Mel Frequency Cepstral Coefficient (MFCC) tutorial

The first step in any automatic speech recognition system is to extract features i.e. identify the components of the audio signal that are good for identifying the linguistic content and discarding all the other stuff which carries information like background noise, emotion etc.

The main point to understand about speech is that the sounds generated by a human are filtered by the shape of the vocal tract including tongue, teeth etc. This shape determines what sound comes out. If we can determine the shape accurately, this should give us an accurate representation of the phoneme being produced. The shape of the vocal tract manifests itself in the envelope of the short time power spectrum, and the job of MFCCs is to accurately represent this envelope. This page will provide a short tutorial on MFCCs.

Mel Frequency Cepstral Coefficents (MFCCs) are a feature widely used in automatic speech and speaker recognition. They were introduced by Davis and Mermelstein in the 1980's, and have been state-of-the-art ever since. Prior to the introduction of MFCCs, Linear Prediction Coefficients (LPCs) and Linear Prediction Cepstral Coefficients (LPCCs) (click here for a tutorial on cepstrum and LPCCs) and were the main feature type for automatic speech recognition (ASR), especially with HMM classifiers. This page will go over the main aspects of MFCCs, why they make a good feature for ASR, and how to implement them.

Steps at a Glance

We will give a high level intro to the implementation steps, then go in depth why we do the things we do. Towards the end we will go into a more detailed description of how to calculate MFCCs.

1. Frame the signal into short frames.
2. For each frame calculate the periodogram estimate of the power spectrum.
3. Apply the mel filterbank to the power spectra, sum the energy in each filter.
4. Take the logarithm of all filterbank energies.
5. Take the DCT of the log filterbank energies.
6. Keep DCT coefficients 2-13, discard the rest.

There are a few more things commonly done, sometimes the frame energy is appended to each feature vector. Delta and Delta-Delta features are usually also appended. Liftering is also commonly applied to the final features.

Why do we do these things?

We will now go a little more slowly through the steps and explain why each of the steps is necessary.

An audio signal is constantly changing, so to simplify things we assume that on short time scales the audio signal doesn't change much (when we say it doesn't change, we mean statistically i.e. statistically stationary, obviously the samples are constantly changing on even short time scales). This is why we frame the signal into 20-40ms frames. If the frame is much shorter we don't have enough samples to get a reliable spectral estimate, if it is longer the signal changes too much throughout the frame.

The next step is to calculate the power spectrum of each frame. This is motivated by the human cochlea (an organ in the ear) which vibrates at different spots depending on the frequency of the incoming sounds. Depending on the location in the cochlea that vibrates (which wobbles small hairs), different nerves fire informing the brain that certain frequencies are present. Our periodogram estimate performs a similar job for us, identifying which frequencies are present in the frame.

The periodogram spectral estimate still contains a lot of information not required for Automatic Speech Recognition (ASR). In particular the cochlea can not discern the difference between two closely spaced frequencies. This effect becomes more pronounced as the frequencies increase. For this reason we take clumps of periodogram bins and sum them up to get an idea of how much energy exists in various frequency regions. This is performed by our Mel filterbank: the first filter is very narrow and gives an indication of how much energy exists near 0 Hertz. As the frequencies get higher our filters

Contents

Further reading

We recommend these books if you're interested in finding out more.

get wider as we become less concerned about variations. We are only interested in roughly how much energy occurs at each spot. The Mel scale tells us exactly how to space our filterbanks and how wide to make them. See below for how to calculate the spacing.

Once we have the filterbank energies, we take the logarithm of them. This is also motivated by human hearing: we don't hear loudness on a linear scale. Generally to double the percieved volume of a sound we need to put 8 times as much energy into it. This means that large variations in energy may not sound all that different if the sound is loud to begin with. This compression operation makes our features match more closely what humans actually hear. Why the logarithm and not a cube root? The logarithm allows us to use cepstral mean subtraction, which is a channel normalisation technique.

The final step is to compute the DCT of the log filterbank energies. There are 2 main reasons this is performed. Because our filterbanks are all overlapping, the filterbank energies are quite correlated with each other. The DCT decorrelates the energies which means diagonal covariance matrices can be used to model the features in e.g. a HMM classifier. But notice that only 12 of the 26 DCT coefficients are kept. This is because the higher DCT coefficients represent fast changes in the filterbank energies and it turns out that these fast changes actually degrade ASR performance, so we get a small improvement by dropping them.

What is the Mel scale?

The Mel scale relates perceived frequency, or pitch, of a pure tone to its actual measured frequency. Humans are much better at discerning small changes in pitch at low frequencies than they are at high frequencies. Incorporating this scale makes our features match more closely what humans hear.

The formula for converting from frequency to Mel scale is:

$$M(f) = 1125 \ln(1 + f/700) \qquad (1)$$

To go from Mels back to frequency:

$$M^{-1}(m) = 700(\exp(m/1125) - 1) \qquad (2)$$

Implementation steps

We start with a speech signal, we'll assume sampled at 16kHz.

1. Frame the signal into 20-40 ms frames. 25ms is standard. This means the frame length for a 16kHz signal is 0.025*16000 = 400 samples. Frame step is usually something like 10ms (160 samples), which allows some overlap to the frames. The first 400 sample frame starts at sample 0, the next 400 sample frame starts at sample 160 etc. until the end of the speech file is reached. If the speech file does not divide into an even number of frames, pad it with zeros so that it does.

The next steps are applied to every single frame, one set of 12 MFCC coefficients is extracted for each frame. A short aside on notation: we call our time domain signal $s(n)$. Once it is framed we have $s_i(n)$ where n ranges over 1-400 (if our frames are 400 samples) and i ranges over the number of frames. When we calculate the complex DFT, we get $S_i(k)$ - where the i denotes the frame number corresponding to the time-domain frame. $P_i(k)$ is then the power spectrum of frame i.

2. To take the Discrete Fourier Transform of the frame, perform the following:

$$S_i(k) = \sum_{n=1}^{N} s_i(n)h(n)e^{-j2\pi kn/N} \qquad 1 \le k \le K$$

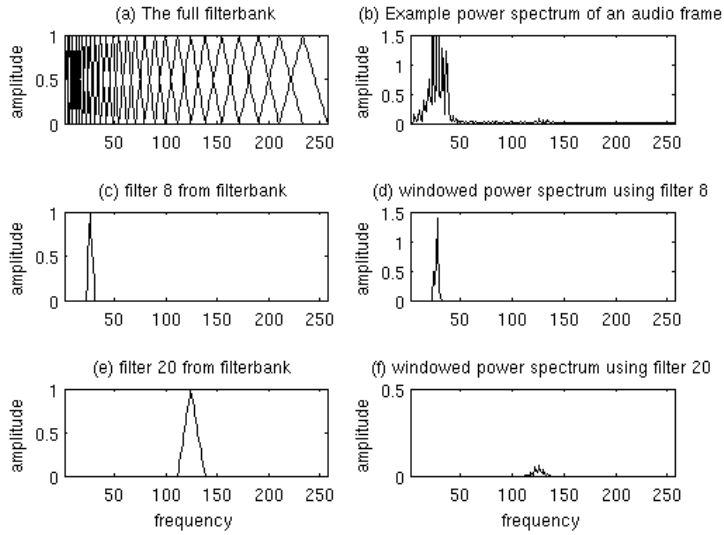
where $h(n)$ is an N sample long analysis window (e.g. hamming window), and K is the length of the DFT. The periodogram-based power spectral estimate for the speech frame $s_i(n)$ is given by:

$$P_i(k) = \frac{1}{N}|S_i(k)|^2$$

This is called the Periodogram estimate of the power spectrum. We take the absolute value of the complex fourier transform, and square the result. We would generally perform a 512 point FFT and keep only the first 257 coefficents.

3. Compute the Mel-spaced filterbank. This is a set of 20-40 (26 is standard) triangular filters that we apply to the periodogram power spectral estimate from step 2. Our filterbank comes in the form of 26 vectors of length 257 (assuming the FFT settings fom step 2). Each vector is mostly zeros, but is non-zero for a certain section of the spectrum. To calculate filterbank energies we multiply each filterbank with the power spectrum, then add up the coefficents. Once this is performed we are left with 26

numbers that give us an indication of how much energy was in each filterbank. For a detailed explanation of how to calculate the filterbanks see below. Here is a plot to hopefully clear things up:



Plot of Mel Filterbank and windowed power spectrum

4. Take the log of each of the 26 energies from step 3. This leaves us with 26 log filterbank energies.

5. Take the Discrete Cosine Transform (DCT) of the 26 log filterbank energies to give 26 cepstral coefficents. For ASR, only the lower 12-13 of the 26 coefficients are kept.

The resulting features (12 numbers for each frame) are called Mel Frequency Cepstral Coefficients.

Computing the Mel filterbank

In this section the example will use 10 filterbanks because it is easier to display, in reality you would use 26-40 filterbanks.

To get the filterbanks shown in figure 1(a) we first have to choose a lower and upper frequency. Good values are 300Hz for the lower and 8000Hz for the upper frequency. Of course if the speech is sampled at 8000Hz our upper frequency is limited to 4000Hz. Then follow these steps:

1. Using equation 1, convert the upper and lower frequencies to Mels. In our case 300Hz is 401.25 Mels and 8000Hz is 2834.99 Mels.
2. For this example we will do 10 filterbanks, for which we need 12 points. This means we need 10 additional points spaced linearly between 401.25 and 2834.99. This comes out to:

   ```
   m(i) = 401.25, 622.50, 843.75, 1065.00, 1286.25, 1507.50, 1728.74,
          1949.99, 2171.24, 2392.49, 2613.74, 2834.99
   ```

3. Now use equation 2 to convert these back to Hertz:

   ```
   h(i) = 300, 517.33, 781.90, 1103.97, 1496.04, 1973.32, 2554.33,
          3261.62, 4122.63, 5170.76, 6446.70, 8000
   ```

   Notice that our start- and end-points are at the frequencies we wanted.
4. We don't have the frequency resolution required to put filters at the exact points calculated above, so we need to round those frequencies to the nearest FFT bin. This process does not affect the accuracy of the features. To convert the frequncies to fft bin numbers we need to know the FFT size and the sample rate,

   ```
   f(i) = floor((nfft+1)*h(i)/samplerate)
   ```

   This results in the following sequence:

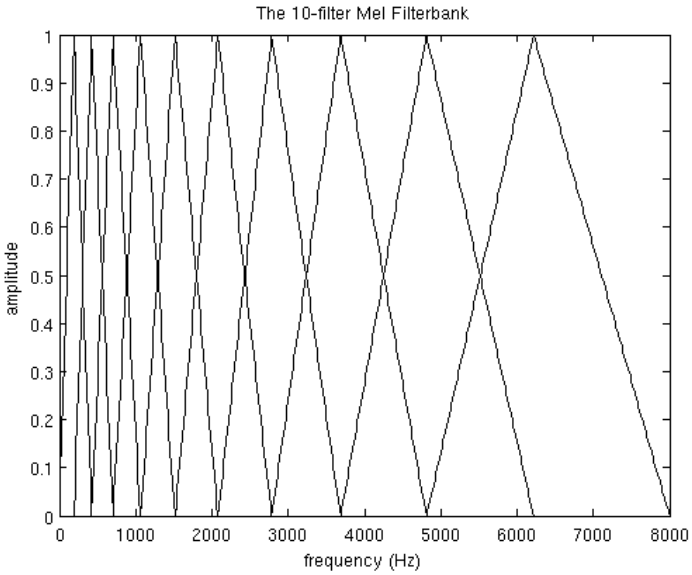
   ```
   f(i) =  9, 16,  25,   35,   47,   63,   81,  104,  132, 165,  206,  256
   ```

   We can see that the final filterbank finishes at bin 256, which corresponds to 8kHz with a 512 point FFT size.
5. Now we create our filterbanks. The first filterbank will start at the first point, reach its peak at the second point, then return to zero at the 3rd point. The second filterbank will start at the 2nd point, reach its max at the 3rd, then be zero at the 4th etc. A formula for calculating these is as follows:

$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \le k \le f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)} & f(m) \le k \le f(m+1) \\ 0 & k > f(m+1) \end{cases}$$

where M is the number of filters we want, and $f()$ is the list of M+2 Mel-spaced frequencies.

The final plot of all 10 filters overlayed on each other is:



A Mel-filterbank containing 10 filters. This filterbank starts at 0Hz and ends at 8000Hz. This is a guide only, the worked example above starts at 300Hz.

Deltas and Delta-Deltas

Also known as differential and acceleration coefficients. The MFCC feature vector describes only the power spectral envelope of a single frame, but it seems like speech would also have information in the dynamics i.e. what are the trajectories of the MFCC coefficients over time. It turns out that calculating the MFCC trajectories and appending them to the original feature vector increases ASR performance by quite a bit (if we have 12 MFCC coefficients, we would also get 12 delta coefficients, which would combine to give a feature vector of length 24).

To calculate the delta coefficients, the following formula is used:

$$d_t = \frac{\sum_{n=1}^{N} n(c_{t+n} - c_{t-n})}{2\sum_{n=1}^{N} n^2}$$

where d_t is a delta coefficient, from frame t computed in terms of the static coefficients c_{t+N} to c_{t-N}. A typical value for N is 2. Delta-Delta (Acceleration) coefficients are calculated in the same way, but they are calculated from the deltas, not the static coefficients.

Implementations

I have implemented MFCCs in python, available here. Use the 'Download ZIP' button on the right hand side of the page to get the code. Documentation can be found at readthedocs. If you have any troubles or queries about the code, you can leave a comment at the bottom of this page.

There is a good MATLAB implementation of MFCCs over here.

References

Davis, S. Mermelstein, P. (1980) *Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences*. In IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 28 No. 4, pp. 357-366

X. Huang, A. Acero, and H. Hon. *Spoken Language Processing: A guide to theory, algorithm, and system development*. Prentice Hall, 2001.

Related pages on this site:

- A tutorial on LPCCs and Cepstrum
- Hidden Markov Model (HMM) tutorial
- Gaussian Mixture Models (GMMs) and the EM Algorithm
- An Intuitive Guide to the Discrete Fourier Transform

ybl krq ibf kfnlh r kfsqyrdq mlxdqh mv trppvdqx
- tfqzstdsh

Questions/Feedback

Notice a problem? We'd like to fix it!
Leave a comment on the page and we'll take a look.

Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between

Speech processing plays an important role in any speech system whether its Automatic Speech Recognition (ASR) or speaker recognition or something else. Mel-Frequency Cepstral Coefficients (MFCCs) were very popular features for a long time; but more recently, filter banks are becoming increasingly popular. In this post, I will discuss filter banks and MFCCs and why are filter banks becoming increasingly popular.

Computing filter banks and MFCCs involve somewhat the same procedure, where in both cases filter banks are computed and with a few more extra steps MFCCs can be obtained. In a nutshell, a signal goes through a pre-emphasis filter; then gets sliced into (overlapping) frames and a window function is applied to each frame; afterwards, we do a Fourier transform on each frame (or more specifically a Short-Time Fourier Transform) and calculate the power spectrum; and subsequently compute the filter banks. To obtain MFCCs, a Discrete Cosine Transform (DCT) is applied to the filter banks retaining a number of the resulting coefficients while the rest are discarded. A final step in both cases, is mean normalization.

Setup

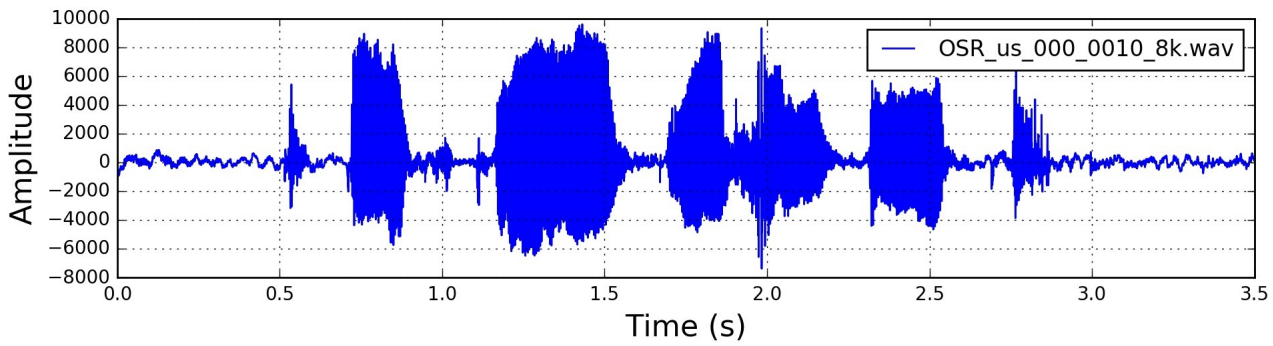
For this post, I used a 16-bit PCM wav file from here, called “OSR_us_000_0010_8k.wav”, which has a sampling frequency of 8000 Hz. The wav file is a clean speech signal comprising a single voice uttering some sentences with some pauses in-between. For simplicity, I used the first 3.5 seconds of the signal which corresponds roughly to the first sentence in the wav file.

I’ll be using Python 2.7.x, NumPy and SciPy. Some of the code used in this post is based on code available in this repository.

```
import numpy
import scipy.io.wavfile
from scipy.fftpack import dct

sample_rate, signal = scipy.io.wavfile.read('OSR_us_000_0010_8k.wav')  # File
assumed to be in the same directory
signal = signal[0:int(3.5 * sample_rate)]  # Keep the first 3.5 seconds
```

The raw signal has the following form in the time



Pre-Emphasis

The first step is to apply a pre-emphasis filter on the signal to amplify the high frequencies. A pre-emphasis filter is useful in several ways: (1) balance the frequency spectrum since high frequencies usually have smaller magnitudes compared to lower frequencies, (2) avoid numerical problems during the Fourier transform operation and (3) may also improve the Signal-to-Noise Ratio (SNR).

