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.

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

* *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.*

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

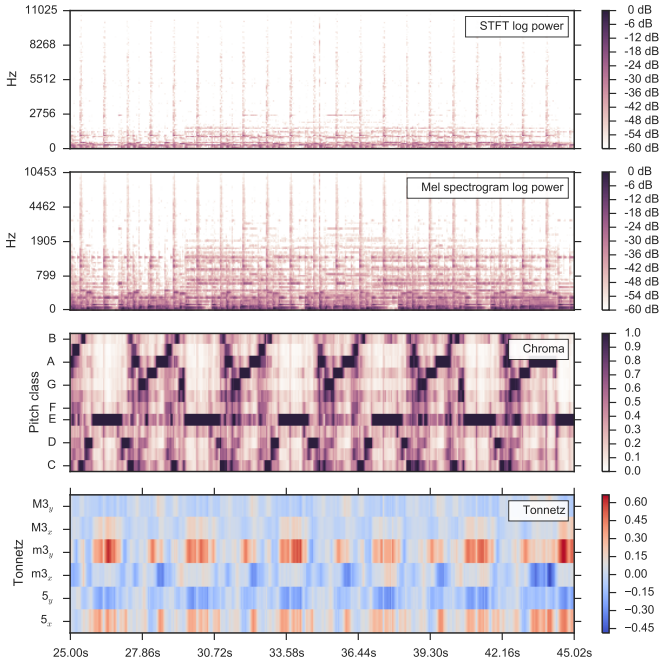

Fig. 1: *First: the short-time Fourier transform of a 20-second audio clip (`librosa.stft`). Second: the corresponding Mel spectrogram, using 128 Mel bands (`librosa.feature.melspectrogram`). Third: the corresponding chromagram (`librosa.feature.chroma_cqt`). Fourth: the Tonnetz features (`librosa.feature.tonnetz`).*

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

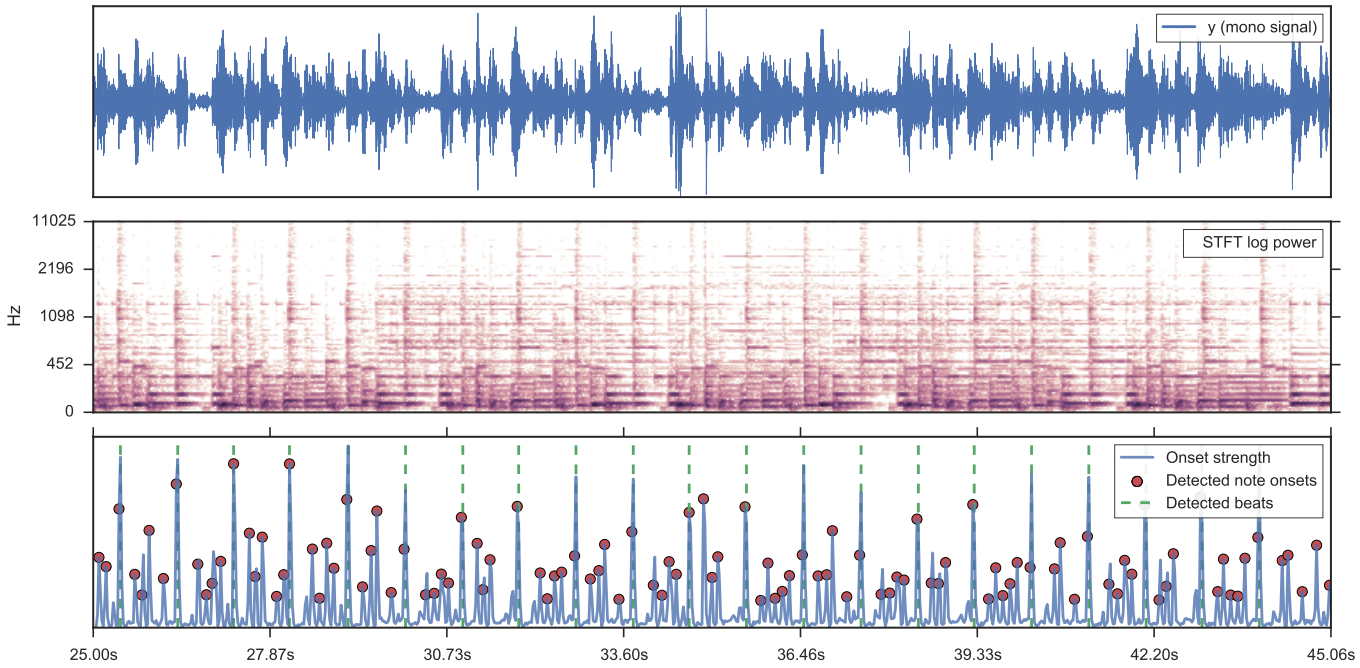

Fig. 2: *Top: a waveform plot for a 20-second audio clip* `y`*, generated by* `librosa.display.waveplot`*. Middle: the log-power short-time Fourier transform (STFT) spectrum for* `y` *plotted on a logarithmic frequency scale, generated by* `librosa.display.specshow`*. Bottom: the onset strength function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

