` or `median`) across specified column intervals.

Display

The `display` module provides simple interfaces to visually render audio data through `matplotlib` [Hunter07]. The first function, `display.waveplot` simply renders the amplitude envelope of an audio signal `y` using matplotlib's `fill_between` function. For efficiency purposes, the signal is dynamically down-sampled. Mono signals are rendered symmetrically about the horizontal axis; stereo signals are rendered with the left-channel's amplitude above the axis and the right-channel's below. An example of `waveplot` is depicted in Figure 2 (top).

The second function, `display.specshow` wraps matplotlib's `imshow` function with default settings (`origin` and `aspect`) adapted to the expected defaults for visualizing spectrograms. Additionally, `specshow` dynamically selects appropriate colormaps (binary, sequential, or diverging) from the data type and range.[11] Finally, `specshow` provides a variety of acoustically relevant axis labeling and scaling parameters. Examples of `specshow` output are displayed in Figures 1 and 2 (middle).

Onsets, tempo, and beats

While the spectral feature representations described above capture frequency information, time information is equally important for many applications in MIR. For instance, it can be beneficial to analyze signals indexed by note or beat events, rather than absolute time. The `onset` and `beat` submodules implement functions to estimate various aspects of timing in music.

8. `chroma_stft` is based upon the reference implementation provided at http://labrosa.ee.columbia.edu/matlab/chroma-ansyn/

9. For display purposes, spectrograms are scaled by `librosa.logamplitude`. We refer readers to the accompanying IPython notebook for the full source code to recontsruct figures.

10. `spectral_*` functions are derived from MATLAB reference implementations provided by the METLab at Drexel University. http://music.ece.drexel.edu/

11. If the `seaborn` package [Waskom14] is available, its version of *cubehelix* is used for sequential data.

More specifically, the `onset` module provides two functions: `onset_strength` and `onset_detect`. The `onset_strength` function calculates a thresholded spectral flux operation over a spectrogram, and returns a one-dimensional array representing the amount of increasing spectral energy at each frame. This is illustrated as the blue curve in the bottom panel of Figure 2. The `onset_detect` function, on the other hand, selects peak positions from the onset strength curve following the heuristic described by Boeck et al. [Boeck12]. The output of `onset_detect` is depicted as red circles in the bottom panel of Figure 2.

The `beat` module provides functions to estimate the global tempo and positions of beat events from the onset strength function, using the method of Ellis [Ellis07]. More specifically, the beat tracker first estimates the tempo, which is then used to set the target spacing between peaks in an onset strength function. The output of the beat tracker is displayed as the dashed green lines in Figure 2 (bottom).

Tying this all together, the tempo and beat positions for an input signal can be easily calculated by the following code fragment:

```
>>> y, sr = librosa.load(FILENAME)
>>> tempo, frames = librosa.beat.beat_track(y=y,
...                                         sr=sr)
>>> beat_times = librosa.frames_to_time(frames,
...                                     sr=sr)
```

Any of the default parameters and analyses may be overridden. For example, if the user has calculated an onset strength envelope by some other means, it can be provided to the beat tracker as follows:

```
>>> oenv = some_other_onset_function(y, sr)
>>> librosa.beat.beat_track(onset_envelope=oenv)
```

All detection functions (beat and onset) return events as frame indices, rather than absolute timing. The downside of this is that it is left to the user to convert frame indices back to absolute time. However, in our opinion, this is outweighed by two practical benefits: it simplifies the implementations, and it makes the results directly accessible to frame-indexed functions such as `librosa.feature.sync`.

Structural analysis

Onsets and beats provide relatively low-level timing cues for music signal processing. Higher-level analyses attempt to detect larger structure in music, e.g., at the level of bars or functional components such as *verse* and *chorus*. While this is an active area of research that has seen rapid progress in recent years, there are some useful features common to many approaches. The `segment` submodule contains a few useful functions to facilitate structural analysis in music, falling broadly into two categories.