The pre-emphasis filter can be applied to a signal x

using the first order filter in the following equation:

$$y(t)=x(t)-\alpha x(t-1)$$

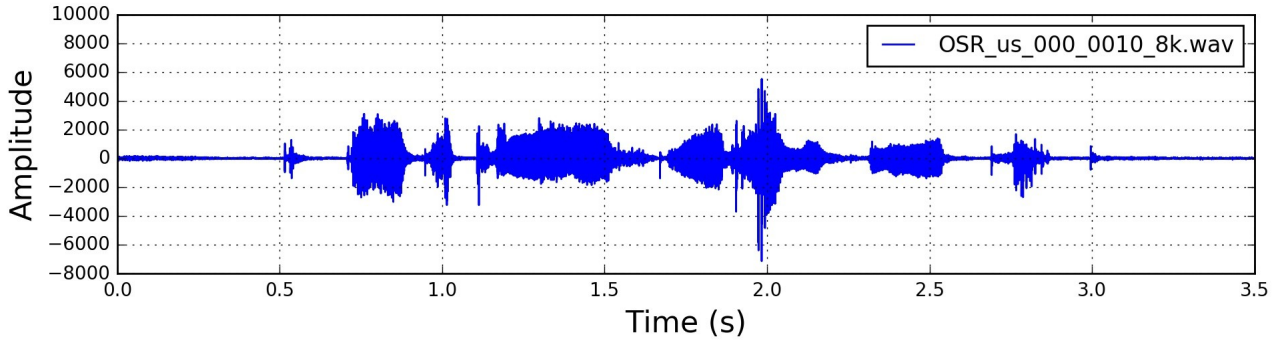
which can be easily implemented using the following line, where typical values for the filter coefficient (α

) are 0.95 or 0.97, `pre_emphasis = 0.97`:

```
emphasized_signal = numpy.append(signal[0], signal[1:] - pre_emphasis *
signal[:-1])
```

Pre-emphasis has a modest effect in modern systems, mainly because most of the motivations for the pre-emphasis filter can be achieved using mean normalization (discussed later in this post) except for avoiding the Fourier transform numerical issues which should not be a problem in modern FFT implementations.

The signal after pre-emphasis has the following form in the time domain



Framing

After pre-emphasis, we need to split the signal into short-time frames. The rationale behind this step is that frequencies in a signal change over time, so in most cases it doesn't make sense to do the Fourier transform across the entire signal in that we would lose the frequency contours of the signal over time. To avoid that, we can safely assume that frequencies in a signal are stationary over a very short period of time. Therefore, by doing a Fourier transform over this short-time frame, we can obtain a good approximation of the frequency contours of the signal by concatenating adjacent frames.

Typical frame sizes in speech processing range from 20 ms to 40 ms with 50% (+/-10%) overlap between consecutive frames. Popular settings are 25 ms for the frame size, `frame_size = 0.025` and a 10 ms stride (15 ms overlap), `frame_stride = 0.01`.

```
frame_length, frame_step = frame_size * sample_rate, frame_stride * sample_rate
# Convert from seconds to samples
signal_length = len(emphasized_signal)
frame_length = int(round(frame_length))
frame_step = int(round(frame_step))
num_frames = int(numpy.ceil(float(numpy.abs(signal_length - frame_length)) /
frame_step))  # Make sure that we have at least 1 frame

pad_signal_length = num_frames * frame_step + frame_length
z = numpy.zeros((pad_signal_length - signal_length))
pad_signal = numpy.append(emphasized_signal, z) # Pad Signal to make sure that
all frames have equal number of samples without truncating any samples from the
original signal

indices = numpy.tile(numpy.arange(0, frame_length), (num_frames, 1)) +
numpy.tile(numpy.arange(0, num_frames * frame_step, frame_step), (frame_length,
1)).T
frames = pad_signal[indices.astype(numpy.int32, copy=False)]
```

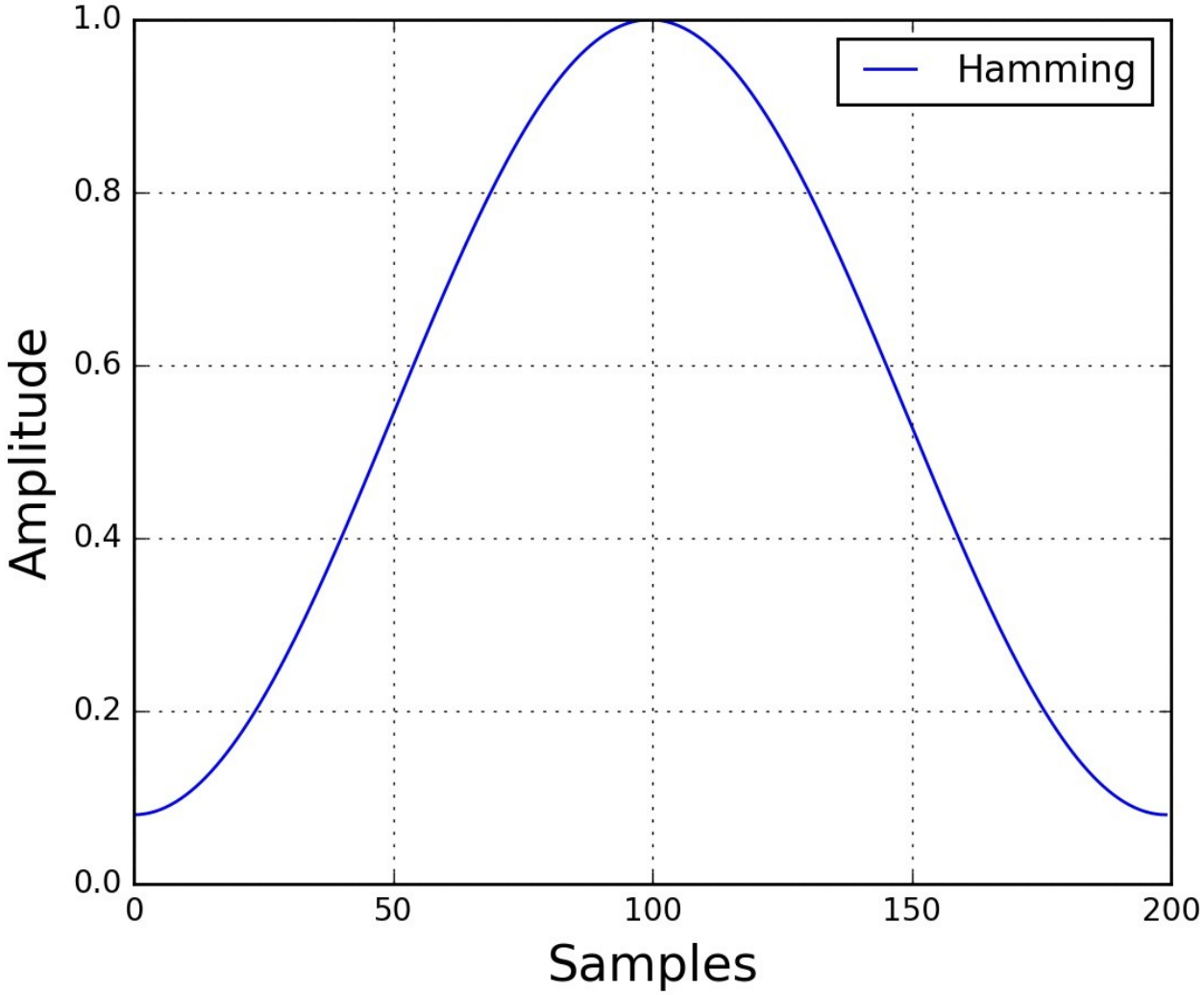
Window

After slicing the signal into frames, we apply a window function such as the Hamming window to each frame. A Hamming window has the following form:

$$w[n]=0.54-0.46cos(2\pi nN-1)$$

where, $0\leq n\leq N-1$

, N is the window length. Plotting the previous equation yields the following plot


Hamming
Amplitude
Samples
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200

There are several reasons why we need to apply a window function to the frames, notably to counteract the assumption made by the FFT that the data is infinite and to reduce spectral leakage.

```
frames *= numpy.hamming(frame_length)
# frames *= 0.54 - 0.46 * numpy.cos((2 * numpy.pi * n) / (frame_length - 1))  #
Explicit Implementation **
```

Fourier-Transform and Power Spectrum

We can now do an N

-point FFT on each frame to calculate the frequency spectrum, which is also called Short-Time Fourier-Transform (STFT), where N

is typically 256 or 512, `NFFT = 512`; and then compute the power spectrum (periodogram) using the following equation:

$$P=|FFT(xi)|2N$$

where, xi

is the ith frame of signal x

. This could be implemented with the following lines:

```
mag_frames = numpy.absolute(numpy.fft.rfft(frames, NFFT))  # Magnitude of the
FFT
pow_frames = ((1.0 / NFFT) * ((mag_frames) ** 2))  # Power Spectrum
```

Filter Banks

The final step to computing filter banks is applying triangular filters, typically 40 filters, `nfilt = 40` on a Mel-scale to the power spectrum to extract frequency bands. The Mel-scale aims to mimic the non-linear human ear perception of sound, by being more discriminative at lower frequencies and less discriminative at higher frequencies. We can convert between Hertz (f

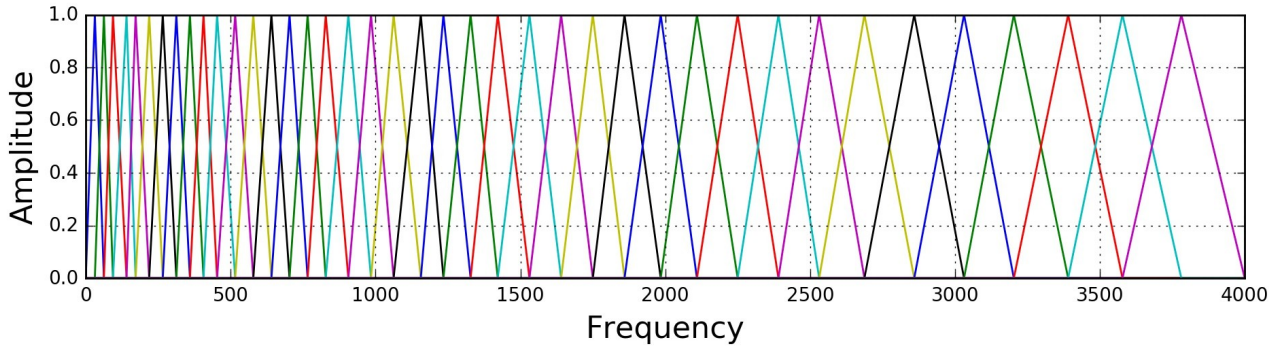
) and Mel (m

) using the following equations:

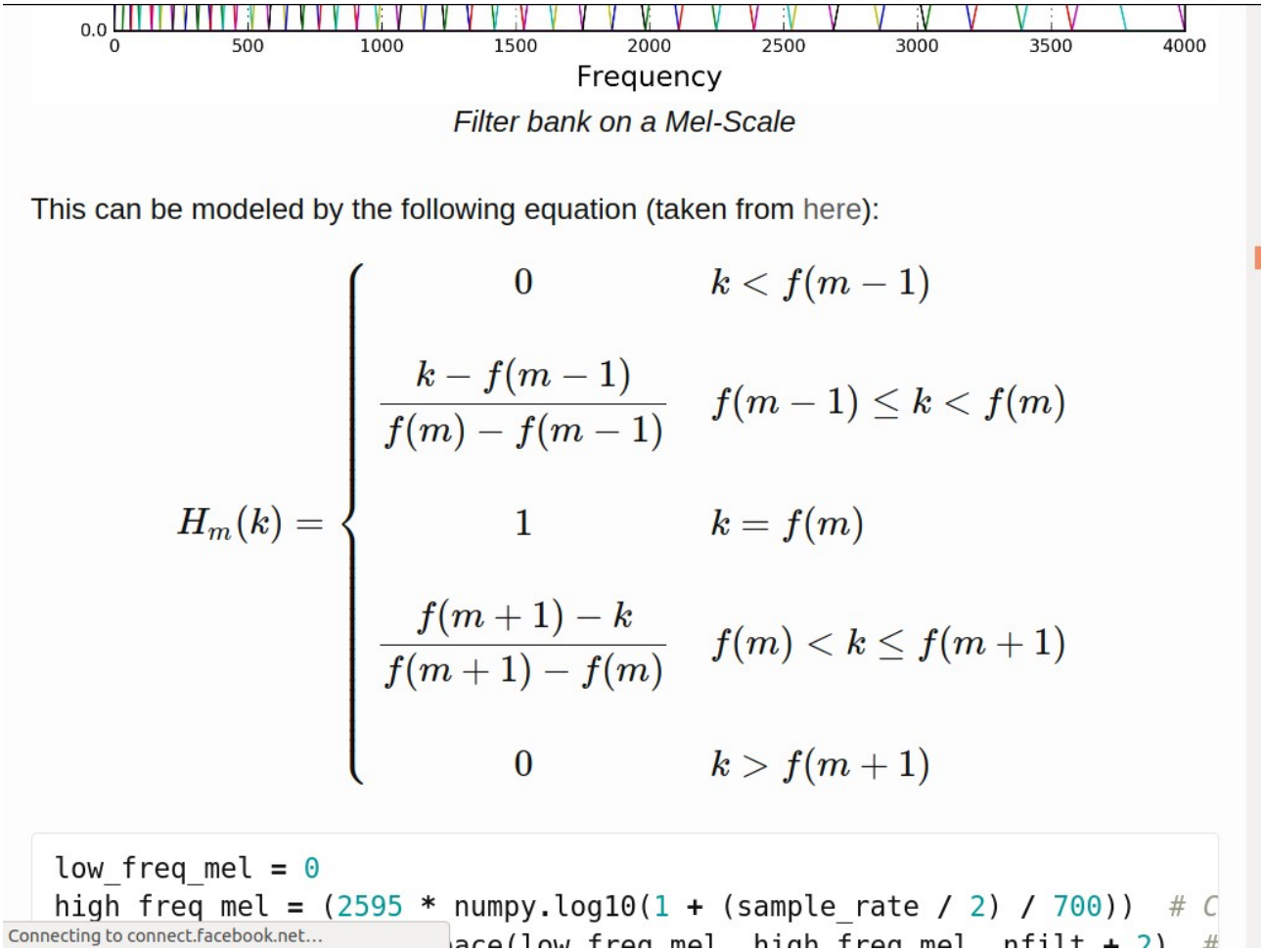
$$m=2595\log10(1+f700)$$

$$f=700(10m/2595−1)$$

Each filter in the filter bank is triangular having a response of 1 at the center frequency and decrease linearly towards 0 till it reaches the center frequencies of the two adjacent filters where the response is 0, as shown in this figure



This can be modeled by the following equation (taken from here):



Filter bank on a Mel-Scale

This can be modeled by the following equation (taken from here):

$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \le k < f(m) \\ 1 & k = f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)} & f(m) < k \le f(m+1) \\ 0 & k > f(m+1) \end{cases}$$

```
low_freq_mel = 0
high_freq_mel = (2595 * numpy.log10(1 + (sample_rate / 2) / 700))  # C
```

Connecting to connect.facebook.net...

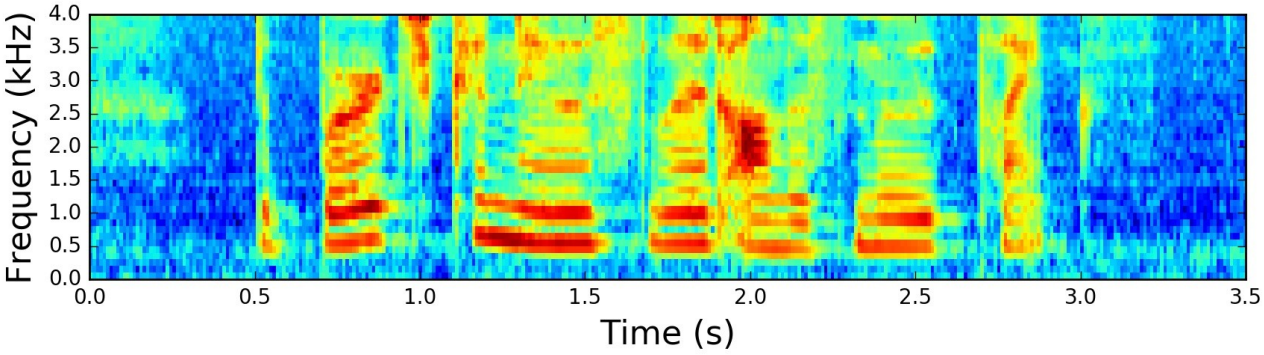
```
low_freq_mel = 0
high_freq_mel = (2595 * numpy.log10(1 + (sample_rate / 2) / 700))  # Convert Hz
to Mel
mel_points = numpy.linspace(low_freq_mel, high_freq_mel, nfilt + 2)  # Equally
spaced in Mel scale
hz_points = (700 * (10**(mel_points / 2595) - 1))  # Convert Mel to Hz
bin = numpy.floor((NFFT + 1) * hz_points / sample_rate)

fbank = numpy.zeros((nfilt, int(numpy.floor(NFFT / 2 + 1))))
for m in range(1, nfilt + 1):
    f_m_minus = int(bin[m - 1])   # left
```

```
    f_m = int(bin[m])             # center
    f_m_plus = int(bin[m + 1])    # right

    for k in range(f_m_minus, f_m):
        fbank[m - 1, k] = (k - bin[m - 1]) / (bin[m] - bin[m - 1])
    for k in range(f_m, f_m_plus):
        fbank[m - 1, k] = (bin[m + 1] - k) / (bin[m + 1] - bin[m])
filter_banks = numpy.dot(pow_frames, fbank.T)
filter_banks = numpy.where(filter_banks == 0, numpy.finfo(float).eps,
filter_banks)  # Numerical Stability
filter_banks = 20 * numpy.log10(filter_banks)  # dB
```

After applying the filter bank to the power spectrum (periodogram) of the signal, we obtain the following spectrogram



If the Mel-scaled filter banks were the desired features then we can skip to mean normalization.