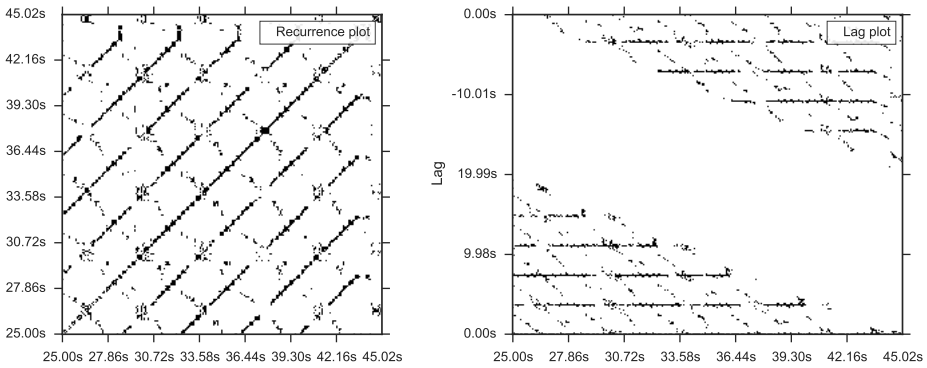

Fig. 3: *Left: the recurrence plot derived from the chroma features displayed in Figure 1. Right: the corresponding time-lag plot.*

```
>>> 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.:

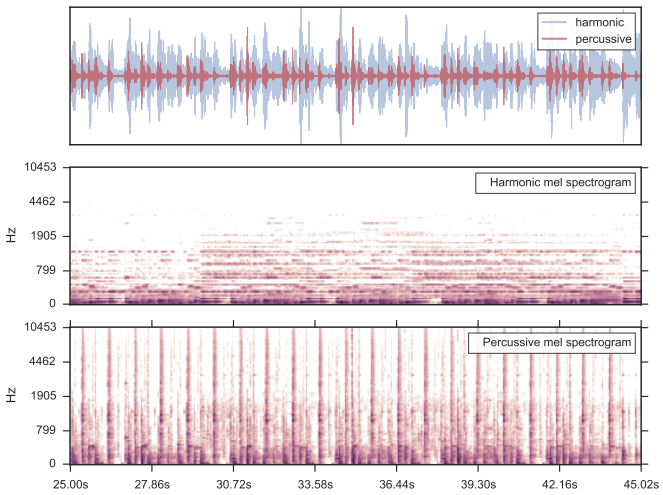

Fig. 4: *Top: the separated harmonic and percussive waveforms. Middle: the Mel spectrogram of the harmonic component. Bottom: the Mel spectrogram of the percussive component.*

```
>>> 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.

Python For Audio Signal Processing

John GLOVER, Victor LAZZARINI and **Joseph TIMONEY**
The Sound and Digital Music Research Group
National University of Ireland, Maynooth
Ireland
{John.C.Glover, Victor.Lazzarini}@nuim.ie
JTimoney@cs.nuim.ie

Abstract

This paper discusses the use of Python for developing audio signal processing applications. Overviews of Python language, NumPy, SciPy and Matplotlib are given, which together form a powerful platform for scientific computing. We then show how SciPy was used to create two audio programming libraries, and describe ways that Python can be integrated with the SndObj library and Pure Data, two existing environments for music composition and signal processing.

Keywords

Audio, Music, Signal Processing, Python, Programming

1 Introduction

There are many problems that are common to a wide variety of applications in the field of audio signal processing. Examples include procedures such as loading sound files or communicating between audio processes and sound cards, as well as digital signal processing (DSP) tasks such as filtering and Fourier analysis [Allen and Rabiner, 1977]. It often makes sense to rely on existing code libraries and frameworks to perform these tasks. This is particularly true in the case of building prototypes, a practise common to both consumer application developers and scientific researchers, as these code libraries allows the developer to focus on the novel aspects of their work.