First, there are functions to calculate and manipulate *recurrence* or *self-similarity* plots. The `segment.recurrence_matrix` constructs a binary *k*-nearest-neighbor similarity matrix from a given feature array and a user-specified distance function. As displayed in Figure 3 (left), repeating sequences often appear as diagonal bands in the recurrence plot, which can be used to detect musical structure. It is sometimes more convenient to operate in *time-lag* coordinates, rather than *time-time*, which transforms diagonal structures into more easily detectable horizontal structures (Figure 3, right) [Serra12]. This is facilitated by the `recurrence_to_lag` (and `lag_to_recurrence`) functions.

Second, temporally constrained clustering can be used to detect feature change-points without relying upon repetition. This is implemented in librosa by the `segment.agglomerative` function, which uses `scikit-learn`'s implementation of Ward's agglomerative clustering method [Ward63] to partition the input into a user-defined number of contiguous components. In practice, a user can override the default clustering parameters by providing an existing `sklearn.cluster.AgglomerativeClustering` object as an argument to `segment.agglomerative()`.

Decompositions

Many applications in MIR operate upon latent factor representations, or other decompositions of spectrograms. For example, it is common to apply non-negative matrix factorization (NMF) [Lee99] to magnitude spectra, and analyze the statistics of the resulting time-varying activation functions, rather than the raw observations.

The `decompose` module provides a simple interface to factor spectrograms (or general feature arrays) into *components* and *activations*:

```
>>> comps, acts = librosa.decompose.decompose(S)
```

By default, the `decompose()` function constructs a `scikit-learn` NMF object, and applies its `fit_transform()` method to the transpose of `S`. The resulting basis components and activations are accordingly transposed, so that `comps.dot(acts)` approximates `S`. If the user wishes to apply some other decomposition technique, any object fitting the `sklearn.decomposition` interface may be substituted:

```
>>> T = SomeDecomposer()
>>> librosa.decompose.decompose(S, transformer=T)
```

In addition to general-purpose matrix decomposition techniques, librosa also implements the harmonic-percussive source separation (HPSS) method of Fitzgerald [Fitzgerald10] as `decompose.hpss`. This technique is commonly used in MIR to suppress transients when analyzing pitch content, or suppress stationary signals when detecting onsets or other rhythmic elements. An example application of HPSS is illustrated in Figure 4.

Effects

The `effects` module provides convenience functions for applying spectrogram-based transformations to time-domain signals. For instance, rather than writing

```
>>> D = librosa.stft(y)
>>> Dh, Dp = librosa.decompose.hpss(D)
>>> y_harmonic = librosa.istft(Dh)
```