Mel-frequency Cepstral Coefficients (MFCCs)

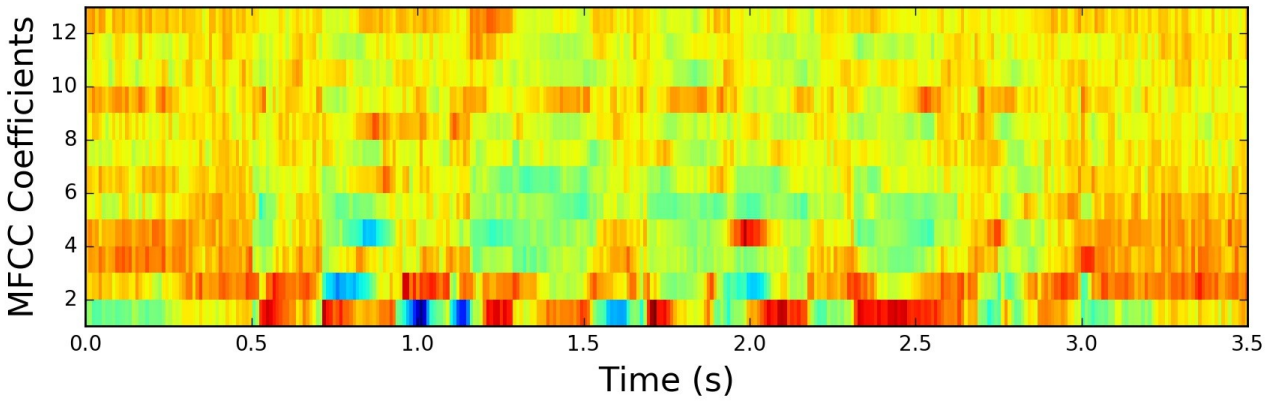
It turns out that filter bank coefficients computed in the previous step are highly correlated, which could be problematic in some machine learning algorithms. Therefore, we can apply Discrete Cosine Transform (DCT) to decorrelate the filter bank coefficients and yield a compressed representation of the filter banks. Typically, for Automatic Speech Recognition (ASR), the resulting cepstral coefficients 2-13 are retained and the rest are discarded; `num_ceps = 12`. The reasons for discarding the other coefficients is that they represent fast changes in the filter bank coefficients and these fine details don't contribute to Automatic Speech Recognition (ASR).

```
mfcc = dct(filter_banks, type=2, axis=1, norm='ortho')[:, 1 : (num_ceps + 1)] #
Keep 2-13
```

One may apply sinusoidal liftering1 to the MFCCs to de-emphasize higher MFCCs which has been claimed to improve speech recognition in noisy signals.

```
(nframes, ncoeff) = mfcc.shape
n = numpy.arange(ncoeff)
lift = 1 + (cep_lifter / 2) * numpy.sin(numpy.pi * n / cep_lifter)
mfcc *= lift  #*
```

The resulting MFCCs

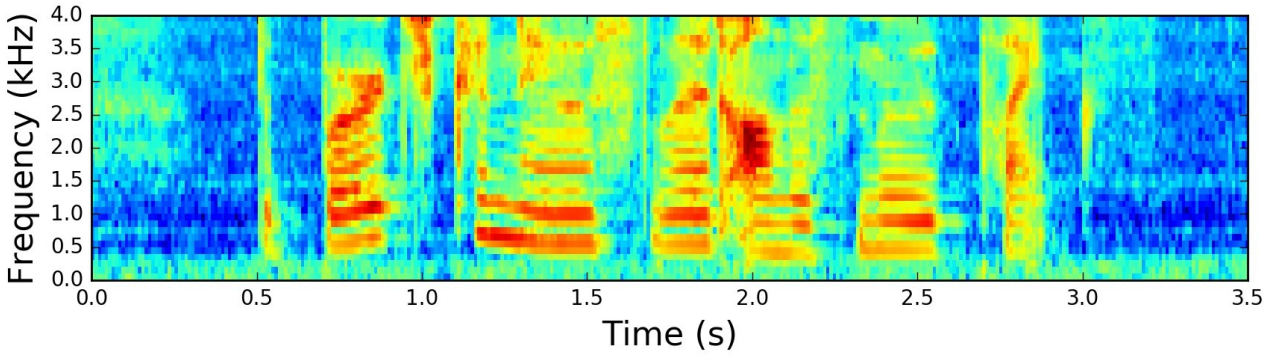


Mean Normalization

As previously mentioned, to balance the spectrum and improve the Signal-to-Noise (SNR), we can simply subtract the mean of each coefficient from all frames.

```
filter_banks -= (numpy.mean(filter_banks, axis=0) + 1e-8)
```

The mean-normalized filter banks

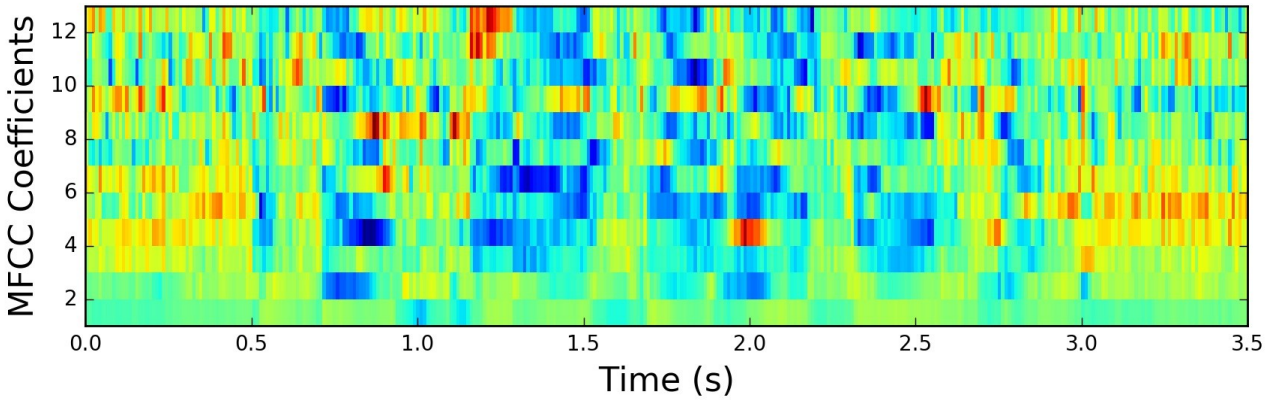


Normalized Filter Banks

and similarly for MFCCs:

```
mfcc -= (numpy.mean(mfcc, axis=0) + 1e-8)
```

The mean-normalized MFCCs:



Normalized MFCCs

Filter Banks vs MFCCs

To this point, the steps to compute filter banks and MFCCs were discussed in terms of their motivations and implementations. It is interesting to note that all steps needed to compute filter banks were motivated by the nature of the speech signal and the human perception of such signals. On the contrary, the extra steps needed to compute MFCCs were motivated by the limitation of some machine learning algorithms. The Discrete Cosine Transform (DCT) was needed to decorrelate filter bank coefficients, a process also referred to as whitening. In particular, MFCCs were very popular when Gaussian Mixture Models - Hidden Markov Models (GMMs-HMMs) were very popular and together, MFCCs and GMMs-HMMs co-evolved to be the standard way of doing Automatic Speech Recognition (ASR)[2]. With the advent of Deep Learning in speech systems, one might question if MFCCs are still the right choice given that deep neural networks are less susceptible to highly correlated input and therefore the Discrete Cosine Transform (DCT) is no longer a necessary step. It is beneficial to note that Discrete Cosine Transform (DCT) is a linear transformation, and therefore undesirable as it discards some information in speech signals which are highly non-linear.

It is sensible to question if the Fourier Transform is a necessary operation. Given that the Fourier Transform itself is also a linear operation, it might be beneficial to ignore it and attempt to learn directly from the signal in the time domain. Indeed, some recent work has already attempted this and positive results were reported. However, the Fourier transform operation is a difficult operation to learn and may arguably increase the amount of data and model complexity needed to achieve the same performance. Moreover, in doing Short-Time Fourier Transform (STFT), we've assumed the signal to be stationary within this short time and therefore the linearity of the Fourier transform would not pose a critical problem.

Conclusion

In this post, we've explored the procedure to compute Mel-scaled filter banks and Mel-Frequency Cepstrum Coefficients (MFCCs). The motivations and implementation of each step in the procedure were discussed. We've also argued the reasons behind the increasing popularity of filter banks compared to MFCCs.

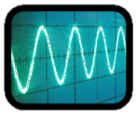
tl;dr: Use Mel-scaled filter banks if the machine learning algorithm is not susceptible to highly correlated input. Use MFCCs if the machine learning algorithm is susceptible to correlated input.

---

1. Liftering is filtering in the cepstral domain. Note the abuse of notation in *spec*tral and *ceps*tral with *fil*tering and *lif*tering respectively. ↩
2. An excellent discussion on this topic is in this thesis. ↩

Discrete Cosine Transformations

The topic of this post is the *Discrete Cosine Transformation*, abbreviated pretty universally as DCT.



DCTs are used to convert data into the summation of a series of cosine waves oscillating at different frequencies (more on this later). They are widely used in image and audio compression.

They are very similar to *Fourier Transforms*, but DCT involves the use of just Cosine functions and real coefficients, whereas *Fourier Transformations* make use of both Sines and Cosines and require the use of complex numbers. DCTs are simpler to calculate. Both Fourier and DCT convert data from a *spatial-domain* into a *frequency-domain* and their respective inverse functions convert things back the other way.

Why are DCTs so useful? As mentioned above they are used extensively in image and audio compression. To compress analog signals, often we discard information (called *lossy compression*), to enable efficient compaction. We have to be carefaul about what information in a signal we should discard (or smooth out) when removing bits to compress a signal. DCT helps with this process.

Thankfully, our eyes, ears and brain are analog devices and we are less sensitive to distortion around edges, and we are less likely to notice subtle differences fine textures. Also, for many audio signals and graphical images the amplitudes and pixels are often similar to their near neighbors. These factors provide a solution; if we are careful at removing the *higher-frequency* elements of an analog signal (things that change between short 'distances' in the data) there is a good chance that, if we don't take this too far, our brains might not perceive a difference.

JPEG

The JPEG *(Joint Photographic Experts Group)* format uses DCT to compress images (we'll describe how later). Below is a test image of some fern leaves. In raw format this image is 67,854 bytes in size. To the right of it are a series of images made with increasing levels of compression. At each stage, the image storage size gets smaller, but frequency information in the image is lost as increasingly higher compression is applied. With a small amount of compression, it's practically impossible for the brain to notice the difference. As we move further to the right, defects become more obvious.

File Size: 67,854 bytes | File Size: 7,078 bytes | File Size: 5,072 bytes | File Size: 3,518 bytes | File Size: 2,048 bytes | File Size: 971 bytes

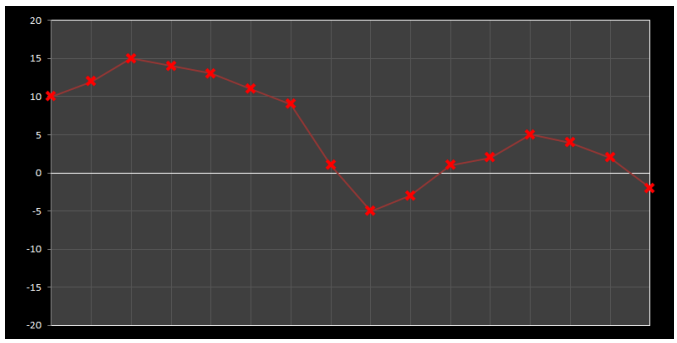
How is this compression achieved? By using a DCT transform, the image is shifted into the frequency domain. Then, depending on how much compression is required, the higher frequency coefficients of the signal are masked off and removed (the digital equivalent of applying a *low-pass analog filter*). When the image is recreated using the truncated coefficients, the higher frequency components are not present.

Notice the 'blocky' appearance of the image on the far right? This is an artifact of how the DCT compression is calculated. Again, we'll look at this later.

Example: One Dimension

Let's start with analysis in one dimension. Imagine we have 16 points as shown below. Time is nominally on the x-axis, with a variety of values on the y-axis. We're going to apply DCT to these data and see the effects how these individual cosine components add together to approximate the source.

The math required to permform DCT is not very complicated, but it's also not rated **PG-13**. I'm opening with the concepts, then I'm going to brush straight past the implementation steps and move onto the results. For those that like to code, and want to experiment with this, let me give the standard advice: "*"Google is your friend!"*" – you can find plenty of implementations of DCT in the language of your choice on the web.

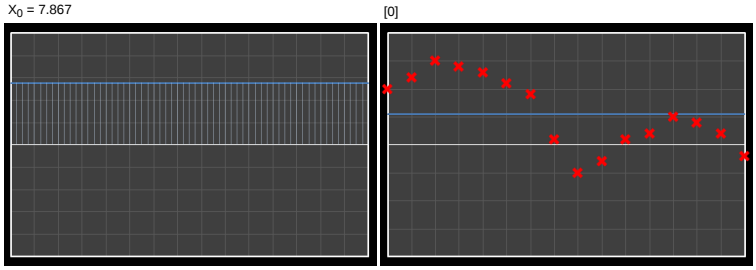


Using DCT, we can break this curve into a series of Cosine waves of various frequencies. It is by the superposition (adding together) of these fundamental waves that we recreate the original waveform.

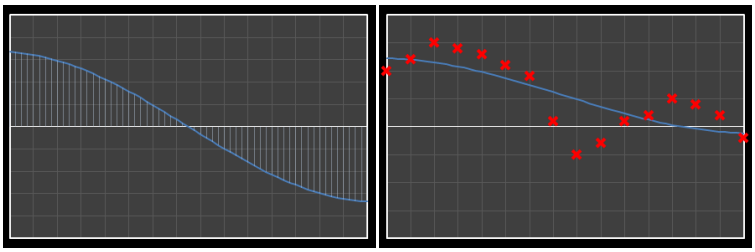
For the above sixteen points, I've broken down the data using DCT. The graphs on the left below show the cosine function of that frequency component and the image on the right shows the superposition of this to the running total (this component added to all those above it). The lowest frequency components are shown first, and as we move down the page, the higher frequency components are added. Above each graph is a number showing the coefficient 'weight' of each frequency component. Typically (though not always), these numbers get smaller as the contribution to the overall shape from these higher frequencies gets smaller.

As we move down the charts we can see that shape getting closer and closer in approximation to the original data (the errors between the actual data and the curve approximation getting smaller – the differences being the higher frequency 'wiggles' in the raw data). Depending on the level of compression we needed we could truncate the higher frequency components as needed and decided where to draw the line at a 'good-enough' approximation.

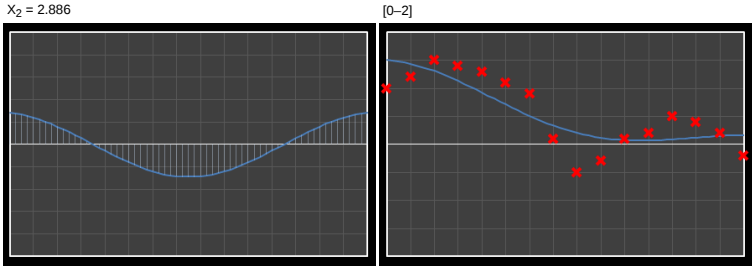
$X_0 = 7.867$ [0]



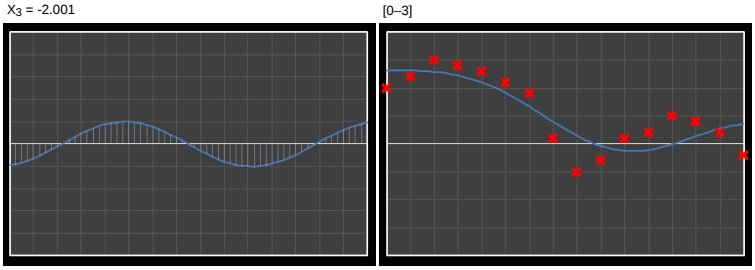
$X_1 = 6.742$ [0–1]



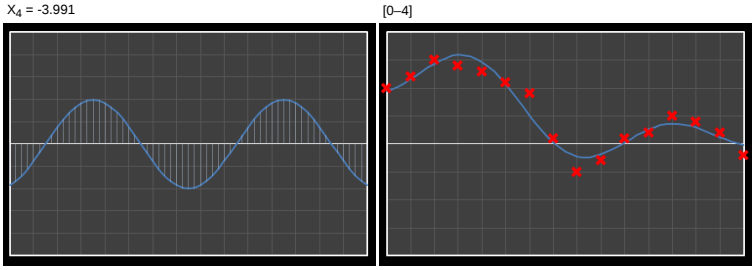
X_2 = 2.886 [0–2]



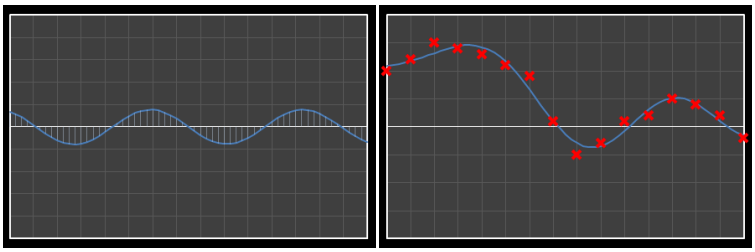
X_3 = -2.001 [0–3]



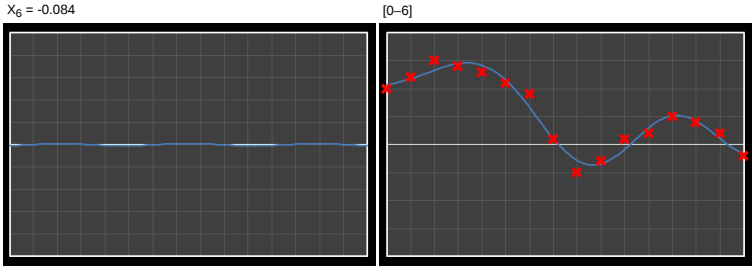
X_4 = -3.991 [0–4]



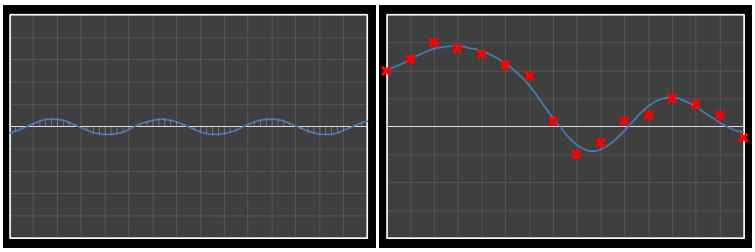
X_5 = 1.546 [0–5]



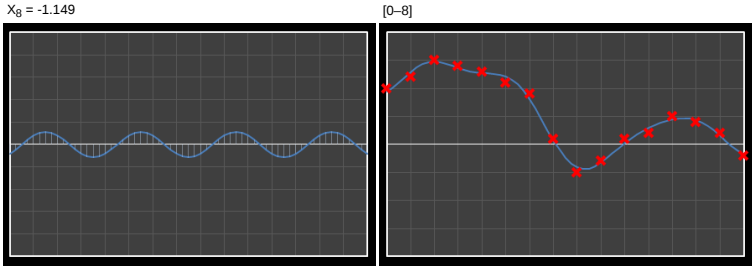
X_6 = -0.084 [0–6]