Audio signal processing libraries are available for general purpose programming languages such as the GNU Scientific Library (GSL) for C/C++ [Galassi et al., 2009], which provides a comprehensive array of signal processing tools. However, it generally takes a lot more time to develop applications or prototypes in C/C++ than in a more lightweight scripting language. This is one of the reasons for the popularity of tools such as MATLAB [MathWorks, 2010], which allow the developer to easily manipulate matrices of numerical data, and includes implementations of many standard signal processing techniques. The major downside to MATLAB is that it is not free and not open source, which is a considerable problem for researchers who want to share code and collaborate. GNU Octave [Eaton, 2002] is an open source alternative to MATLAB. It is an interpreted language with a syntax that is very similar to MATLAB, and it is possible to write scripts that will run on both systems. However, with both MATLAB and Octave this increase in short-term productivity comes at a cost. For anything other than very basic tasks, tools such as integrated development environments (IDEs), debuggers and profilers are certainly a useful resource if not a requirement. All of these tools exist in some form for MATLAB/Octave, but users must invest a considerable amount of time in learning to use a programming language and a set of development tools that have a relatively limited application domain when compared with general purpose programming languages. It is also generally more difficult to integrate MATLAB/Octave programs with compositional tools such as Csound [Vercoe et al., 2011] or Pure Data [Puckette, 1996], or with other technologies such as web frameworks, cloud computing platforms and mobile applications, all of which are becoming increasingly important in the music industry.

For developing and prototyping audio signal processing applications, it would therefore be advantageous to combine the power and flexibility of a widely adopted, open source, general purpose programming language with the quick development process that is possible when using interpreted languages that are focused on signal processing applications. Python [van Rossum and Drake, 2006], when used in conjunction with the extension modules NumPy [Oliphant, 2006], SciPy [Jones et al., 2001] and Matplotlib [Hunter, 2007] has all of these characteristics.

Section 2 provides a brief overview of the Python programming language. In Section 3 we discuss NumPy, SciPy and Matplotlib, which add a rich set of scientific computing functions to the Python language. Section 4 describes two libraries created by the authors that rely on SciPy, Section 5 shows how these Python programs can be integrated with other software tools for music composition, with final conclusions given in Section 6.

2 Python

Python is an open source programming language that runs on many platforms including Linux, Mac OS X and Windows. It is widely used and actively developed, has a vast array of code libraries and development tools, and integrates well with many other programming languages, frameworks and musical applications. Some notable features of the language include:

- It is a mature language and allows for programming in several different paradigms including imperative, object-orientated and functional styles.
- The clean syntax puts an emphasis on producing well structured and readable code. Python source code has often been compared to executable pseudocode.
- Python provides an interactive interpreter, which allows for rapid code development, prototyping and live experimentation.
- The ability to extend Python with modules written in C/C++ means that functionality can be quickly prototyped and then optimised later.
- Python can be embedded into existing applications.
- Documentation can be generated automatically from the comments and source code.
- Python bindings exist for cross-platform GUI toolkits such as Qt [Nokia, 2011].
- The large number of high-quality library modules means that you can quickly build sophisticated programs.

A complete guide to the language, including a comprehensive tutorial is available online at http://python.org.

3 Python for Scientific Computing

Section 3.1 provides an overview of three packages that are widely used for performing efficient numerical calculations and data visualisation using Python. Example programs that make use of these packages are given in Section 3.2.

3.1 NumPy, SciPy and Matplotlib

Python's scientific computing prowess comes largely from the combination of three related extension modules: NumPy, SciPy and Matplotlib. NumPy [Oliphant, 2006] adds a homogenous, multidimensional array object to Python. It also provides functions that perform efficient calculations based on array data. NumPy is written in C, and can be extended easily via its own C-API. As many existing scientific computing libraries are written in Fortran, NumPy comes with a tool called f2py which can parse Fortran files and create a Python extension module that contains all the subroutines and functions in those files as callable Python methods.

SciPy builds on top of NumPy, providing modules that are dedicated to common issues in scientific computing, and so it can be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. A complete list of SciPy modules is available online at http://docs.scipy.org, but examples include:

File input/output (scipy.io): Provides functions for reading and writing files in many different data formats, including *.wav*, *.csv* and matlab data files (*.mat*).

Fourier transforms (scipy.fftpack): Contains implementations of 1-D and 2-D fast Fourier transforms, as well as Hilbert and inverse Hilbert transforms.

Signal processing (scipy.signal): Provides implementations of many useful signal processing techniques, such as waveform generation, FIR and IIR filtering and multi-dimensional convolution.