one may simply write

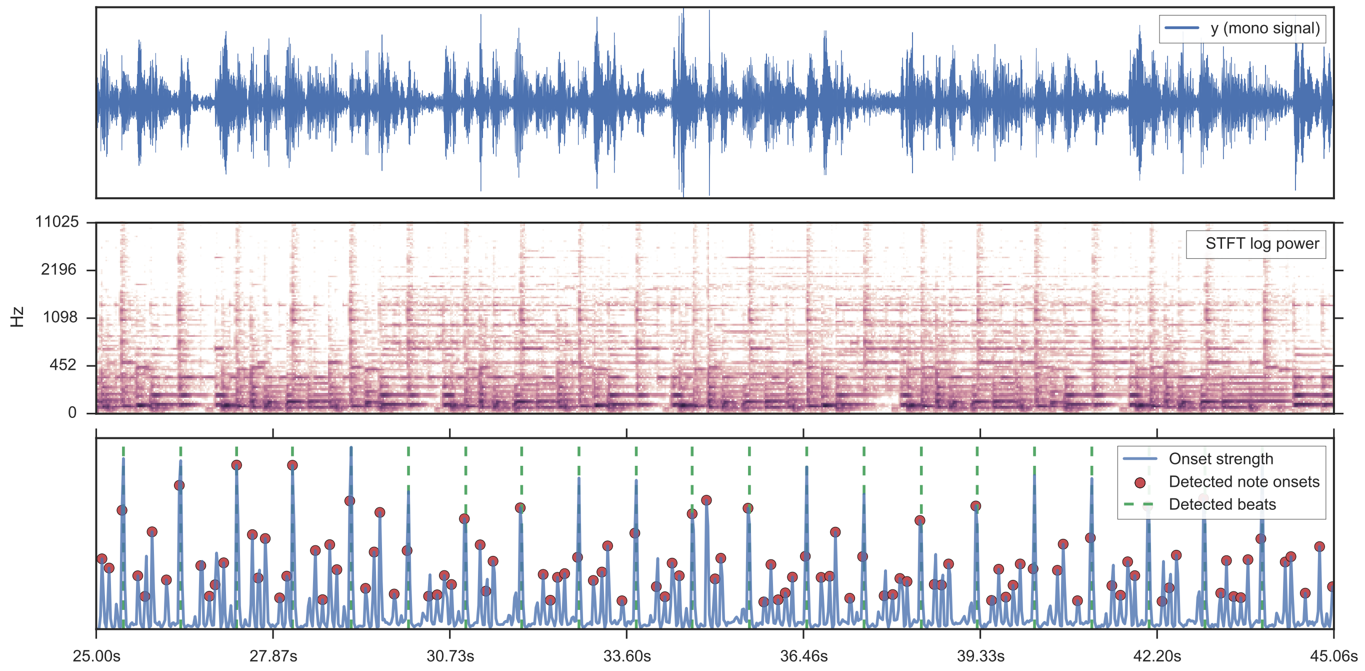


Fig. 2: *Top: a waveform plot for a 20-second audio clip* `y`*, generated by* `librosa.display.waveplot`*. Middle: the log-power short-time Fourier transform (STFT) spectrum for* `y` *plotted on a logarithmic frequency scale, generated by* `librosa.display.specshow`*. Bottom: the onset strength function (*`librosa.onset.onset_strength`*), detected onset events (*`librosa.onset.onset_detect`*), and detected beat events (*`librosa.beat.beat_track`*) for* `y`*.*

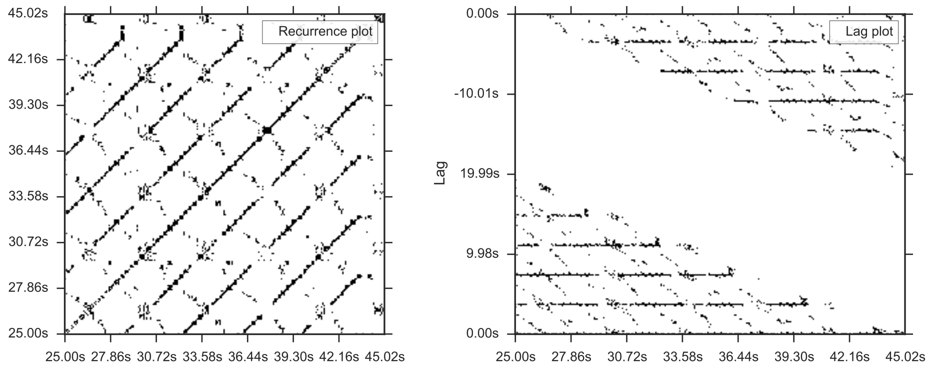


Fig. 3: *Left: the recurrence plot derived from the chroma features displayed in Figure 1. Right: the corresponding time-lag plot.*

```
>>> y_harmonic = librosa.effects.harmonic(y)
```

Convenience functions are provided for HPSS (retaining the harmonic, percussive, or both components), time-stretching and pitch-shifting. Although these functions provide no additional functionality, their inclusion results in simpler, more readable application code.

Output

The `output` module includes utility functions to save the results of audio analysis to disk. Most often, this takes the form of annotated instantaneous event timings or time intervals, which are saved in plain text (comma- or tab-separated values) via `output.times_csv` and `output.annotation`, respectively. These functions are somewhat redundant with alternative functions for text output (e.g., `numpy.savetxt`), but provide sanity checks for length agreement and semantic validation of time intervals. The resulting outputs are designed to work with other common MIR tools, such as `mir_eval` [Raffel14] and `sonic-visualiser` [Cannam10].

The `output` module also provides the `write_wav` function for saving audio in `.wav` format. The `write_wav` simply wraps the built-in `scipy` wav-file writer (`scipy.io.wavfile.write`) with validation and optional normalization, thus ensuring that the resulting audio files are well-formed.