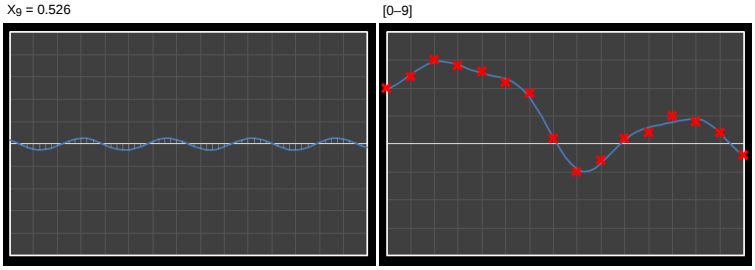
X_7 = -0.703 [0–7]



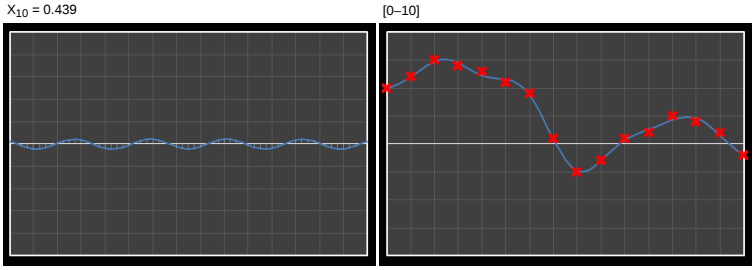
X_8 = -1.149 [0–8]



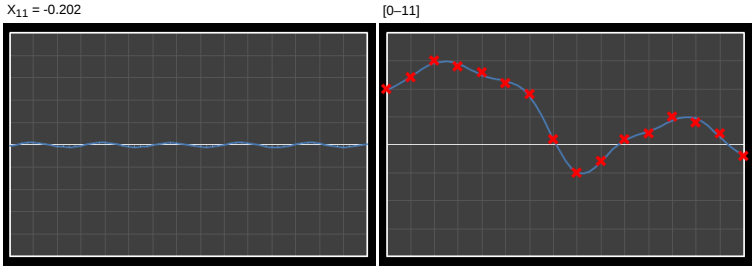
X_9 = 0.526 [0–9]



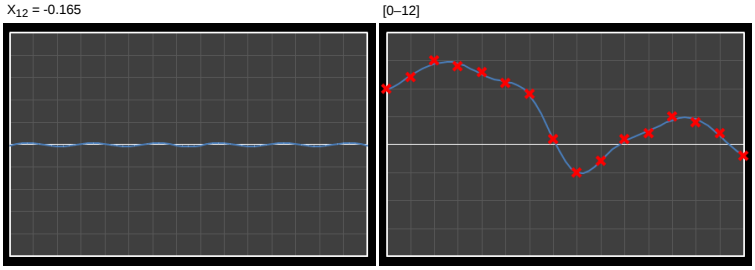
X_{10} = 0.439 [0–10]



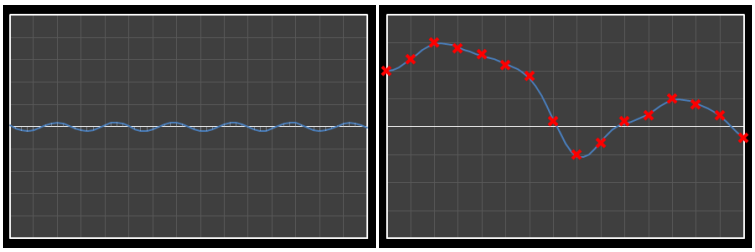
X_{11} = -0.202 [0–11]



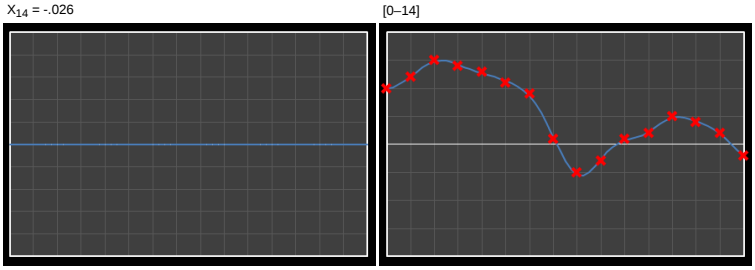
X_{12} = -0.165 [0–12]



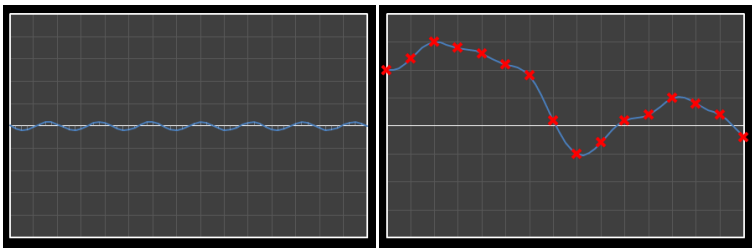
X_{13} = 0.382 [0–13]



X_{14} = -.026 [0–14]

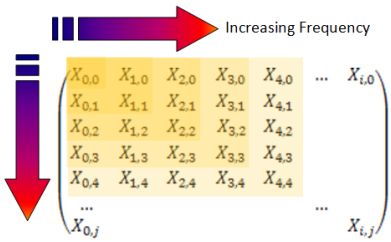


X_{15} = 0.368 [0–15]



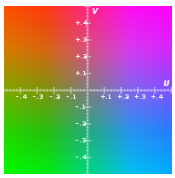
Example: Two Dimensions (and the basics of JPEG compression)

We can apply just the same technique in two dimensions, this time breaking the image into blocks of pixels and looking at the harmonics in each block. The result of this analysis is a matrix of coefficients. Moving down and to the right are the coefficients with increasingly higher frequency components. As before, we can compress an image by masking off and truncating the coefficients at frequencies higher than we care about.



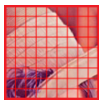
In JPEG compression, the block size used is 8 x 8, resulting in a similar size matrix of coefficients. However, for my examples, I'm going to select a larger block size of 16 x 16 (I think it is easier to visualize things at this size.)

As you probably know, colors in computer images are often describe by the relative mixing of their Red , Green and Blue components. This is, internally, how computers store the image data, but this is not the only way to describe colors. Another method is called YUV. It's outside the scope of this article (follow the link for more background), but this format describes colors by their *luminance* and *chrominance* (Sort of like the brightness of the color and the shade of color).



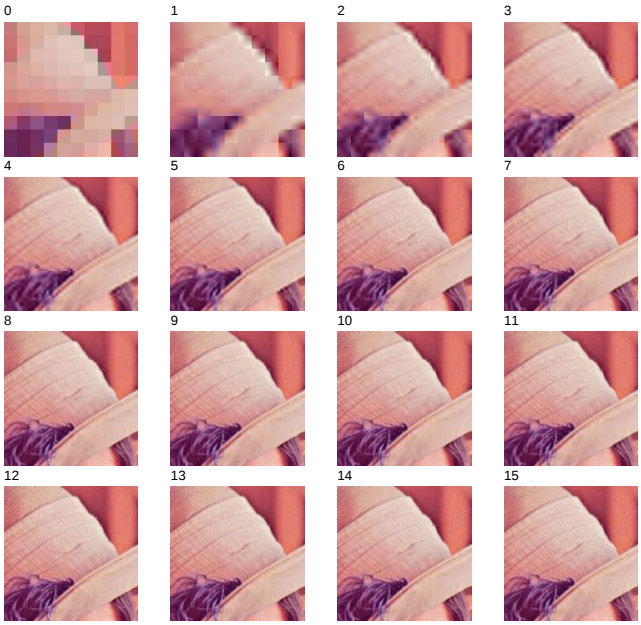
Our eyes are more sensitive to changes in brightness than changes in shade, and this can be exploited for more compression by converting RGB colors in YUV space, then UV channels can be sub-sampled (quantized) and reduced in dynamic range – Because of lower senstitivity you need *fewer distinct levels of shade of a color* than the brightness of a color to still maintain a smooth image.

Because we are working on image processing, we have to roll out Lenna, the "*The First Lady of the Internet*". This image is probably the most widely used test image in computer history (click on the link for background details).



We're going to look at a selection of 16 x 16 pixel blocks around the hat. Below are sixteen version of this region rendered using coefficients truncated at various levels. Initially the individual blocks are clearly visible. As the coefficients are truncated higher, the edges

of the blocks become less discernable but the images still look a little blurry. As the coefficient cut-off increases, the images get sharper.

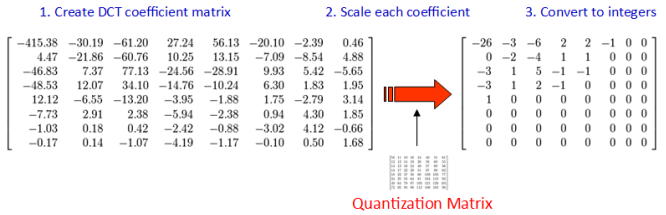


JPEG Advanced

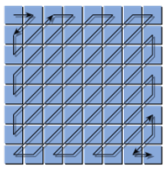
The above paragraphs explain the concept of how JPEG compression works (reduction of the higher frequency components), in reality, it is a little more complicated.

The DCT is applied over an 8 x 8 block, to produce an 8 x 8 matrix of frequency coefficients. Each block is calculated distinctly, and the color of the top-left pixel of each block determines a fixed reference and the other pixels in the same block are described *relative* to this pixel (this helps reduce the dynamic range needed for the DCT function, and typically benefits the quality of the image as often there is not a massive variation in colors between pixels in the same block).

Rather than simply truncating and masking off the higher frequency components of the DCT matrix as we did above, however, in JPEG the frequency coefficients are scaled (individually) by a *quantization matrix*. This matrix scales (divides) each coefficient by a numeric term. The quantization matrix is pre-calculated and defined by the JPEG standard and naturally favors the items in the top left corner of the matrix (the more frequency significant terms) than the lower right. Each coefficient has a different weighting.



The results of this division are then converted to integers (from floating point), causing further quantization. A typical output matrix of this process is show above on the right. Notice this, as we expect, biases the top left corner where we know the most *'important'* coefficients reside.



The final step in the JPEG compression is called *'entropy encoding'* and this is the mechanism for how the coefficients of the final matrix are encapsulated.

The technique used is a run-length encoding algorithm which losslessly compresses the matrix because there is often redundancy of multiple occurrences of repeated values. This works especially well because, rather than simply reading the values in a traditional, grid format, the input stream to the compression *zig-zags* through the matrix starting at the top left corner, and ending in the lower right.

As you can appreciate, this increases the chances that adjacent values will be of similar value (and often, as you can see in our example, it's typical for there to be many zeros at the end of the string which compact very well indeed!)

Finally, let's end with a little bit of fun ...

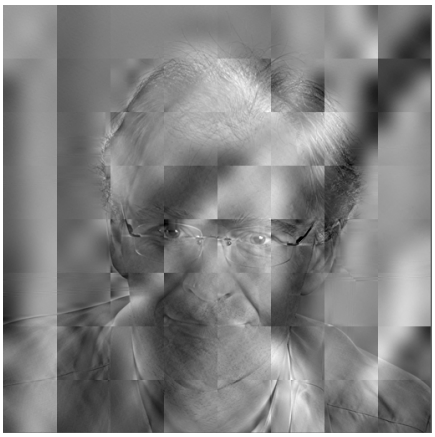
It is possible to merge two images with different spatial frequencies.

When this hybrid image is viewed from a close distance, the higher-frequency elements stand out, revealing these components of the image in high contrast. When the hybrid image is viewed from a far distance, these higher frequency subtleties are not discernible and the eye/brain smoothes and interpolates the lower frequency components.

To make a hybrid image containing two images we use the digital equivalent of a high-pass and low-pass filter. In the example below, I've taken the *Lena* image and passed it through DCT to convert it to the frequency domain. (To make this a little clearer, for this example, I've also converted the image to gray scale and also used a larger block size). Then I removed the high frequency coefficients from the matrix (a low pass filter).

Next I took another image (this time a self-portrait), converted this to gray scale, passed it through DCT, but this time only preserved the higher frequency coefficients of the frequency matrix (a high pass filter).

These two sparse matrices were then combined and passed into the inverse DCT function. Here is the hybrid result:



If you look at the hybrid image whilst sitting directly infront of your monitor/tablet you should clearly see the ghostly outline of my face and shirt (lucky you!). Now get up and walk to the other side of the room and look at the image again. This time you should see the Lena image, and only the Lena image – my face has gone. Spooky!

Can't see it? Click here to show the image slowly reduced for you.

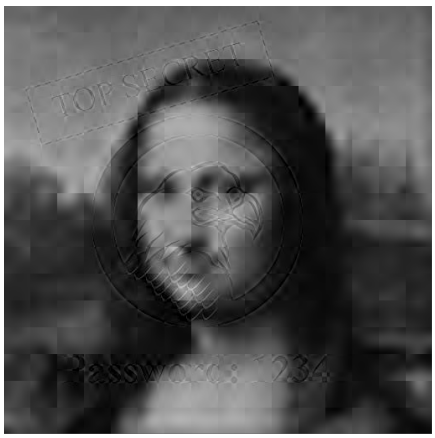
You can also experience the same effect by squinting at the hybrid image. Squinting artificially reduces the size of your pupil creating a Pinhole effect, increasing the depth of field for your eyes.

Hidden Letters

This hybridization is not just academic. Have you ever been paranoid that someone is looking over your shoulder as you use the ATM?, or watching you from the opposite side of the aisle you type something sensitive on your laptop during a flight?

Imagine if these devices used a hybrid image technology! They could display a hybrid image created from a superposition of different sources. Not only would the evesdropper not see what you are seeing, but you could make the *"fake"* image (the low-frequency domain image) display bogus information. Only someone viewing from the appropriate spatial-distance would see the correct image (with care, more than two images could be combined by selecting the appropriate *band-pass* filters and combining the resultant matrices).

To show an example of this look at this hybrid image below of the *Mona Lisa*. Viewed close up you should be able to make out the password. Now look at the screen from a few feet away. See it now?

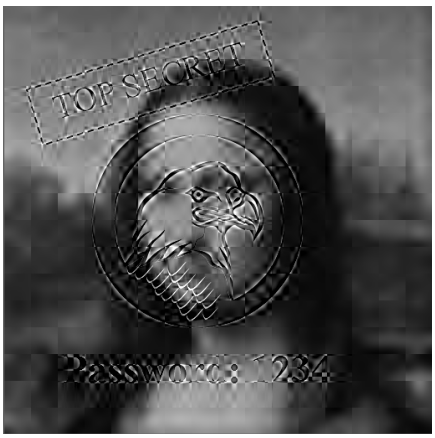


Need more convincing?

The image below is the same image as the one to the left. All I've done, if you *View Source* for the page, is overide the width of the image from the default of 512 pixels to 128 pixels. Your browser has scaled down the image using an appropiate algorithm, and has averaged out the pixels, reducing the higher frequency components.

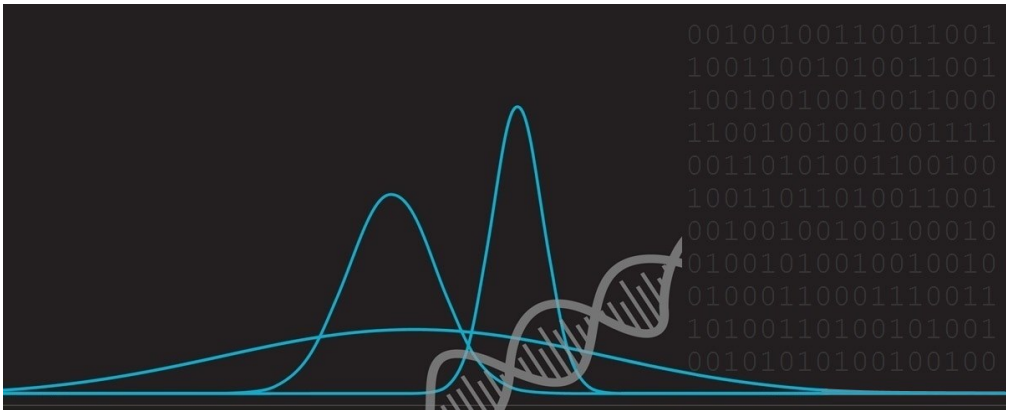
This is similar to what happens when you stand further away from the screen.

Can't see it? Here's a version where the *"Hidden Text"* is even higher contrast.



If you squint at this one, you should still be able to make the hidden text go away.

You can find a complete list of all the articles here. Click here to receive email alerts on new articles.



- PythonInMusic
- PythonInMusic
- FrontPage
- RecentChanges
- FindPage
- HelpContents
- PythonInMusic

Page

- Immutable Page
- Info
- Attachments
- More Actions:

User

- Login

This page is divided in four sections: Music software written in Python, Music software supporting Python, Music programming in Python, and a category of unsorted (may still fit in the above)

Music software written in Python

Audio Players

- **Bluemindo** - Bluemindo is a really simple but powerful audio player in Python/PyGTK, using Gstreamer. Bluemindo is a free (as in freedom) software, released under GPLv3, only.
- **cplay** - a curses front-end for various audio players
- **edna** - an MP3 server, edna allows you to access your MP3 collection from any networked computer. The web pages are dynamically constructed, adjusting to directory structure and the files in those directories. This is much nicer than using simple directory indexing. Rather than directly serving up an MP3, the software serves up a playlist. This gets

passed to your player (e.g. WinAmp) which turns around with an HTTP request to stream the MP3.