Interpolation (scipy.interpolate): Consists of linear interpolation functions and cubic splines in several dimensions.

Matplotlib is a library of 2-dimensional plotting functions that provides the ability to quickly visualise data from NumPy arrays, and produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, providing similar functionality to MATLAB or GNU Plot [Williams et al., 2011]. It can also be used in Python scripts, web applications servers or in combination with several GUI toolkits.

3.2 SciPy Examples

Listing 1 shows how SciPy can be used to read in the samples from a flute recording stored in a file called *flute.wav*, and then plot them using Matplotlib. The call to the *read* function on line 5 returns a tuple containing the sampling rate of the audio file as the first entry and the audio samples as the second entry. The samples are stored in a variable called audio, with the first 1024 samples being plotted in line 8. In lines 10, 11 and 13 the axis labels and the plot title are set, and finally the plot is displayed in line 15. The image produced by Listing 1 is shown in Figure 1.

```
1 from scipy.io.wavfile import read
2 import matplotlib.pyplot as plt
3
4 # read audio samples
5 input_data = read("flute.wav")
6 audio = input_data[1]
7 # plot the first 1024 samples
8 plt.plot(audio[0:1024])
9 # label the axes
10 plt.ylabel("Amplitude")
11 plt.xlabel("Time (samples)")
12 # set the title
13 plt.title("Flute Sample")
14 # display the plot
15 plt.show()
```

Listing 1: Plotting Audio Files

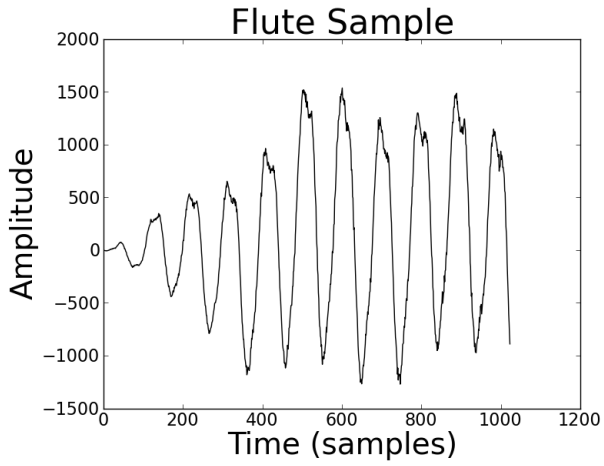

Figure 1: Plot of audio samples, generated by the code given in Listing 1.

In Listing 2, SciPy is used to perform a Fast Fourier Transform (FFT) on a windowed frame of audio samples then plot the resulting magnitude spectrum. In line 11, the SciPy *hann* function is used to compute a 1024 point Hanning window, which is then applied to the first 1024 flute samples in line 12. The FFT is computed in line 14, with the complex coefficients converted into polar form and the magnitude values stored in the variable *mags*. The magnitude values are converted from a linear to a decibel scale in line 16, then normalised to have a maximum value of 0 dB in line 18. In lines 20-26 the magnitude values are plotted and displayed. The resulting image is shown in Figure 2.

```
1 import scipy
2 from scipy.io.wavfile import read
3 from scipy.signal import hann
4 from scipy.fftpack import rfft
5 import matplotlib.pyplot as plt
6
7 # read audio samples
8 input_data = read("flute.wav")
9 audio = input_data[1]
10 # apply a Hanning window
11 window = hann(1024)
12 audio = audio[0:1024] * window
13 # fft
14 mags = abs(rfft(audio))
15 # convert to dB
16 mags = 20 * scipy.log10(mags)
17 # normalise to 0 dB max
18 mags -= max(mags)
19 # plot
20 plt.plot(mags)
21 # label the axes
22 plt.ylabel("Magnitude (dB)")
23 plt.xlabel("Frequency Bin")
24 # set the title
25 plt.title("Flute Spectrum")
26 plt.show()
```

Listing 2: Plotting a magnitude spectrum

4 Audio Signal Processing With Python

This section gives an overview of how SciPy is used in two software libraries that were created by the authors. Section 4.1 gives an overview of Simpl [Glover et al., 2009], while Section 4.2 introduces Modal, our new library for musical note onset detection.