Caching

MIR applications typically require computing a variety of features (e.g., MFCCs, chroma, beat timings, etc) from each audio signal in a collection. Assuming the application programmer is content with default parameters, the simplest way to achieve this is to call each function using audio time-series input, e.g.:

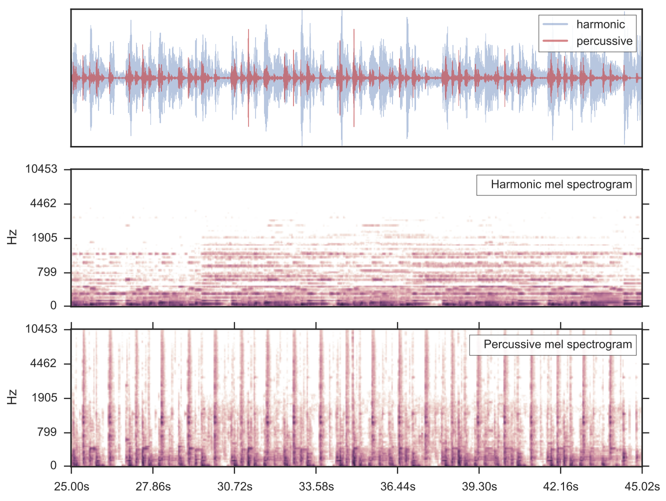


Fig. 4: *Top: the separated harmonic and percussive waveforms. Middle: the Mel spectrogram of the harmonic component. Bottom: the Mel spectrogram of the percussive component.*

```
>>> mfcc = librosa.feature.mfcc(y=y, sr=sr)
>>> tempo, beats = librosa.beat.beat_track(y=y,
...                                         sr=sr)
```

However, because there are shared computations between the different functions—`mfcc` and `beat_track` both compute log-scaled Mel spectrograms, for example—this results in redundant (and inefficient) computation. A more efficient implementation of the above example would factor out the redundant features:

```
>>> lms = librosa.logamplitude(
...     librosa.feature.melspectrogram(y=y,
...                                     sr=sr))
>>> mfcc = librosa.feature.mfcc(S=lms)
>>> tempo, beats = librosa.beat.beat_track(S=lms,
...                                         sr=sr)
```

Although it is more computationally efficient, the above example is less concise, and it requires more knowledge of the implementations on behalf of the application programmer. More generally, nearly all functions in librosa eventually depend upon STFT calculation, but it is rare that the application programmer will need the STFT matrix as an end-result.

One approach to eliminate redundant computation is to decompose the various functions into blocks which can be arranged in a computation graph, as is done in Essentia [Bogdanov13]. However, this approach necessarily constrains the function interfaces, and may become unwieldy for common, simple applications.

Instead, librosa takes a lazy approach to eliminating redundancy via *output caching*. Caching is implemented through an extension of the `Memory` class from the `joblib` package[12], which provides disk-backed memoization of function outputs. The cache object (`librosa.cache`) operates as a decorator on all non-trivial computations. This way, a user can write simple application code (i.e., the first example above) while transparently eliminating redundancies and achieving speed comparable to the more advanced implementation (the second example).

Parameter	Description	Values
`fmax`	Maximum frequency value (Hz)	8000, **11025**
`n_mels`	Number of Mel bands	32, 64, **128**
`aggregate`	Spectral flux aggregation function	`np.mean`, **np.median**
`ac_size`	Maximum lag for onset autocorrelation (s)	2, **4**, 8
`std_bpm`	Deviation of tempo estimates from 120.0 BPM	0.5, **1.0**, 2.0
`tightness`	Penalty for deviation from estimated tempo	50, **100**, 400

TABLE 1: *The parameter grid for beat tracking optimization. The best configuration is indicated in bold.*

The cache object is disabled by default, but can be activated by setting the environment variable `LIBROSA_CACHE_DIR` prior to importing the package. Because the `Memory` object does not implement a cache eviction policy (as of version 0.8.4), it is recommended that users purge the cache after processing each audio file to prevent the cache from filling all available disk space[13]. We note that this can potentially introduce race conditions in multi-processing environments (i.e., parallel batch processing of a corpus), so care must be taken when scheduling cache purges.

Parameter tuning

Some of librosa's functions have parameters that require some degree of tuning to optimize performance. In particular, the performance of the beat tracker and onset detection functions can vary substantially with small changes in certain key parameters.