- **Listen** - Music management and playback for GNOME
- **MediaCore Audio/Podcast Player and CMS** - Web based CMS for music management in video, audio and podcast form. All audio, video, and podcasts added to the system are playable from any browser.
- **MMA** - Musical Midi Accompaniment. If you follow the above link you will find that Pymprovisator is no longer developed due to the fact that there is this similar, but more powerful GPL Python software.
- **Peyote** - Peyote is an audio player with friendly MC-like interface. Peyote is designed specifically for work easy with cue sheets.
- **Pymprovisator** - Pymprovisator is a program that emulates the program Band in a Box from PG Music. You can think in it like the electronic version of the books+CD from Jamey Aebersold. You set the basic parameters in a song: title, style, key, chords sequence,... and the program will generate a Midi file with the correct accompaniment. (dev suspended)
- **Pymps** - Pymps is the PYthon Music Playing System - a web based mp3/ogg jukebox. It's written in Python and utilises the PostgreSQL database.
- **MusicPlayer** - MusicPlayer is a high-quality music player implemented in Python, using FFmpeg and PortAudio.
- **Pymserv** - PyMServ is a graphical client for mserv, a music server. It is written in Python using pygtk and gconf to store prefs.
- **Pytone** - Pytone is a music jukebox written in Python with a curses based GUI. While providing advanced features like crossfading and multiple players, special emphasis is put on ease of use, turning PyTone into an ideal jukebox system for use at parties.
- **Quod Libet** - Quod Libet is a GTK+-based audio player written in Python. It lets you make playlists based on regular expressions. It lets you display and edit any tags you want in the file. And it lets you do this for all the file formats it supports -- Ogg Vorbis, FLAC, MP3, Musepack, and MOD.
- **TheTurcanator** - a small midi piano tutor for windows and mac. Includes CoreMIDI wrapper written in pyrex.
- **LinuxBand** - LinuxBand is a GUI front-end for MMA (Musical MIDI Accompaniment). Type in the chords, choose the groove and LinuxBand will play a musical accompaniment for you.

Audio Convertors

- **audio-convert-mod** - audio-convert-mod is a simple audio file converter that supports many formats. At just a right-click, you can convert any amount of music files to WAV, MP3, AAC, Ogg and more. audio-convert-mod was designed with the same principles as fwbackups - keeping things

simple.

- **SoundConverter** - SoundConverter is a simple audio file converter for the GNOME desktop, using GStreamer for conversion. It can read anything GStreamer has support for, and writes to WAV, MP3, AAC, Ogg or FLAC files.
- **Python Audio Tools** - Python audio tools are a collection of audio handling programs which work from the command line. These include programs for CD extraction, track conversion from one audio format to another, track renaming and retagging, track identification, CD burning from tracks, and more. Supports internationalized track filenames and metadata using Unicode. Works with high-definition, multi-channel audio as well as CD-quality. Track conversion uses multiple CPUs or CPU cores if available to greatly speed the transcoding process. Track metadata can be retrieved from FreeDB, MusicBrainz or compatible servers. Audio formats supported are: WAV, AIFF, FLAC, Apple Lossless, Shorten, WavPack, MP3, MP2, M4A, Ogg Vorbis, Ogg Speex, Ogg FLAC, & Sun AU

Music Notation

- **Abjad** - Abjad is a Python API for Formalized Score Control. Abjad is designed to help composers build up complex pieces of music notation in an iterative and incremental way. You can use Abjad to create a symbolic representation of all the notes, rests, staves, nested rhythms, beams, slurs and other notational elements in any score. Because Abjad wraps the powerful LilyPond music notation package, you can use Abjad to control extremely fine-grained typographic details of all elements of any score.
- **Frescobaldi** - is a LilyPond music score editor written in Python using PyQt4 and PyKDE4. Clicking a button runs LilyPond on the current document and displays the PDF in a preview window. There are some nice editing tools and a powerful score wizard to quickly setup a template score.
- **mingus** - mingus is an advanced music theory and notation package for Python. It can be used to play around with music theory, to build editors, educational tools and other applications that need to process music. It can also be used to create sheet music with LilyPond and do automated musicological analysis.
- **Kodou** - Kodou is a small package for algorithmic music notation which runs on top of the LilyPond compiler. It has been designed with the goal of being as minimalistic as possible, yet allowing creation of complex musical structures. It's public API consists of only two objects: a class Part (representing a musical line which itself could be mono- or polyphonic) and a main processing function kodou. Kodou's main strength however lies in it's treatment of time: rhythm is not constructed dependent on duration, but from points (notes/chords) alongside of a timeline and as such as a function of time. Setting variant durations other than the ones imposed by

Kodou is possible.

- see also 'music21' below

Musical Analysis

- **music21** - a toolkit developed at MIT for computational musicology, music theory, and generative composition. Provides expandable objects and methods for most common theoretical problems. Supports music import via MusicXML, Humdrum/Kern, Musedata, ABC, and MIDI, output via MusicXML, Lilypond, and MIDI, and can easily integrate with notation editors (Finale, Sibelius, or MuseScore) and other audio and DAW software (via MIDI).
- **pcsets** - Pitch Class Sets are a mathematical model for analyzing and composing music.
- **PyOracle** - Module for Audio Oracle and Factor Oracle Musical Analysis.

Audio Analysis

- **Friture** - Friture is a graphical program designed to do time-frequency analysis on audio input in real-time. It provides a set of visualization widgets to display audio data, such as a scope, a spectrum analyser, a rolling 2D spectrogram.
- **LibXtract** - LibXtract is a simple, portable, lightweight library of audio feature extraction functions. The purpose of the library is to provide a relatively exhaustive set of feature extraction primatives that are designed to be 'cascaded' to create a extraction hierarchies.
- **Yaafe** - Yet Another Audio Feature Extractor is a toolbox for audio analysis. Easy to use and efficient at extracting a large number of audio features simultaneously. WAV and MP3 files supported, or embedding in C++, Python or Matlab applications.
- **Aubio** - Aubio is a tool designed for the extraction of annotations from audio signals. Its features include segmenting a sound file before each of its attacks, performing pitch detection, tapping the beat and producing midi streams from live audio.
- **LibROSA** - A python module for audio and music analysis. It is easy to use, and implements many commonly used features for music analysis.

Ear Training

- **GNU Solfege** - GNU Solfege is a computer program written to help you practice ear training. It can be useful when practicing the simple and mechanical exercises.

cSound

- **athenaCL** - modular, polyphonic, poly-paradigm algorithmic music composition in an interactive command-line environment. The athenaCL system is an open-source, cross-platform, object-oriented composition tool written in Python; it can be scripted and embedded, includes integrated instrument libraries, post-tonal and microtonal pitch modeling tools, multiple-format graphical outputs, and musical output in Csound, MIDI, audio file, XML, and text formats.
- **Cabel** - Visual way to create csound instruments.
- **Dex Tracker** - Front end for csound that includes a tracker style score editor in a grid, text editor, cabel tested with Python 2.5.
- **Ounk** is a Python audio scripting environment that uses Csound as it's engine.
- **Cecilia** is a csound frontend that lets you create your own GUI (grapher, sliders, toggles, popup menus) using a simple syntax. Cecilia comes with a lots of original builtin modules for sound effects and synthesis. Previously written in tcl/tk, Cecilia was entirely rewritten with Python/wxPython and uses the Csound API for communicating between the interface and the audio engine. Version 4.02 beta is the current release.
- see also 'blue' below

Audio (Visual) Programming Frameworks

- **Peace Synthesizer Framework** - "Peace Synthesizer Framework" is Cross Platform Scriptable Real-Time Visualization & Sound. It has internal and external real-time scriptable visualization and sound generation and also support Nintendo system [Famicom] - like sound Emulation for 8-bits style chiptune music.
- **Hypersonic** - Hypersonic is for building and manipulating sound processing pipelines. It is designed for real-time control. It includes objects for oscillators, filters, file-io, soundcard and memory operations.

Music programming in Python

Playing & creating sound

- **pydub** - Pydub is a simple and easy high level interface based on ffmpeg and influenced by jquery. It manipulates audio, adding effects, id3 tags, slicing, concatenating audio tracks. Supports python 2.6, 2.7, 3.2, 3.3
- **audiere** - Audiere is a high-level audio API. It can play Ogg VorbisAU, MP3, FLACAS, uncompressed WAV, AIFF, MOD, S3M, XM, and ITAN files. For audio output, Audiere supports DirectSound or WinMM in Windows, OSS on Linux and Cygwin, and SGI AL on IRIX.
- **audiolab** - audiolab is a small Python package (now part of scikits) to

import data from audio files to numpy arrays and export data from numpy arrays to audio files. It uses libsndfile from Erik Castro de Lopo for the underlying IO, which supports many different audio formats: http://www.mega-nerd.com/libsndfile/

- **GStreamer Python Bindings** - GStreamer is a big multimedia library, it is very simple to use it with these python bindings. Many applications rely on it (Exaile, Pitivi, Jokosher, Listen usw.). Online documentation can be found on http://pygstdocs.berlios.de/
- **improviser** - Automatic music generation software. Experiments in musical content generation.
- **python-musical** - Python library for music theory, synthesis, and playback. Contains a collection of audio wave generators and filters powered by numpy. Also contains a pythonic music theory library for handling notes, chords, scales. Can load, save, and playback audio.
- **LoopJam** - Instant 1 click remixing of sample loops, able to boost your creativity and multiply your sample loop library. Remix audio loops on a slice level, apply up to 9 FX to individual slices or create countless versions using LJ's auto-remix feature (jam) which re-arranges the audio loop forming musical patterns.
- **Loris** - Loris is an Open Source C++ class library implementing analysis, manipulation, and synthesis of digitized sounds using the Reassigned Bandwidth-Enhanced Additive Sound Model. Loris supports modified resynthesis and manipulations of the model data, such as time- and frequency-scale modification and sound morphing. Loris includes support and wrapper code for building extension modules for various scripting languages (Python, Tcl, Perl).
- **MusicKit** - The MusicKit is an object-oriented software system for building music, sound, signal processing, and MIDI applications. It has been used in such diverse commercial applications as music sequencers, computer games, and document processors. Professors and students in academia have used the MusicKit in a host of areas, including music performance, scientific experiments, computer-aided instruction, and physical modeling. PyObjC is required to use this library in Python.
- **pyao** - pyao provides Python bindings for libao, a cross-platform audio output library. It supports audio output on Linux (OSS, ALSA, PulseAudio, esd), MacOS X, Windows, *BSD and some more. There are ready-to-use packages in Debian/Ubuntu, and Audio output is as easy as: `import ao; pcm = ao.AudioDevice("pulse"); pcm.play(data)`
- **pyAudio** - PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. Using PyAudio, you can easily use Python to play and record audio on a variety of platforms. Seems to be a successor of fastaudio, a once popular binding for PortAudio
- **pyFluidSynth** - Python bindings for FluidSynth, a MIDI synthesizer that

uses SoundFont instruments. This module contains Python bindings for FluidSynth. FluidSynth is a software synthesizer for generating music. It works like a MIDI synthesizer. You load patches, set parameters, then send NOTEON and NOTEOFF events to play notes. Instruments are defined in SoundFonts, generally files with the extension SF2. FluidSynth can either be used to play audio itself, or you can call a function that returns chunks of audio data and output the data to the soundcard yourself.

- **Pygame** - Pygame is a set of Python modules designed for writing games. It is written on top of the excellent SDL library. This allows you to create fully featured games and multimedia programs in the Python language. Pygame is highly portable and runs on nearly every platform and operating system. .ogg .wav .midi .mod .xm .mp3. Sound output. midi input and output. Load sounds into numeric and numpy arrays.
- **PyMedia** - (Not updated since 2006) PyMedia is a Python module for the multimedia purposes. It provides rich and simple interface for the digital media manipulation(wav, mp3, ogg, avi, divx, dvd, cdda etc). It includes parsing, demutiplexing, multiplexing, coding and decoding. It can be compiled for Windows, Linux and cygwin.
- **pyo** - pyo is a Python module containing classes for a wide variety of audio signal processing types. With pyo, user will be able to include signal processing chains directly in Python scripts or projects, and to manipulate them in real time through the interpreter. Tools in pyo module offer primitives, like mathematical operations on audio signal, basic signal processing (filters, delays, synthesis generators, etc.), but also complex algorithms to create sound granulation and others creative audio manipulations. pyo supports OSC protocol (Open Sound Control), to ease communications between softwares, and MIDI protocol, for generating sound events and controlling process parameters. pyo allows creation of sophisticated signal processing chains with all the benefits of a mature, and wildly used, general programming language.
- **Zyne** - Zyne is a Python modular synthesizer using pyo as its audio engine. Zyne comes with more than 10 builtin modules implementing different kind of synthesis engines and provides a simple API to create your own custom modules.
- **Soundgrain** - Soundgrain is a graphical interface where users can draw and edit trajectories to control granular sound synthesis modules. Soundgrain is written with Python and WxPython and uses pyo as its audio engine.
- **Pyper** - (Not updated since early 2005) Pyper is a musical development environment. It allows you to write Python scripts that generates music in real-time. Pyper uses QuickTime Musical Instruments for synthesis.
- **pySonic** - (Not updated since 2005) pySonic is a Python wrapper around the high performance, cross platform, but closed source, FMOD sound library. You get all the benefits of the FMOD library, but in a Pythonic, object

oriented package.

- **PySndObj** - The Sound Object Library is an object-oriented audio processing library. It provides objects for synthesis and processing of sound that can be used to build applications for computer-generated music. The core code, including soundfile and text input/output, is fully portable across several platforms. Platform-specific code includes realtime audio IO and MIDI input support for Linux (OSS,ALSA and Jack), Windows (MME and ASIO), MacOS X (CoreAudio, but no MIDI at moment), Silicon Graphics (Irix) machines and any Open Sound System-supported UNIX. The SndObj library also exists as Python module, aka PySndObj. The programming principles for Python SndObj programming are similar to the ones used in C++. It is also possible to use the Python interpreter for on-the-fly synthesis programming.
- **PySynth** - A simple music synthesizer.
- **Snack** - (last update: December 2005) The Snack Sound Toolkit is designed to be used with a scripting language such as Tcl/Tk or Python. Using Snack you can create powerful multi-platform audio applications with just a few lines of code. Snack has commands for basic sound handling, such as playback, recording, file and socket I/O. Snack also provides primitives for sound visualization, e.g. waveforms and spectrograms. It was developed mainly to handle digital recordings of speech (being developped at the KTH music&speech department), but is just as useful for general audio. Snack has also successfully been applied to other one-dimensional signals. The combination of Snack and a scripting language makes it possible to create sound tools and applications with a minimum of effort. This is due to the rapid development nature of scripting languages. As a bonus you get an application that is cross-platform from start. It is also easy to integrate Snack based applications with existing sound analysis software.
- **AudioLazy** - Real-Time Expressive Digital Signal Processing (DSP) Package for Python, using any Python iterable as a [-1;1] range audio source. Has time-variant linear filters as well as LTI filters using Z-Transform equations like `1 - z ** -1`, as well as analysis (ZCR / zero crossing rate, LPC / Linear Predictive Coding, AMDF, etc.), synthesis (table lookup, ADSR, etc.), ear modeling (Patterson-Holdsworth with gammatone filters and ERB models), and multiple implementation of common filters (lowpass, highpass, comb, resonator), among several other resources (e.g. Lagrange polynomial interpolation, simple converters among MIDI pitch / frequency / string). Works mainly with `Stream` instances for its signal outputs, a generator-like (lazy) iterable with elementwise/broadcast-style operators similar to the Numpy array operators. Integrated with Matplotlib for LTI filter plotting, although it doesn't require Matplotlib nor Numpy for computation, DSP or I/O. Emphasizes sample-based processing while keeping block-based processing easy to be done, this package can also be

seen as a highly enhanced itertools. Pure Python, multiplatform, compatible with Python 2.7 and 3.2+, uses PyAudio for audio I/O (if needed). Can be used together with Scipy, Sympy, music21 and several other packages, none required for DSP computation based on Python iterables.

- **sounddevice** - This module provides bindings for the PortAudio library (using CFFI) and a few convenience functions to play and record NumPy arrays containing audio signals.

Community

- **PythonSound** - The Python Sound Project aims to develop a productive community around Python, Csound and other synthesis engines as tools for algorithmic and computer assisted composition of electroacoustic music.

Csound

- **CSound / CsoundAC** - Csound is a sound and music synthesis system, providing facilities for composition and performance over a wide range of platforms and for any style of music. The Csound orchestra language features over 1200 unit generators (called "opcodes") covering nearly every sound synthesis method and that the user can combine into "instruments" of unlimited complexity and flexibility. Csound 5 allows Python code to be called from or directly embedded into Csound orchestras. Additionally, the csnd Python extension module wraps the Csound API so that Csound can be embedded into Python applications. CsoundAC (for "Csound Algorithmic Composition") is a GUI front end to Csound with Python scripting and a Python module providing tools for the algorithmic generation or manipulation of Csound scores. Csound on Sourceforge; Csound-Python and Csound (some brief tutorials on the OLPC Wiki)
- **Csound Routines** - set of routines to manipulate and convert csound files
- **PMask** - Python implementation of CMask, a stochastic event generator for Csound.

MP3 stuff and Metadata editors

- **eyed3** - eyeD3 is a Python module and program for processing ID3 tags. Information about mp3 files (i.e bit rate, sample frequency, play time, etc.) is also provided. The formats supported are ID3 v1.0/v1.1 and v2.3/v2.4.
- **mutagen** - Mutagen is a Python module to handle audio metadata. It supports ASF, FLAC, M4A, Monkey's Audio, MP3, Musepack, Ogg FLAC, Ogg Speex, Ogg Theora, Ogg Vorbis, True Audio, WavPack and OptimFROG audio files. All versions of ID3v2 are supported, and all standard ID3v2.4

frames are parsed. It can read Xing headers to accurately calculate the bitrate and length of MP3s. ID3 and APEv2 tags can be edited regardless of audio format. It can also manipulate Ogg streams on an individual packet/page level.