4.1 Simpl

Simpl [1] is an open source library for sinusoidal modelling [Amatriain et al., 2002] written in C/C++ and Python. The aim of this project is

[1] Available at http://simplsound.sourceforge.net

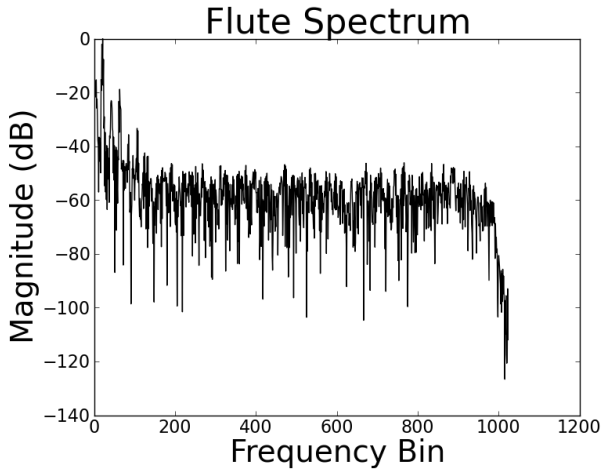

Figure 2: Flute magnitude spectrum produced from code in Listing 2.

to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API, as well as provide implementations of some recently published sinusoidal modelling algorithms. Simpl is primarily intended as a tool for other researchers in the field, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms.

Simpl breaks the sinusoidal modelling process down into three distinct steps: peak detection, partial tracking and sound synthesis. The supported sinusoidal modelling implementations have a Python module associated with every step which returns data in the same format, irrespective of its underlying implementation. This allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without effecting the rest of the network. Each object has a method for real-time interaction as well as non-real-time or batch mode processing, as long as these modes are supported by the underlying algorithm.

All audio in Simpl is stored in NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data. Audio samples are passed into a *PeakDetection* object for analysis, with detected peaks being returned as NumPy arrays that are used to build a list of *Peak* objects. Peaks are then passed to *PartialTracking* objects, which return partials that can be transferred to *Synthesis* objects to create a NumPy array of synthesised audio samples. Simpl also includes a module with plotting functions that use Matplotlib to plot analysis data from the peak detection and partial tracking analysis phases.

An example Python program that uses Simpl is given in Listing 3. Lines 6-8 read in the first 4096 sample values of a recorded flute note. As the default hop size is 512 samples, this will produce 8 frames of analysis data. In line 10 a *SndObjPeakDetection* object is created, which detects sinusoidal peaks in each frame of audio using the algorithm from The SndObj Library [Lazzarini, 2001]. The maximum number of detected peaks per frame is limited to 20 in line 11, before the peaks are detected and returned in line 12. In line 15 a *MQPartialTracking* object is created, which links previously detected sinusoidal peaks together to form partials, using the McAulay-Quatieri algorithm [McAulay and Quatieri, 1986]. The maximum number of partials is limited to 20 in line 16 and the partials are detected and returned in line 17. Lines 18-25 plot the partials, set the figure title, label the axes and display the final plot as shown in Figure 3.

```
1 import simpl
2 import matplotlib.pyplot as plt
3 from scipy.io.wavfile import read
4
5 # read audio samples
6 audio = read("flute.wav")[1]
7 # take just the first few frames
8 audio = audio[0:4096]
9 # Peak detection with SndObj
10 pd = simpl.SndObjPeakDetection()
11 pd.max_peaks = 20
12 pks = pd.find_peaks(audio)
13 # Partial Tracking with
14 # the McAulay-Quatieri algorithm
15 pt = simpl.MQPartialTracking()
16 pt.max_partials = 20
17 partls = pt.find_partials(pks)
18 # plot the detected partials
19 simpl.plot.plot_partials(partls)
20 # set title and label axes
21 plt.title("Flute Partials")
22 plt.ylabel("Frequency (Hz)")
23 plt.xlabel("Frame Number")
24 plt.show()
```

Listing 3: A Simpl example

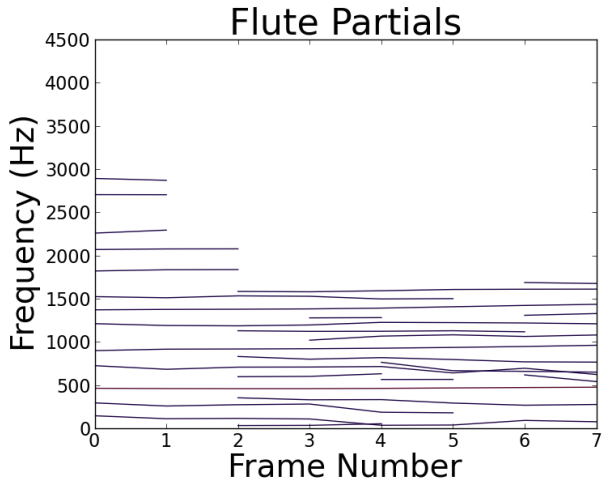

Figure 3: Partials detected in the first 8 frames of a flute sample, produced by the code in Listing 3. Darker colours indicate lower amplitude partials.