After standardizing certain default parameters—sampling rate, frame length, and hop length—across all functions, we optimized the beat tracker settings using the parameter grid given in Table 1. To select the best-performing configuration, we evaluated the performance on a data set comprised of the Isophonics Beatles corpus[14] and the SMC Dataset2 [Holzapfel12] beat annotations. Each configuration was evaluated using `mir_eval` [Raffel14], and the configuration was chosen to maximize the Correct Metric Level (Total) metric [Davies14].

Similarly, the onset detection parameters (listed in Table 2) were selected to optimize the F1-score on the Johannes Kepler University onset database.[15]

We note that the "optimal" default parameter settings are merely estimates, and depend upon the datasets over which they are selected. The parameter settings are therefore subject to change in the future as larger reference collections become available. The optimization framework has been factored out into a separate repository, which may in subsequent versions grow to include additional parameters.[16]

12. https://github.com/joblib/joblib
13. The cache can be purged by calling `librosa.cache.clear()`.
14. http://isophonics.net/content/reference-annotations
15. https://github.com/CPJKU/onset_db
16. https://github.com/bmcfee/librosa_parameters

Parameter	Description	Values
`fmax`	Maximum frequency value (Hz)	8000, **11025**
`n_mels`	Number of Mel bands	32, 64, **128**
`aggregate`	Spectral flux aggregation function	**np.mean**, `np.median`
`delta`	Peak picking threshold	0.0--0.10 (**0.07**)

TABLE 2: *The parameter grid for onest detection optimization. The best configuration is indicated in bold.*

Conclusion

This document provides a brief summary of the design considerations and functionality of librosa. More detailed examples, notebooks, and documentation can be found in our development repository and project website. The project is under active development, and our roadmap for future work includes efficiency improvements and enhanced functionality of audio coding and file system interactions.

Citing librosa

We request that when using librosa in academic work, authors cite the Zenodo reference [McFee15]. For references to the *design* of the library, citation of the present document is appropriate.

Acknowledgements

BM acknowledges support from the Moore-Sloan Data Science Environment at NYU. Additional support was provided by NSF grant IIS-1117015.

References

[Pedregosa11] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. *Scikit-learn: Machine learning in Python.* The Journal of Machine Learning Research 12 (2011): 2825-2830.

[Bergstra11] Bergstra, James, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins et al. *Theano: Deep learning on gpus with python.* In NIPS 2011, BigLearning Workshop, Granada, Spain. 2011.

[Jones01] Jones, Eric, Travis Oliphant, and Pearu Peterson. *SciPy: Open source scientific tools for Python.* http://www.scipy.org/ (2001).

[VanDerWalt11] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. *The NumPy array: a structure for efficient numerical computation.* Computing in Science & Engineering 13, no. 2 (2011): 22-30.

[Abe95] Abe, Toshihiko, Takao Kobayashi, and Satoshi Imai. *Harmonics tracking and pitch extraction based on instantaneous frequency.* International Conference on Acoustics, Speech, and Signal Processing, ICASSP-95., Vol. 1. IEEE, 1995.

[Schoerkhuber10] Schoerkhuber, Christian, and Anssi Klapuri. *Constant-Q transform toolbox for music processing.* 7th Sound and Music Computing Conference, Barcelona, Spain. 2010.

[Smith11] Smith, J.O. "Sinusoidal Peak Interpolation", in Spectral Audio Signal Processing, https://ccrma.stanford.edu/~jos/sasp/Sinusoidal_Peak_Interpolation.html , online book, 2011 edition, accessed 2015-06-15.

[Stevens37] Stevens, Stanley Smith, John Volkmann, and Edwin B. Newman. *A scale for the measurement of the psychological magnitude pitch.* The Journal of the Acoustical Society of America 8, no. 3 (1937): 185-190.

[Slaney98] Slaney, Malcolm. *Auditory toolbox.* Interval Research Corporation, Tech. Rep 10 (1998): 1998.

[Young97] Young, Steve, Evermann, Gunnar, Gales, Mark, Hain, Thomas, Kershaw, Dan, Liu, Xunying (Andrew), Moore, Gareth, Odell, Julian, Ollason, Dave, Povey, Dan, Valtchev, Valtcho, and Woodland, Phil. *The HTK book.* Vol. 2. Cambridge: Entropic Cambridge Research Laboratory, 1997.