- **ID3.py** - This module allows one to read and manipulate so-called ID3 informational tags on MP3 files through an object-oriented Python interface.
- **id3reader.py** - Id3reader.py is a Python module that reads ID3 metadata tags in MP3 files. It can read ID3v1, ID3v2.2, ID3v2.3, or ID3v2.4 tags. It does not write tags at all.
- **mpgedit** - mpgedit is an MPEG 1 layer 1/2/3 (mp3), MPEG 2, and MPEG 2.5 audio file editor that is capable of processing both Constant Bit Rate (CBR) and Variable Bit Rate (VBR) encoded files. mpgedit can cut an input MPEG file into one or more output files, as well as join one or more input MPEG files into a single output file. Since no file decoding / encoding occurs during editing, there is no audio quality loss when editing with mpgedit. A Python development toolkit enables Python developers to utilize the core mpgedit API, providing access to mp3 file playback, editing and indexing functionality.
- **m3ute2** - m3ute2 is program for copying, moving, and otherwise organizing M3U playlists and directories. m3ute2 can also generate detailed reports about lists of files.
- **mmpython** - MMPython is a Media Meta Data retrieval framework. It retrieves metadata from mp3, ogg, avi, jpg, tiff and other file formats. Among others it thereby parses ID3v2, ID3v1, EXIF, IPTC and Vorbis data into an object oriented struture.
- **KaaMetadata** Sucessor of MMPython.
- **PyID3** - pyid3 is a pure Python library for reading and writing id3 tags (version 1.0, 1.1, 2.3, 2.4, readonly support for 2.2). What makes this better than all the others? Testing! This library has been tested against some 200+ MB of just tags.
- **beets** - music tag correction and cataloging tool. Consists of both a command-line interface for music manipulation and a library for building related tools. Can automatically correct tags using the MusicBrainz database.
- see also: PySonic for programmable MP3 playback

MIDI Mania

- **pygame.midi** - is a portmidi wrapper orginally based on the pyportmidi wrapper. Also pygame.music can play midi files. Can get input from midi devices and can output to midi devices. For osx, linux and windows. New with pygame 1.9.0. *python -m pygame.examples.midi --output*

- **pyMIDI** - Provides object oriented programmatic manipulation of MIDI streams. Using this framework, you can read MIDI files from disk, build new MIDI streams, process, or filter preexisting streams, and write your changes back to disk. If you install this package on a Linux platform with alsalib, you can take advantage of the ALSA kernel sequencer, which provides low latency scheduling and receiving of MIDI events. SWIG is required to compile the ALSA extension sequencer extension. Although OS-X and Windows provide similar sequencer facilities, the current version of the API does not yet support them. Some buggs are remaining in this package (for example when trying to delete a track), it has not been updated since 2006. This package is by Giles Hall. A sourceforge download.
- **midi.py** - (DEAD LINK) - Python MIDI classes: meaningful data structures that represent MIDI events and other objects. You can read MIDI files to create such objects, or generate a collection of objects and use them to write a MIDI file.
- **MIDI.py** - This module offers functions: concatenate_scores(), grep(), merge_scores(), mix_scores(), midi2opus(), midi2score(), opus2midi(), opus2score(), play_score(), score2midi(), score2opus(), score2stats(), score_type(), segment(), timeshift() and to_millisecs(). Uses Python3. There is a call-compatible Lua module.
- **PMIDI** - The PMIDI library allows the generation of short MIDI sequences in Python code.The interface allows a programmer to specify songs, instruments, measures, and notes. Playback is handled by the Windows MIDI stream API so proper playback timing is handled by the OS rather than by client code. The library is especially useful for generating earcons.
- **portmidizero** - portmidizero is a simple ctypes wrapper for PortMidi in pure Python.
- **PyChoReLib** - Python Chord Recognition Library. This is a library that implements the transformation from a list of notenames to a chord name. The system can be taught new chords by example: tell it that ['c', 'e', 'g'] is called a 'C' chord, and using its built-in music knowledge it immediately recognizes all major triads in all keys and all inversions/permutations. Comes with a real-time midi-input demo program (needs PyPortMidi).
- **PyMIDI** - The MIDI module provides MIDI input parsers for Python. Package not updated since 2000.
- **PyPortMidi** - PyPortMidi is a Python wrapper for PortMidi. PortMidi is a cross-platform C library for realtime MIDI control. Using PyPortMidi, you can send and receive MIDI data in realtime from Python. Besides using PyPortMidi to communicate to synthesizers and the like, it is possible to use PyPortMidi as a way to send MIDI messages between software packages on the same computer. PyPortMidi is now maintained at http://bitbucket.org/aalex/pyportmidi/
- **PythonMIDI** - The Python Midi package is a collection of classes handling Midi in and output in the Python programming language.

- **PySeq** - Python bindings for ALSA using ctypes
- **milk** - Superceding the older Nam, milk provides Python with classes representing key MIDI sequencer components: MIDI I/O, EventLists, Plugins and a realtime Flow class. The components can be freely interconnected in a fashion very similar to physical MIDI cabling, however the milk event system is not limited to MIDI events alone; you can define your own extensions should the need arise. Website says it is unpolished and unfinished.
- **pyrtmidi** - rtmidi provides realtime MIDI input/output across Linux (ALSA), Macintosh OS X, SGI, and Windows (Multimedia Library) operating systems. It is very fast, has a clean and pythonic interface, and supports virtual ports, according to author Patrick Kidd. In fact it is a wrapper for Gary Scavone's rtmidi from here, rather than the address on this website:
- **rtmidi-python** - Another RtMidi wrapper.
- **winmidi.pyd** - A demo? of a Python extension interfacing to the native windows midi libs that developed from earlier attempts.
- **win32midi** - Some Python samples to demonstrate how to output MIDI stream on MS windows platform. Unlike previous links, these samples playback MIDI by directly calling the Win32 MIDI APIs without an intermediate portable library. It provides a simple player class for playing with MIDI sound using the synthesizer on the soundcard/onboard soundchip. A sample script is provided for testing it out. As it is still a work in progress, bugs are expected.
- **midiutil** - A pure Python library for generating Midi files
- **Pyknon** - Pyknon is a simple music library for Python hackers. With Pyknon you can generate Midi files quickly and reason about musical proprieties.
- **Desfonema Sequencer** - A tracker minded MIDI sequencer for Linux (ALSA) written in Python/PyGTK
- **python-music-gen** - Simple library to generate midi patterns from numbers. Useful for building generative music tools.
- **fluidsynth-gui** - Graphical User Interface for FluidSynth, and an alternative to Qsynth.

Other protocols

- **OSC.py** - Python classes for OpenSoundControl library client functionality. The OSC homepage is at http://opensoundcontrol.org
- **Twisted-osc** - OSC Library for Twisted, an event-driven Python framework. It could really be ported to a non-Twisted framework as well, but is currently in the process of possibly become an official part of Twisted.
- **aiosc** - Minimalistic OSC communication module using asyncio.

- **pyalsaaudio** - This package contains wrappers for accessing the ALSA API (The Advanced Linux Sound Architecture (ALSA) provides audio and MIDI functionality to the Linux operating system) from Python. It is fairly complete for PCM devices and Mixer access.
- **pkaudio** - pkaudio is a collection of Python-based modules for midi input, osc communication with supercollider, and pyqt functionality.
- **PyJack** - This is a Python C extension module which provides an interface to the Jack Audio Server. It is possible to access the Jack graph to perform port connections/disconnections, monitor graph change events, and to perform realtime audio capture and playback using Numeric Python arrays. This is released under the GPL.

MAX/MSP & PureData

- **mxdublin** - mxdublin is an object oriented framework to generate events in pd and max. pd, short for Pure Data, a graphical Computer Music System written by Miller S. Puckette. mxdublin is a real time Python user environment working within pd/max. It is designed to put logic into a sequence of events. Python has been chosen has the interface language to build and run sequencing objects. Has a prerequisites, the users needs to know a minimal of Python and pd/max.
- **net.loadbang.jython** is a package which supports the Python scripting/programming language within MXJ for Max/MSP. We use the Jython interpreter, which allows Python and Java to interact, and gives Python access to the standard Java libraries (as well as any other Java code available to MXJ).
- **OpenExposition** - OpenExposition is a library aimed at automatic generation of user interfaces. The programmer only needs to specify what parts of the application need to be exposed to the user, and OpenExposition does the rest. At present, OpenExposition allows access to variables (either directly or through a pair of set/get methods), and class methods. It can construct the user interface graphically (using either the multi-platform FLTK library or Cocoa on Mac OS X), programatically (through Python), aurally (using the speech synthesis and recognition capabilities on Mac OS X), and by building MAX/MSP externals that can then be used in MAX/MSP.
- **Py/pyext** - Python script objects is an object library providing a full integration of the Python scripting language into the PD (and in the future Max/MSP) real-time system. With the py object you can load Python modules and execute the functions therein. With pyext you can use Python classes to represent full-featured pd/Max message objects. Multithreading (detached methods) is supported for both objects. You can send messages to named objects or receive (with pyext) with Python methods.
- **Purity** is a Python library for Pure Data dynamic patching. The idea is to

be able to harness the power of Pure Data for audio programming without having to use its graphical interface. Python's clear and intuitive syntax can be used with profit in order to create intricate patches with advanced string handling, graphical user interfaces and asynchronous network operations. Purity uses Twisted, an event-driven Python framework.

Music software supporting Python

Multitrack Studios

- **REAPER** - "Audio Production Without Limits": REAPER is a professional digital audio workstation (DAW) for Windows, OS X and WINE. It comes with an uncrippled evaluation licence and supports advanced audio and MIDI recording, arranging and mixing. The support of several plugin formats (like VST, DX and AU) as well as the extremely flexible routing capabilities make it a powerful production suite. Since version 3.12 REAPER is scriptable with Python, allowing access to internal actions and parts of the API.
- **Ableton Live** - Award-winning commercial music creation, production and performance platform for Mac OS and Windows. Live is far and away one of the most interesting and groundbreaking audio recording and sequencing tools to come along in the past five years. Live uses Python internally and an experimental API has been exposed at this site, and there is a discussion group here.
- **blue** - blue is a Java program for use with Csound. It's interface is much like a digital multitrack, but differs in that there timelines within timelines (polyObjects). This allows for a compositional organization in time that seems to me to be very intuitive, informative, and flexible. soundObjects are the building blocks within blue's score timeline. soundObjects can be lists of notes, algorithmic generators, Python script code, csound instrument definitions, and whatever plugins that are developed for blue. these soundObjects may be text based, but they can be completely GUI based as well.
- **Jokosher** - Jokosher is a simple yet powerful multi-track studio. With it you can create and record music, podcasts and more, all from an integrated simple environment. Jokosher is written in Python and uses the GNOME platform and the GTK widget set. The audio engine is powered by GStreamer, and we use Cairo for some of the graphics.
- **Jeskola Buzz Modular** - Buzz is a modular audio host that saw the beginning of it's development in 1997 leading the way in open plugin-format hosting (pre-VST) and a unique spin on modular routing using modules called Machines in the form of Generators, Effects, Control machines. Buzz's implementation of Python comes through the use of the Control plugin called PyBuzz, a fully customizable and assignable meta-editor created by Leonard Ritter (creator of the Lunar audio library and the

Linux modular host, Aldrin). A discussion group can be found here

- **PyDAW** - PyDAW is a powerful pattern-based DAW and plugin suite for producing electronic music. The UI is written entirely in Python/PyQt, and the audio engine in C.

PythonInMusic (last edited 2019-02-17 13:20:10 by AmirTeymuri)

- MoinMoin Powered
- Python Powered
- GPL licensed
- Valid HTML 4.01

Unable to edit the page? See the FrontPage for instructions.

```
7     54 data.py
8     23 LICENSE.txt
2     48 main.py
9    100 midi_to_statematrix.py
     26 midi_to_statematrix.pyc
3    403 model.py
10     61 multi_training.py
     21 multi_training.pyc
11     18 out_to_in_op.py
      0 __pycache__
1     57 readme.md
4      7 setup1.sh
 5    11 setup2.sh
  6   34 setup_optional.sh
12     72 visualize.py
    935 total
```

Biaxial Recurrent Neural Network for Music Composition

This code implements a recurrent neural network trained to generate classical music. The model, which uses LSTM layers and draws inspiration from convolutional neural networks, learns to predict which notes will be played at each time step of a musical piece.

You can read about its design and hear examples on [this blog post](http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/).

Requirements

This code is written in Python, and depends on having Theano and theano-lstm (which can be installed with pip) installed. The bare minimum you should need to do to get everything running, assuming you have Python, is

```
sudo pip install --upgrade theano
sudo pip install numpy scipy theano-lstm python-midi
```

In addition, the included setup scripts should set up the environment exactly as it was when I trained the network on an Amazon EC2 g2.2xlarge instance with an external EBS volume. Installing it with other setups will likely be slightly different.

Using it

First, you will need to obtain a large selection of midi music, preferably in 4/4 time, with notes correctly aligned to beats. These can be placed in a directory "music".

To use the model, you need to first create an instance of the Model class:

```python
import model
m = model.Model([300,300],[100,50], dropout=0.5)
```

where the numbers are the sizes of the hidden layers in the two parts of the network architecture. This will take a while, as th is is where Theano will compile its optimized functions.

Next, you need to load in the data:
```python
import multi_training
pcs = multi_training.loadPieces("music")
```

Then, after creating an "output" directory for trained samples, you can start training:
```python
multi_training.trainPiece(m, pcs, 10000)
```

This will train using 10000 batches of 10 eight-measure segments at a time, and output a sampled output and the learned paramete rs every 500 iterations.

Finally, you can generate a full composition after training is c omplete. The function `gen_adaptive` in main.py will generate a piece and also prevent long empty gaps by increasing note probab ilities if the network stops playing for too long.
```python
gen_adaptive(m,pcs,10,name="composition")
```

There are also mechanisms to observe the hidden activations and memory cells of the network, but these are still a work in progr ess at the moment.

Right now, there is no separate validation step, because my init ial goal was to produce interesting music, not to assess the acc uracy of this method. It does, however, print out the cost on th e training set after every 100 iterations during training.