4.2 Modal

Modal [2] is a new open source library for musical onset detection, written in C++ and Python and released under the terms of the GNU General Public License (GPL). Modal consists of two main components: a code library and a database of audio samples. The code library includes implementations of three widely used onset detection algorithms from the literature and four novel onset detection systems created by the authors. The onset detection systems can work in a real-time streaming situation as well as in non-real-time. For more information on onset detection in general, a good overview is given in Bello et al. (2005).

The sample database contains a collection of audio samples that have creative commons licensing allowing for free reuse and redistribution, together with hand-annotated onset locations for each sample. It also includes an application that allows for the labelling of onset locations in audio files, which can then be added to the database. To the best of our knowledge, this is the only freely distributable database of audio samples together with their onset locations that is currently available. The Sound Onset Labellizer [Leveau et al., 2004] is a similar reference collection, but was not available at the time of publication. The sample set used by the Sound Onset Labellizer also makes use of files from the RWC database [Goto et al., 2002], which although publicly available is not free and does not allow free redistribution.

[2] Available at http://github.com/johnglover/modal

Modal makes extensive use of SciPy, with NumPy arrays being used to contain audio samples and analysis data from multiple stages of the onset detection process including computed onset detection functions, peak picking thresholds and the detected onset locations, while Matplotlib is used to plot the analysis results. All of the onset detection algorithms were written in Python and make use of SciPy's signal processing modules. The most computationally expensive part of the onset detection process is the calculation of the onset detection functions, so Modal also includes C++ implementations of all onset detection function modules. These are made into Python extension modules using SWIG [Beazley, 2003]. As SWIG extension modules can manipulate NumPy arrays, the C++ implementations can be seamlessly interchanged with their pure Python counterparts. This allows Python to be used in areas that it excels in such as rapid prototyping and in "glueing" related components together, while languages such as C and C++ can be used later in the development cycle to optimise specific modules if necessary.

Listing 4 gives an example that uses Modal, with the resulting plot shown in Figure 4. In line 12 an audio file consisting of a sequence of percussive notes is read in, with the sample values being converted to floating-point values between -1 and 1 in line 14. The onset detection process in Modal consists of two steps, creating a detection function from the source audio and then finding onsets, which are peaks in this detection function that are above a given threshold value. In line 16 a *ComplexODF* object is created, which calculates a detection function based on the complex domain phase and energy approach described by Bello et al. (2004). This detection function is computed and saved in line 17. Line 19 creates an *OnsetDetection* object which finds peaks in the detection function that are above an adaptive median threshold [Brossier et al., 2004]. The onset locations are calculated and saved on lines 21-22. Lines 24-42 plot the results. The figure is divided into 2 subplots, the first (upper) plot shows the original audio file (dark grey) with the detected onset locations (vertical red dashed lines). The second (lower) plot shows the detection function (dark grey) and the adaptive threshold value (green).

```
1 from modal.onsetdetection \
2     import OnsetDetection
3 from modal.detectionfunctions \
```

```
    import ComplexODF
from modal.ui.plot import \
    (plot_detection_function,
     plot_onsets)
import matplotlib.pyplot as plt
from scipy.io.wavfile import read

# read audio file
audio = read("drums.wav")[1]
# values between -1 and 1
audio = audio / 32768.0
# create detection function
codf = ComplexODF()
odf = codf.process(audio)
# create onset detection object
od = OnsetDetection()
hop_size = codf.get_hop_size()
onsets = od.find_onsets(odf) * \
    hop_size
# plot onset detection results
plt.subplot(2,1,1)
plt.title("Audio And Detected "
          "Onsets")
plt.ylabel("Sample Value")
plt.xlabel("Sample Number")
plt.plot(audio, "0.4")
plot_onsets(onsets)
plt.subplot(2,1,2)
plt.title("Detection Function "
          "And Threshold")
plt.ylabel("Detection Function "
           "Value")
plt.xlabel("Sample Number")
plot_detection_function(odf,
                        hop_size)
thresh = od.threshold
plot_detection_function(thresh,
                        hop_size,
                        "green")
plt.show()
```

Listing 4: Modal example

5 Integration With Other Music Applications

This section provides examples of SciPy integration with two established tools for sound design and composition. Section 5.1 shows SciPy integration with The SndObj Library, with Section 5.2 providing an example of using SciPy in conjunction with Pure Data.