[Harte06] Harte, C., Sandler, M., & Gasser, M. (2006). *Detecting Harmonic Change in Musical Audio.* In Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia (pp. 21-26). Santa Barbara, CA, USA: ACM Press. doi:10.1145/1178723.1178727.

[Jiang02] Jiang, Dan-Ning, Lie Lu, Hong-Jiang Zhang, Jian-Hua Tao, and Lian-Hong Cai. *Music type classification by spectral contrast feature.* In ICME'02. vol. 1, pp. 113-116. IEEE, 2002.

[Klapuri07] Klapuri, Anssi, and Manuel Davy, eds. *Signal processing methods for music transcription.* Springer Science & Business Media, 2007.

[Hunter07] Hunter, John D. *Matplotlib: A 2D graphics environment.* Computing in science and engineering 9, no. 3 (2007): 90-95.

[Waskom14] Michael Waskom, Olga Botvinnik, Paul Hobson, John B. Cole, Yaroslav Halchenko, Stephan Hoyer, Alistair Miles, et al. *Seaborn: v0.5.0 (November 2014).* ZENODO, 2014. doi:10.5281/zenodo.12710.

[Boeck12] Böck, Sebastian, Florian Krebs, and Markus Schedl. *Evaluating the Online Capabilities of Onset Detection Methods.* In 11th International Society for Music Information Retrieval Conference (ISMIR 2012), pp. 49-54. 2012.

[Ellis07] Ellis, Daniel P.W. *Beat tracking by dynamic programming.* Journal of New Music Research 36, no. 1 (2007): 51-60.

[Serra12] Serra, Joan, Meinard Müller, Peter Grosche, and Josep Lluis Arcos. *Unsupervised detection of music boundaries by time series structure features.* In Twenty-Sixth AAAI Conference on Artificial Intelligence. 2012.

[Ward63] Ward Jr, Joe H. *Hierarchical grouping to optimize an objective function.* Journal of the American statistical association 58, no. 301 (1963): 236-244.

[Lee99] Lee, Daniel D., and H. Sebastian Seung. *Learning the parts of objects by non-negative matrix factorization.* Nature 401, no. 6755 (1999): 788-791.

[Fitzgerald10] Fitzgerald, Derry. *Harmonic/percussive separation using median filtering.* 13th International Conference on Digital Audio Effects (DAFX10), Graz, Austria, 2010.

[Cannam10] Cannam, Chris, Christian Landone, and Mark Sandler. *Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files.* In Proceedings of the international conference on Multimedia, pp. 1467-1468. ACM, 2010.

[Holzapfel12] Holzapfel, Andre, Matthew E.P. Davies, José R. Zapata, João Lobato Oliveira, and Fabien Gouyon. *Selective sampling for beat tracking evaluation.* Audio, Speech, and Language Processing, IEEE Transactions on 20, no. 9 (2012): 2539-2548.

[Davies14] Davies, Matthew E.P., and Boeck, Sebastian. *Evaluating the evaluation measures for beat tracking.* In 15th International Society for Music Information Retrieval Conference (ISMIR 2014), 2014.

[Raffel14] Raffel, Colin, Brian McFee, Eric J. Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel PW Ellis. *mir eval: A transparent implementation of common MIR metrics.* In 15th International Society for Music Information Retrieval Conference (ISMIR 2014), pp. 367-372. 2014.

[Bogdanov13] Bogdanov, Dmitry, Nicolas Wack, Emilia Gómez, Sankalp Gulati, Perfecto Herrera, Oscar Mayor, Gerard Roma, Justin Salamon, José R. Zapata, and Xavier Serra. *Essentia: An Audio Analysis Library for Music Information Retrieval.* In 12th International Society for Music Information Retrieval Conference (ISMIR 2013), pp. 493-498. 2013.

[McFee15] Brian McFee, Matt McVicar, Colin Raffel, Dawen Liang, Oriol Nieto, Josh Moore, Dan Ellis, et al. *Librosa: v0.4.0.* Zenodo, 2015. doi:10.5281/zenodo.18369.