If you want to save your model weights, you can do
```python
pickle.dump( m.learned_config, open( "path_to_weight_file.p", "w b" ) )
```
and if you want to load them, you can do
```python
m.learned_config = pickle.load(open( "path_to_weight_file.p", "r b" ) )
```
```
import cPickle as pickle
import gzip
import numpy
from midi_to_statematrix import *

import multi_training
import model

def gen_adaptive(m,pcs,times,keep_thoughts=False,name="final"):
```

```
        xIpt, xOpt = map(lambda x: numpy.array(x, dtype='int8'),
 multi_training.getPieceSegment(pcs))
        all_outputs = [xOpt[0]]
        if keep_thoughts:
                all_thoughts = []
        m.start_slow_walk(xIpt[0])
        cons = 1
        for time in range(multi_training.batch_len*times):
                resdata = m.slow_walk_fun( cons )
                nnotes = numpy.sum(resdata[-1][:,0])
                if nnotes < 2:
                        if cons > 1:
                                cons = 1
                        cons -= 0.02
                else:
                        cons += (1 - cons)*0.3
                all_outputs.append(resdata[-1])
                if keep_thoughts:
                        all_thoughts.append(resdata)
        noteStateMatrixToMidi(numpy.array(all_outputs),'output/'
+name)
        if keep_thoughts:
                pickle.dump(all_thoughts, open('output/'+name+'.
p','wb'))

def fetch_train_thoughts(m,pcs,batches,name="trainthoughts"):
        all_thoughts = []
        for i in range(batches):
                ipt, opt = multi_training.getPieceBatch(pcs)
                thoughts = m.update_thought_fun(ipt,opt)
                all_thoughts.append((ipt,opt,thoughts))
        pickle.dump(all_thoughts, open('output/'+name+'.p','wb')
)

if __name__ == '__main__':

        pcs = multi_training.loadPieces("music")

        m = model.Model([300,300],[100,50], dropout=0.5)

        multi_training.trainPiece(m, pcs, 10000)

        pickle.dump( m.learned_config, open( "output/final_learn
ed_config.p", "wb" ) )
import theano, theano.tensor as T
import numpy as np
import theano_lstm

from out_to_in_op import OutputFormToInputFormOp

from theano_lstm import Embedding, LSTM, RNN, StackedCells, Laye
r, create_optimization_updates, masked_loss, MultiDropout

def has_hidden(layer):
    """
    Whether a layer has a trainable
```

```
    initial hidden state.
    """
    return hasattr(layer, 'initial_hidden_state')

def matrixify(vector, n):
    # Cast n to int32 if necessary to prevent error on 32 bit sy
stems
    return T.repeat(T.shape_padleft(vector),
                    n if (theano.configdefaults.local_bitwidth()
 == 64) else T.cast(n,'int32'),
                    axis=0)

def initial_state(layer, dimensions = None):
    """
    Initalizes the recurrence relation with an initial hidden st
ate
    if needed, else replaces with a "None" to tell Theano that
    the network **will** return something, but it does not need
    to send it to the next step of the recurrence
    """
    if dimensions is None:
        return layer.initial_hidden_state if has_hidden(layer) e
lse None
    else:
        return matrixify(layer.initial_hidden_state, dimensions)
 if has_hidden(layer) else None

def initial_state_with_taps(layer, dimensions = None):
    """Optionally wrap tensor variable into a dict with taps=[-1
]"""
    state = initial_state(layer, dimensions)
    if state is not None:
        return dict(initial=state, taps=[-1])
    else:
        return None

class PassthroughLayer(Layer):
    """
    Empty "layer" used to get the final output of the LSTM
    """

    def __init__(self):
        self.is_recursive = False

    def create_variables(self):
        pass

    def activate(self, x):
        return x

    @property
    def params(self):
        return []

    @params.setter
    def params(self, param_list):
```

```
        pass


def get_last_layer(result):
    if isinstance(result, list):
        return result[-1]
    else:
        return result

def ensure_list(result):
    if isinstance(result, list):
        return result
    else:
        return [result]


class Model(object):

    def __init__(self, t_layer_sizes, p_layer_sizes, dropout=0):

        self.t_layer_sizes = t_layer_sizes
        self.p_layer_sizes = p_layer_sizes

        # From our architecture definition, size of the notewise
 input
        self.t_input_size = 80

        # time network maps from notewise input size to various
hidden sizes
        self.time_model = StackedCells( self.t_input_size, cellt
ype=LSTM, layers = t_layer_sizes)
        self.time_model.layers.append(PassthroughLayer())

        # pitch network takes last layer of time model and state
 of last note, moving upward
        # and eventually ends with a two-element sigmoid layer
        p_input_size = t_layer_sizes[-1] + 2
        self.pitch_model = StackedCells( p_input_size, celltype=
LSTM, layers = p_layer_sizes)
        self.pitch_model.layers.append(Layer(p_layer_sizes[-1],
2, activation = T.nnet.sigmoid))

        self.dropout = dropout

        self.conservativity = T.fscalar()
        self.srng = T.shared_randomstreams.RandomStreams(np.rand
om.randint(0, 1024))

        self.setup_train()
        self.setup_predict()
        self.setup_slow_walk()

    @property
    def params(self):
        return self.time_model.params + self.pitch_model.params
```

```
    @params.setter
    def params(self, param_list):
        ntimeparams = len(self.time_model.params)
        self.time_model.params = param_list[:ntimeparams]
        self.pitch_model.params = param_list[ntimeparams:]

    @property
    def learned_config(self):
        return [self.time_model.params, self.pitch_model.params,
 [l.initial_hidden_state for mod in (self.time_model, self.pitch
_model) for l in mod.layers if has_hidden(l)]]

    @learned_config.setter
    def learned_config(self, learned_list):
        self.time_model.params = learned_list[0]
        self.pitch_model.params = learned_list[1]
        for l, val in zip((l for mod in (self.time_model, self.p
itch_model) for l in mod.layers if has_hidden(l)), learned_list[
2]):
            l.initial_hidden_state.set_value(val.get_value())

    def setup_train(self):

        # dimensions: (batch, time, notes, input_data) with inpu
t_data as in architecture
        self.input_mat = T.btensor4()
        # dimensions: (batch, time, notes, onOrArtic) with 0:on,
 1:artic
        self.output_mat = T.btensor4()

        self.epsilon = np.spacing(np.float32(1.0))

        def step_time(in_data, *other):
            other = list(other)
            split = -len(self.t_layer_sizes) if self.dropout els
e len(other)
            hiddens = other[:split]
            masks = [None] + other[split:] if self.dropout else
[]
            new_states = self.time_model.forward(in_data, prev_h
iddens=hiddens, dropout=masks)
            return new_states

        def step_note(in_data, *other):
            other = list(other)
            split = -len(self.p_layer_sizes) if self.dropout els
e len(other)
            hiddens = other[:split]
            masks = [None] + other[split:] if self.dropout else
[]
            new_states = self.pitch_model.forward(in_data, prev_
hiddens=hiddens, dropout=masks)
            return new_states

        # We generate an output for each input, so it doesn't ma
ke sense to use the last output as an input.
```

```
        # Note that we assume the sentinel start value is alread
y present
        # TEMP CHANGE: NO SENTINEL
        input_slice = self.input_mat[:,0:-1]
        n_batch, n_time, n_note, n_ipn = input_slice.shape

        # time_inputs is a matrix (time, batch/note, input_per_n
ote)
        time_inputs = input_slice.transpose((1,0,2,3)).reshape((
n_time,n_batch*n_note,n_ipn))
        num_time_parallel = time_inputs.shape[1]

        # apply dropout
        if self.dropout > 0:
            time_masks = theano_lstm.MultiDropout( [(num_time_pa
rallel, shape) for shape in self.t_layer_sizes], self.dropout)
        else:
            time_masks = []

        time_outputs_info = [initial_state_with_taps(layer, num_
time_parallel) for layer in self.time_model.layers]
        time_result, _ = theano.scan(fn=step_time, sequences=[ti
me_inputs], non_sequences=time_masks, outputs_info=time_outputs_
info)

        self.time_thoughts = time_result

        # Now time_result is a list of matrix [layer](time, batc
h/note, hidden_states) for each layer but we only care about
        # the hidden state of the last layer.
        # Transpose to be (note, batch/time, hidden_states)
        last_layer = get_last_layer(time_result)
        n_hidden = last_layer.shape[2]
        time_final = get_last_layer(time_result).reshape((n_time
,n_batch,n_note,n_hidden)).transpose((2,1,0,3)).reshape((n_note,
n_batch*n_time,n_hidden))

        # note_choices_inputs represents the last chosen note. S
tarts with [0,0], doesn't include last note.
        # In (note, batch/time, 2) format
        # Shape of start is thus (1, N, 2), concatenated with al
l but last element of output_mat transformed to (x, N, 2)
        start_note_values = T.alloc(np.array(0,dtype=np.int8), 1
, time_final.shape[1], 2 )
        correct_choices = self.output_mat[:,1:,0:-1,:].transpose
((2,0,1,3)).reshape((n_note-1,n_batch*n_time,2))
        note_choices_inputs = T.concatenate([start_note_values,
correct_choices], axis=0)

        # Together, this and the output from the last LSTM goes
to the new LSTM, but rotated, so that the batches in
        # one direction are the steps in the other, and vice ver
sa.
        note_inputs = T.concatenate( [time_final, note_choices_i
nputs], axis=2 )
        num_timebatch = note_inputs.shape[1]
```

```
        # apply dropout
        if self.dropout > 0:
            pitch_masks = theano_lstm.MultiDropout( [(num_timeba
tch, shape) for shape in self.p_layer_sizes], self.dropout)
        else:
            pitch_masks = []

        note_outputs_info = [initial_state_with_taps(layer, num_
timebatch) for layer in self.pitch_model.layers]
        note_result, _ = theano.scan(fn=step_note, sequences=[no
te_inputs], non_sequences=pitch_masks, outputs_info=note_outputs
_info)

        self.note_thoughts = note_result

        # Now note_result is a list of matrix [layer](note, batc
h/time, onOrArticProb) for each layer but we only care about
        # the hidden state of the last layer.
        # Transpose to be (batch, time, note, onOrArticProb)
        note_final = get_last_layer(note_result).reshape((n_note
,n_batch,n_time,2)).transpose(1,2,0,3)

        # The cost of the entire procedure is the negative log l
ikelihood of the events all happening.
        # For the purposes of training, if the ouputted probabil
ity is P, then the likelihood of seeing a 1 is P, and
        # the likelihood of seeing 0 is (1-P). So the likelihood
 is (1-P)(1-x) + Px = 2Px - P - x + 1
        # Since they are all binary decisions, and are all proba
bilities given all previous decisions, we can just
        # multiply the likelihoods, or, since we are logging the
m, add the logs.

        # Note that we mask out the articulations for those note
s that aren't played, because it doesn't matter
        # whether or not those are articulated.
        # The padright is there because self.output_mat[:,:,:,0]
 -> 3D matrix with (b,x,y), but we need 3d tensor with
        # (b,x,y,1) instead
        active_notes = T.shape_padright(self.output_mat[:,1:,:,0
])
        mask = T.concatenate([T.ones_like(active_notes),active_n
otes], axis=3)

        loglikelihoods = mask * T.log( 2*note_final*self.output_
mat[:,1:] - note_final - self.output_mat[:,1:] + 1 + self.epsilo
n )
        self.cost = T.neg(T.sum(loglikelihoods))

        updates, _, _, _, _ = create_optimization_updates(self.c
ost, self.params, method="adadelta")
        self.update_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs=self.cost,
            updates=updates,
```

```
            allow_input_downcast=True)

        self.update_thought_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs= ensure_list(self.time_thoughts) + ensure_li
st(self.note_thoughts) + [self.cost],
            allow_input_downcast=True)

    def _predict_step_note(self, in_data_from_time, *states):
        # States is [ *hiddens, last_note_choice ]
        hiddens = list(states[:-1])
        in_data_from_prev = states[-1]
        in_data = T.concatenate([in_data_from_time, in_data_from
_prev])

        # correct for dropout
        if self.dropout > 0:
            masks = [1 - self.dropout for layer in self.pitch_mo
del.layers]
            masks[0] = None
        else:
            masks = []

        new_states = self.pitch_model.forward(in_data, prev_hidd
ens=hiddens, dropout=masks)

        # Now new_states is a per-layer set of activations.
        probabilities = get_last_layer(new_states)

        # Thus, probabilities is a vector of two probabilities,
P(play), and P(artic | play)

        shouldPlay = self.srng.uniform() < (probabilities[0] **
self.conservativity)
        shouldArtic = shouldPlay * (self.srng.uniform() < probab
ilities[1])

        chosen = T.cast(T.stack(shouldPlay, shouldArtic), "int8"
)

        return ensure_list(new_states) + [chosen]

    def setup_predict(self):
        # In prediction mode, note steps are contained in the ti
me steps. So the passing gets a little bit hairy.

        self.predict_seed = T.bmatrix()
        self.steps_to_simulate = T.iscalar()

        def step_time(*states):
            # States is [ *hiddens, prev_result, time]
            hiddens = list(states[:-2])
            in_data = states[-2]
            time = states[-1]

            # correct for dropout
```

```
            if self.dropout > 0:
                masks = [1 - self.dropout for layer in self.time
_model.layers]
                masks[0] = None
            else:
                masks = []

            new_states = self.time_model.forward(in_data, prev_h
iddens=hiddens, dropout=masks)

            # Now new_states is a list of matrix [layer](notes,
hidden_states) for each layer
            time_final = get_last_layer(new_states)

            start_note_values = theano.tensor.alloc(np.array(0,d
type=np.int8), 2)

            # This gets a little bit complicated. In the trainin
g case, we can pass in a combination of the
            # time net's activations with the known choices. But
 in the prediction case, those choices don't
            # exist yet. So instead of iterating over the combin
ation, we iterate over only the activations,
            # and then combine in the previous outputs in the st
ep. And then since we are passing outputs to
            # previous inputs, we need an additional outputs_inf
o for the initial "previous" output of zero.
            note_outputs_info = ([ initial_state_with_taps(layer
) for layer in self.pitch_model.layers ] +
                                 [ dict(initial=start_note_value
s, taps=[-1]) ])

            notes_result, updates = theano.scan(fn=self._predict
_step_note, sequences=[time_final], outputs_info=note_outputs_in
fo)

            # Now notes_result is a list of matrix [layer/output
](notes, onOrArtic)
            output = get_last_layer(notes_result)

            next_input = OutputFormToInputFormOp()(output, time
+ 1) # TODO: Fix time
            #next_input = T.cast(T.alloc(0, 3, 4),'int64')

            return (ensure_list(new_states) + [ next_input, time
 + 1, output ]), updates

        # start_sentinel = startSentinel()
        num_notes = self.predict_seed.shape[0]

        time_outputs_info = ([ initial_state_with_taps(layer, nu
m_notes) for layer in self.time_model.layers ] +
                             [ dict(initial=self.predict_seed, t
aps=[-1]),
                               dict(initial=0, taps=[-1]),
                               None ])
```

```
        time_result, updates = theano.scan( fn=step_time,
                                            outputs_info=time_ou
tputs_info,
                                            n_steps=self.steps_t
o_simulate )

        self.predict_thoughts = time_result

        self.predicted_output = time_result[-1]

        self.predict_fun = theano.function(
            inputs=[self.steps_to_simulate, self.conservativity,
 self.predict_seed],
            outputs=self.predicted_output,
            updates=updates,
            allow_input_downcast=True)

        self.predict_thought_fun = theano.function(
            inputs=[self.steps_to_simulate, self.conservativity,
 self.predict_seed],
            outputs=ensure_list(self.predict_thoughts),
            updates=updates,
            allow_input_downcast=True)

    def setup_slow_walk(self):

        self.walk_input = theano.shared(np.ones((2,2), dtype='in
t8'))
        self.walk_time = theano.shared(np.array(0, dtype='int64'
))
        self.walk_hiddens = [theano.shared(np.ones((2,2), dtype=
theano.config.floatX)) for layer in self.time_model.layers if ha
s_hidden(layer)]

        # correct for dropout
        if self.dropout > 0:
            masks = [1 - self.dropout for layer in self.time_mod
el.layers]
            masks[0] = None
        else:
            masks = []

        new_states = self.time_model.forward(self.walk_input, pr
ev_hiddens=self.walk_hiddens, dropout=masks)

        # Now new_states is a list of matrix [layer](notes, hidd
en_states) for each layer
        time_final = get_last_layer(new_states)

        start_note_values = theano.tensor.alloc(np.array(0,dtype
=np.int8), 2)
        note_outputs_info = ([ initial_state_with_taps(layer) fo
r layer in self.pitch_model.layers ] +
                             [ dict(initial=start_note_values, t
aps=[-1]) ])
```

```
        notes_result, updates = theano.scan(fn=self._predict_ste
p_note, sequences=[time_final], outputs_info=note_outputs_info)

        # Now notes_result is a list of matrix [layer/output](no
tes, onOrArtic)
        output = get_last_layer(notes_result)

        next_input = OutputFormToInputFormOp()(output, self.walk
_time + 1) # TODO: Fix time
        #next_input = T.cast(T.alloc(0, 3, 4),'int64')

        slow_walk_results = (new_states[:-1] + notes_result[:-1]
 + [ next_input, output ])

        updates.update({
                self.walk_time: self.walk_time+1,
                self.walk_input: next_input
            })

        updates.update({hidden:newstate for hidden, newstate, la
yer in zip(self.walk_hiddens, new_states, self.time_model.layers
) if has_hidden(layer)})

        self.slow_walk_fun = theano.function(
            inputs=[self.conservativity],
            outputs=slow_walk_results,
            updates=updates,
            allow_input_downcast=True)

    def start_slow_walk(self, seed):
        seed = np.array(seed)
        num_notes = seed.shape[0]

        self.walk_time.set_value(0)
        self.walk_input.set_value(seed)
        for layer, hidden in zip((l for l in self.time_model.lay
ers if has_hidden(l)),self.walk_hiddens):
            hidden.set_value(np.repeat(np.reshape(layer.initial_
hidden_state.get_value(), (1,-1)), num_notes, axis=0))
```

```
sudo apt-get install -y gcc g++ gfortran build-essential git wge
```

```
t linux-image-generic libopenblas-dev python-dev python-pip pyth
on-nose python-numpy python-scipy
sudo pip install --upgrade --no-deps git+git://github.com/Theano
/Theano.git
sudo wget http://developer.download.nvidia.com/compute/cuda/repo
s/ubuntu1404/x86_64/cuda-repo-ubuntu1404_7.0-28_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1404_7.0-28_amd64.deb
sudo apt-get update
sudo apt-get install -y cuda
echo -e "\nexport PATH=/usr/local/cuda/bin:$PATH\n\nexport LD_LI
BRARY_PATH=/usr/local/cuda/lib64" >> .bashrc
sudo rebootsudo apt-get update
sudo apt-get -y dist-upgrade

cuda-install-samples-7.0.sh ~/
cd NVIDIA_CUDA-7.0_Samples/
cd 1_Utilities/deviceQuery
make
./deviceQuery

echo -e "\n[global]\nfloatX=float32\ndevice=gpu\nbase_compiledir
=~/external/.theano/\nallow_gc=False\nwarn_float64=warn\n[mode]=
FAST_RUN\n\n[nvcc]\nfastmath=True\n\n[cuda]\nroot=/usr/local/cud
a\n" >> ~/.theanorc
sudo pip install theano-lstm python-midi

sudo apt-get install htop reptyr

cd /usr/bin/
sudo wget https://raw.githubusercontent.com/aurora/rmate/master/
rmate
sudo chmod 775 rmate

# Mount new EBS volume (at sdf -> xvdf)
sudo fdisk /dev/xvdf
# Parameters:
# n
# p
# 1
#
#
# t
# 1
# 83
# w
sudo mkfs.ext3 -b 4096 /dev/xvdf

cd ~
mkdir external
sudo mount -t ext3 /dev/xvdf external/
sudo chmod 755 external
sudo chown ubuntu external
sudo chgrp ubuntu external
cd external
mkdir neural_music # and then get all files. Or maybe use git?
```

```
sudo dd if=/dev/zero of=~/external/swapfile1  bs=1024 count=4194
304
sudo chown root:root ~/external/swapfile1
sudo chmod 0600 ~/external/swapfile1
sudo mkswap ~/external/swapfile1
sudo swapon ~/external/swapfile1import itertools
from midi_to_statematrix import upperBound, lowerBound

def startSentinel():
    def noteSentinel(note):
        position = note
        part_position = [position]

        pitchclass = (note + lowerBound) % 12
        part_pitchclass = [int(i == pitchclass) for i in range(1
2)]

        return part_position + part_pitchclass + [0]*66 + [1]
    return [noteSentinel(note) for note in range(upperBound-lowe
rBound)]

def getOrDefault(l, i, d):
    try:
        return l[i]
    except IndexError:
        return d

def buildContext(state):
    context = [0]*12
    for note, notestate in enumerate(state):
        if notestate[0] == 1:
            pitchclass = (note + lowerBound) % 12
            context[pitchclass] += 1
    return context

def buildBeat(time):
    return [2*x-1 for x in [time%2, (time//2)%2, (time//4)%2, (t
ime//8)%2]]

def noteInputForm(note, state, context, beat):
    position = note
    part_position = [position]

    pitchclass = (note + lowerBound) % 12
    part_pitchclass = [int(i == pitchclass) for i in range(12)]
    # Concatenate the note states for the previous vicinity
    part_prev_vicinity = list(itertools.chain.from_iterable((get
OrDefault(state, note+i, [0,0]) for i in range(-12, 13))))

    part_context = context[pitchclass:] + context[:pitchclass]

    return part_position + part_pitchclass + part_prev_vicinity
+ part_context + beat + [0]

def noteStateSingleToInputForm(state,time):
    beat = buildBeat(time)
```

```
    context = buildContext(state)
    return [noteInputForm(note, state, context, beat) for note i
n range(len(state))]

def noteStateMatrixToInputForm(statematrix):
    # NOTE: May have to transpose this or transform it in some w
ay to make Theano like it
    #[startSentinel()] +
    inputform = [ noteStateSingleToInputForm(state,time) for tim
e,state in enumerate(statematrix) ]
    return inputform
```



```
import midi, numpy

lowerBound = 24
upperBound = 102

def midiToNoteStateMatrix(midifile):
```

```
    pattern = midi.read_midifile(midifile)

    timeleft = [track[0].tick for track in pattern]

    posns = [0 for track in pattern]

    statematrix = []
    span = upperBound-lowerBound
    time = 0

    state = [[0,0] for x in range(span)]
    statematrix.append(state)
    while True:
        if time % (pattern.resolution / 4) == (pattern.resolutio
n / 8):
            # Crossed a note boundary. Create a new state, defau
lting to holding notes
            oldstate = state
            state = [[oldstate[x][0],0] for x in range(span)]
            statematrix.append(state)

        for i in range(len(timeleft)):
            while timeleft[i] == 0:
                track = pattern[i]
                pos = posns[i]

                evt = track[pos]
                if isinstance(evt, midi.NoteEvent):
                    if (evt.pitch < lowerBound) or (evt.pitch >=
 upperBound):
                        pass
                        # print "Note {} at time {} out of bound
s (ignoring)".format(evt.pitch, time)
                    else:
                        if isinstance(evt, midi.NoteOffEvent) or
 evt.velocity == 0:
                            state[evt.pitch-lowerBound] = [0, 0]
                        else:
                            state[evt.pitch-lowerBound] = [1, 1]
                elif isinstance(evt, midi.TimeSignatureEvent):
                    if evt.numerator not in (2, 4):
                        # We don't want to worry about non-4 tim
e signatures. Bail early!
                        # print "Found time signature event {}.
Bailing!".format(evt)
                        return statematrix

                try:
                    timeleft[i] = track[pos + 1].tick
                    posns[i] += 1
                except IndexError:
                    timeleft[i] = None

            if timeleft[i] is not None:
                timeleft[i] -= 1
```

```
        if all(t is None for t in timeleft):
            break

        time += 1

    return statematrix

def noteStateMatrixToMidi(statematrix, name="example"):
    statematrix = numpy.asarray(statematrix)
    pattern = midi.Pattern()
    track = midi.Track()
    pattern.append(track)

    span = upperBound-lowerBound
    tickscale = 55

    lastcmdtime = 0
    prevstate = [[0,0] for x in range(span)]
    for time, state in enumerate(statematrix + [prevstate[:]]):

        offNotes = []
        onNotes = []
        for i in range(span):
            n = state[i]
            p = prevstate[i]
            if p[0] == 1:
                if n[0] == 0:
                    offNotes.append(i)
                elif n[1] == 1:
                    offNotes.append(i)
                    onNotes.append(i)
            elif n[0] == 1:
                onNotes.append(i)
        for note in offNotes:
            track.append(midi.NoteOffEvent(tick=(time-lastcmdtim
e)*tickscale, pitch=note+lowerBound))
            lastcmdtime = time
        for note in onNotes:
            track.append(midi.NoteOnEvent(tick=(time-lastcmdtime
)*tickscale, velocity=40, pitch=note+lowerBound))
            lastcmdtime = time

        prevstate = state

    eot = midi.EndOfTrackEvent(tick=1)
    track.append(eot)

    midi.write_midifile("{}.mid".format(name), pattern)import os
, random
from midi_to_statematrix import *
from data import *
import pickle
#import cPickle as pickle

import signal
```

```
batch_width = 10 # number of sequences in a batch
batch_len = 16*8 # length of each sequence
division_len = 16 # interval between possible start locations

def loadPieces(dirpath):

    pieces = {}

    for fname in os.listdir(dirpath):
        if fname[-4:] not in ('.mid','.MID'):
            continue

        name = fname[:-4]

        outMatrix = midiToNoteStateMatrix(os.path.join(dirpath,
fname))
        if len(outMatrix) < batch_len:
            continue

        pieces[name] = outMatrix
        print ("Loaded {}".format(name))

    return pieces

def getPieceSegment(pieces):
    piece_output = random.choice(pieces.values())
    start = random.randrange(0,len(piece_output)-batch_len,divis
ion_len)
    # print "Range is {} {} {} -> {}".format(0,len(piece_output)
-batch_len,division_len, start)

    seg_out = piece_output[start:start+batch_len]
    seg_in = noteStateMatrixToInputForm(seg_out)

    return seg_in, seg_out

def getPieceBatch(pieces):
    i,o = zip(*[getPieceSegment(pieces) for _ in range(batch_wid
th)])
    return numpy.array(i), numpy.array(o)

def trainPiece(model,pieces,epochs,start=0):
    stopflag = [False]
    def signal_handler(signame, sf):
        stopflag[0] = True
    old_handler = signal.signal(signal.SIGINT, signal_handler)
    for i in range(start,start+epochs):
        if stopflag[0]:
            break
        error = model.update_fun(*getPieceBatch(pieces))
        if i % 100 == 0:
            print ("epoch {}, error={}".format(i,error))
        if i % 500 == 0 or (i % 100 == 0 and i < 1000):
            xIpt, xOpt = map(numpy.array, getPieceSegment(pieces
))
            noteStateMatrixToMidi(numpy.concatenate((numpy.expan
```

```
d_dims(xOpt[0], 0), model.predict_fun(batch_len, 1, xIpt[0])), a
xis=0),'output/sample{}'.format(i))
            pickle.dump(model.learned_config,open('output/params
{}.p'.format(i), 'wb'))
    signal.signal(signal.SIGINT, old_handler)
import theano, theano.tensor as T
import numpy as np

from data import noteStateSingleToInputForm


class OutputFormToInputFormOp(theano.Op):
    # Properties attribute
    __props__ = ()

    def make_node(self, state, time):
        state = T.as_tensor_variable(state)
        time = T.as_tensor_variable(time)
        return theano.Apply(self, [state, time], [T.bmatrix()])

    # Python implementation:
    def perform(self, node, inputs_storage, output_storage):
        state, time = inputs_storage
        output_storage[0][0] = np.array(noteStateSingleToInputFo
rm(state, time), dtype='int8')import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def actToColor(memcell, activation):
        return [0, sigmoid(activation), sigmoid(memcell)]

def internalMatrixToImgArray(inmat):
        return np.array(
                [[actToColor(m,a) for m,a in zip(row[:len(row)/2
],row[len(row)/2:])]
                        for row in inmat])

def probAndSuccessToImgArray(prob, succ, idx):
        return np.array([[[pr[idx]]*3,[sr[idx],0,0]] for pr, sr
in zip(prob, succ)])

def thoughtsToImageArray(thoughts):

        spacer = np.zeros((thoughts[0].shape[0], 5, 3))

        sequence = [
                        spacer,
                        probAndSuccessToImgArray(thoughts[4],tho
ughts[6], 0),
                        spacer,
                        probAndSuccessToImgArray(thoughts[4],tho
ughts[6], 1)
        ]
```

```
        for thought in thoughts[:-3]:
                sequence = [ spacer, internalMatrixToImgArray(th
ought) ] + sequence

        return (np.concatenate(sequence, axis=1 )*255).astype('u
int8')

def pastColor(prob, succ):
        return [prob[0], succ[0], succ[1]*succ[0]]

def drawPast(probs, succs):
        return np.array([
                [
                        pastColor(probs[time][note_idx], succs[t
ime][note_idx])
                        for time in range(len(probs))
                ]
                for note_idx in range(len(probs[0]))
        ])

def thoughtsAndPastToStackedArray(thoughts, probs, succs, len_pa
st):

        vert_spacer = np.zeros((thoughts[0].shape[0], 5, 3))

        past_out = drawPast(probs, succs)

        if len(probs) < len_past:
                past_out = np.pad(past_out, ((0,0),(len_past-len
(probs),0),(0,0)), mode='constant')

        def add_cur(ipt):
                return np.concatenate((
                        ipt,
                        vert_spacer,
                        probAndSuccessToImgArray(thoughts[-3],th
oughts[-1], 0),
                        vert_spacer,
                        probAndSuccessToImgArray(thoughts[-3],th
oughts[-1], 1)), axis=1)

        horiz_spacer = np.zeros((5, 1, 3))

        rows = [add_cur(past_out[-len_past:])]

        for thought in thoughts[:-3]:
                rows += [ horiz_spacer, add_cur(internalMatrixTo
ImgArray(thought)) ]

        maxlen = max([x.shape[1] for x in rows])
        rows = [np.pad(row, ((0,0),(maxlen-row.shape[1],0),(0,0)
), mode='constant') for row in rows]

        return (np.concatenate(rows, axis=0 )*255).astype('uint8
')
```