5.1 The SndObj Library

The most recent version of The SndObj Library comes with support for passing NumPy arrays

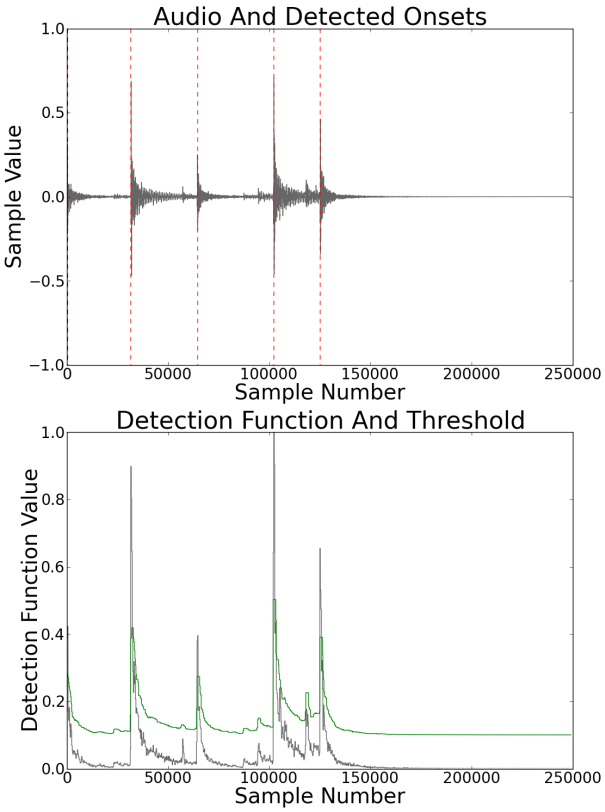

Figure 4: The upper plot shows an audio sample with detected onsets indicated by dashed red lines. The lower plot shows the detection function that was created from the audio file (in grey) and the peak picking threshold (in green).

to and from objects in the library, allowing data to be easily exchanged between SndObj and SciPy audio processing functions. An example of this is shown in Listing 5. An audio file is loaded in line 8, then the *scipy.signal* module is used to low-pass filter it in lines 10-15. The filter cutoff frequency is given as 0.02, with 1.0 being the Nyquist frequency. A *SndObj* called *obj* is created in line 21 that will hold frames of the output audio signal. In lines 24 and 25, a *SndRTIO* object is created and set to write the contents of *obj* to the default sound output. Finally in lines 29-33, each frame of audio is taken, copied into *obj* and then written to the output.

```
from sndobj import \
    SndObj, SndRTIO, SND_OUTPUT
import scipy as sp
from scipy.signal import firwin
from scipy.io.wavfile import read

# read audio file
audio = read("drums.wav")[1]
```

```
# use SciPy to low pass filter
order = 101
cutoff = 0.02
filter = firwin(order, cutoff)
audio = sp.convolve(audio,
                    filter,
                    "same")
# convert to 32-bit floats
audio = sp.asarray(audio,
                   sp.float32)
# create a SndObj that will hold
# frames of output audio
obj = SndObj()
# create a SndObj that will
# output to the sound card
outp = SndRTIO(1, SND_OUTPUT)
outp.SetOutput(1, obj)
# get the default frame size
f_size = outp.GetVectorSize()
# output each frame
i = 0
while i < len(audio):
    obj.PushIn(audio[i:i+f_size])
    outp.Write()
    i += f_size
```

Listing 5: The SndObj Library and SciPy

5.2 Pure Data

The recently released libpd [3] allows Pure Data to be embedded as a DSP library, and comes with a SWIG wrapper enabling it to be loaded as a Python extension module. Listing 6 shows how SciPy can be used in conjunction with libpd to process an audio file and save the result to disk. In lines 7-13 a *PdManager* object is created, that initialises libpd to work with a single channel of audio at a sampling rate of 44.1 KHz. A Pure Data patch is opened in lines 14-16, followed by an audio file being loaded in line 20. In lines 22-29, successive audio frames are processed using the signal chain from the Pure Data patch, with the resulting data converted into an array of integer values and appended to the *out* array. Finally, the patch is closed in line 31 and the processed audio is written to disk in line 33.

```
import scipy as sp
from scipy import int16
from scipy.io.wavfile import \
    read, write
import pylibpd as pd

num_chans = 1
sampling_rate = 44100
# open a Pure Data patch
m = pd.PdManager(num_chans,
                 num_chans,
                 sampling_rate,
                 1)
p_name = "ring_mod.pd"
patch = \
    pd.libpd_open_patch(p_name)
# get the default frame size
f_size = pd.libpd_blocksize()
# read audio file
audio = read("drums.wav")[1]
# process each frame
i = 0
out = sp.array([], dtype=int16)
while i < len(audio):
    f = audio[i:i+f_size]
    p = m.process(f)
    p = sp.fromstring(p, int16)
    out = sp.hstack((out, p))
    i += f_size
# close the patch
pd.libpd_close_patch(patch)
# write the audio file to disk
write("out.wav", 44100, out)
```

Listing 6: Pure Data and SciPy

[3] Available at http://gitorious.org/pdlib/libpd

6 Conclusions

This paper highlighted just a few of the many features that make Python an excellent choice for developing audio signal processing applications. A clean, readable syntax combined with an extensive collection of libraries and an unrestrictive open source license make Python particularly well suited to rapid prototyping and make it an invaluable tool for audio researchers. This was exemplified in the discussion of two open source signal processing libraries created by the authors that both make use of Python and SciPy: Simpl and Modal. Python is easy to extend and integrates well with other programming languages and environments, as demonstrated by the ability to use Python and SciPy in conjunction with established tools for audio signal processing such as The SndObj Library and Pure Data.

7 Acknowledgements

The authors would like to acknowledge the generous support of An Foras Feasa, who funded this research.

References

J.B. Allen and L.R. Rabiner. 1977. A unified approach to short-time Fourier analysis and synthesis. *Proceedings of the IEEE*, 65(11), November.

X. Amatriain, Jordi Bonada, A. Loscos, and Xavier Serra, 2002. *DAFx - Digital Audio Effects*, chapter Spectral Processing, pages 373–438. John Wiley and Sons.

David M. Beazley. 2003. Automated scientific software scripting with SWIG. *Future Generation Computer Systems - Tools for Program Development and Analysis*, 19(5):599–609, July.

Juan Pablo Bello, Chris Duxbury, Mike Davies, and Mark Sandler. 2004. On the use of phase and energy for musical onset detection in the complex domain. *IEEE Signal Processing Letters*, 11(6):553–556, June.

Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark Sandler. 2005. A Tutorial on Onset Detection in Music Signals. *IEEE Transactions on Speech and Audio Processing*, 13(5):1035–1047, September.

Paul Brossier, Juan Pablo Bello, and Mark Plumbley. 2004. Real-time temporal segmentation of note objects in music signals. In *Proceedings of the International Computer Music Conference (ICMC'04)*, pages 458–461.

John W. Eaton. 2002. *GNU Octave Manual*. Network Theory Limited, Bristol, UK.

M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. 2009. *GNU Scientific Library Reference Manual*. Network Theory Limited, Bristol, UK, 3 edition.

John Glover, Victor Lazzarini, and Joseph Timoney. 2009. Simpl: A Python library for sinusoidal modelling. In *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy, September.

Masataka Goto, Hiroki Hashiguchi, Takuichi Nishimura, and Ryuichi Oka. 2002. RWC music database: Popular, classical, and jazz music databases. In *Proceedings of the 3rd International Conference on Music Information Retrieval (ISMIR 2002)*, pages 287–288, October.

John D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001. SciPy: Open source scientific tools for Python. http://www.scipy.org (last accessed 17-02-2011).

Victor Lazzarini. 2001. Sound processing with The SndObj Library: An overview. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, University of Limerick, Ireland, December.

Piere Leveau, Laurent Daudet, and Gael Richard. 2004. Methodology and tools for the evaluation of automatic onset detection algorithms in music. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR)*, Barcelona, Spain, October.

The MathWorks. 2010. *MATLAB Release R2010b*. The MathWorks, Natick, Massachusetts.

Robert McAulay and Thomas Quatieri. 1986. Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-34(4), August.

Nokia. 2011. Qt - a cross-platform application and UI framework. http://qt.nokia.com (last accessed 17-02-2011).

Travis Oliphant. 2006. *Guide To NumPy*. Trelgol Publishing, USA.

Miller Puckette. 1996. Pure Data. In *Proceedings of the International Computer Music Conference (ICMC'96)*, pages 224–227, San Francisco, USA.

Guido van Rossum and Fred L. Drake. 2006. *Python Language Reference Manual*. Network Theory Limited, Bristol, UK.

Barry Vercoe et al. 2011. The Csound Reference Manual. http://www.csounds.com (last accessed 17-02-2011).

Thomas Williams, Colin Kelley, et al. 2011. Gnuplot. http://www.gnuplot.info (last accessed 17-02-2011).