Top 5 Audio Analysis Library for Python : Must for Data Scientist

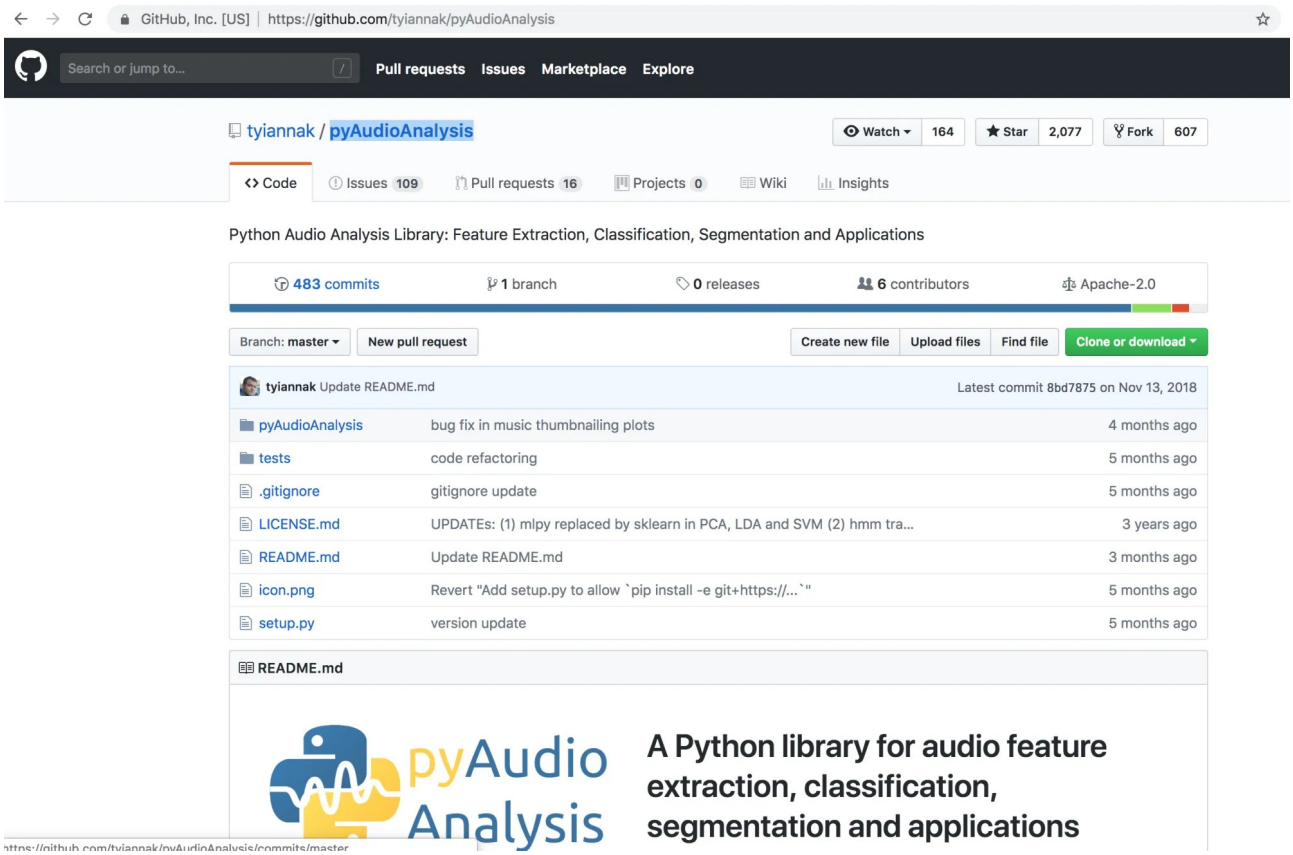
Reading Time: 3 minutes

As a Data Scientist you never know the upcoming stuffs . Right ? The amazing thing of this profession is that you may have to deal with different kind of data formats . Some time it could be text , images or Audio . Yes It could be an audio as well . As a Data Scientist I did not found so many articles on Audio analysis and process library in python . I have documented all my findings this article . This article **” Top 5 Audio Analysis Library for Python : Must for Data Scientist ”** will brief you on this topic . Lets start –

Audio Analysis Library for Python-

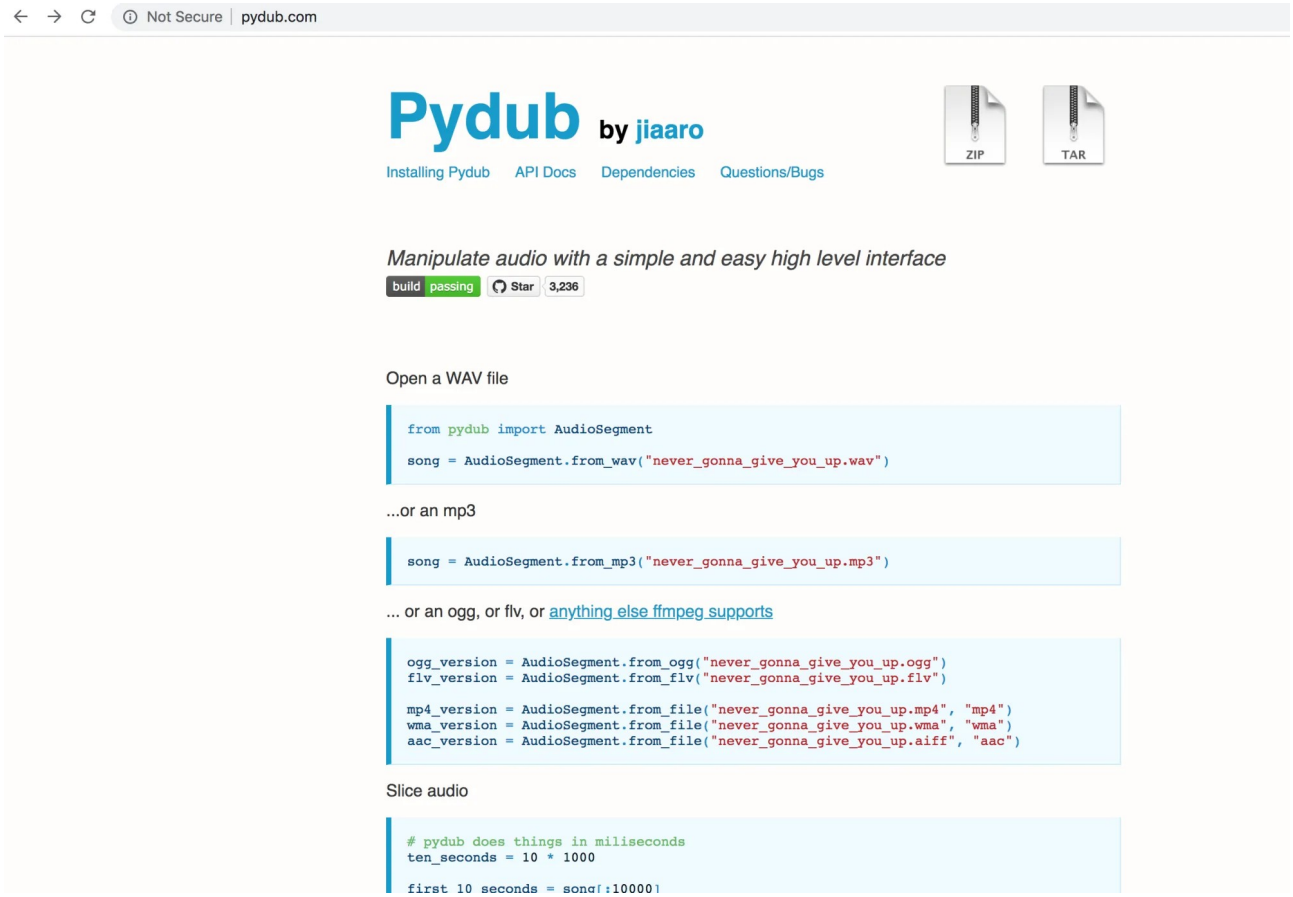
1.PyAudioAnalysis –

This Python module is really good in Audio Processing stuffs like classification . It supports feature engineering operations for supervised and unsupervised learning stuffs .



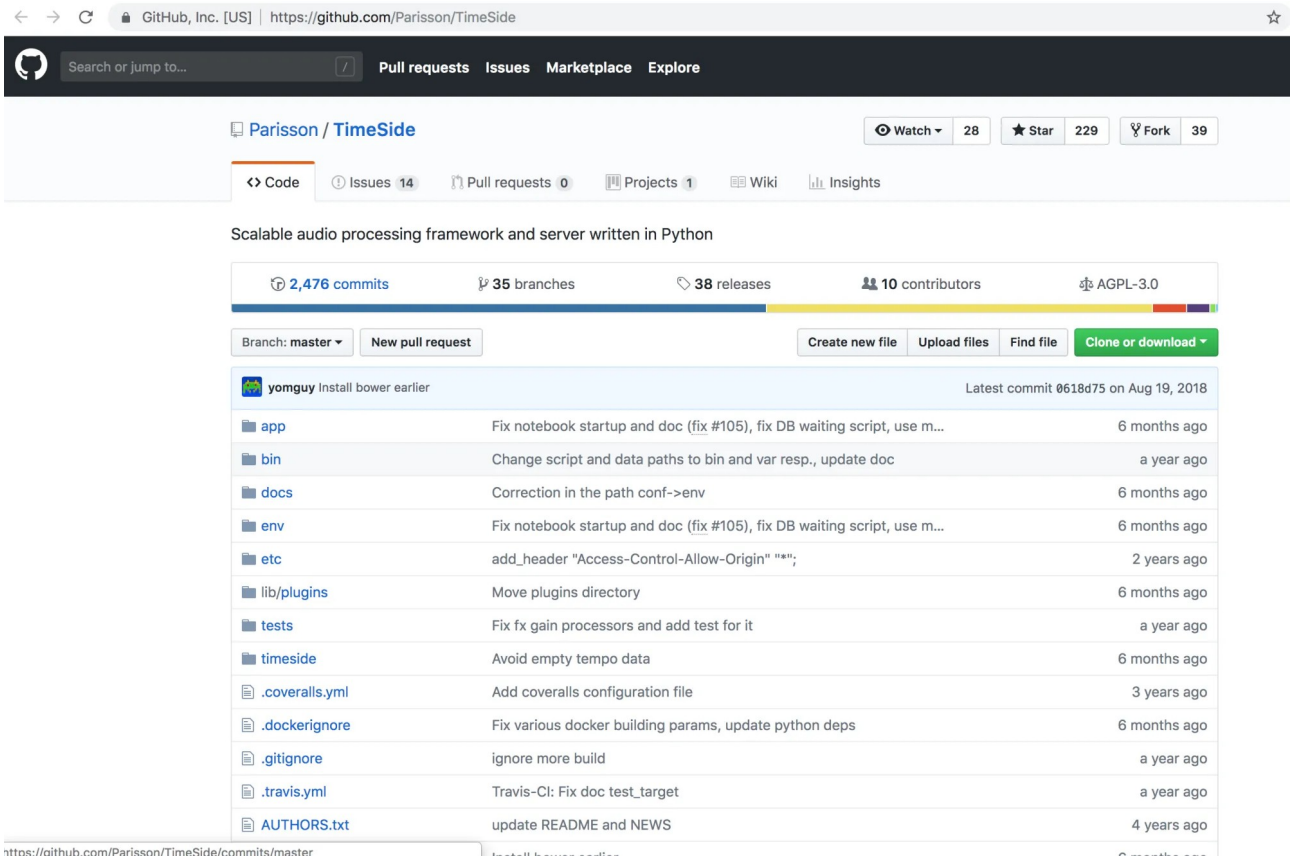
2. Pydub –

It helps to perform various common task in sound processing with python . For example -slicing the sound , concatenating the sound etc .I think you should check it out .



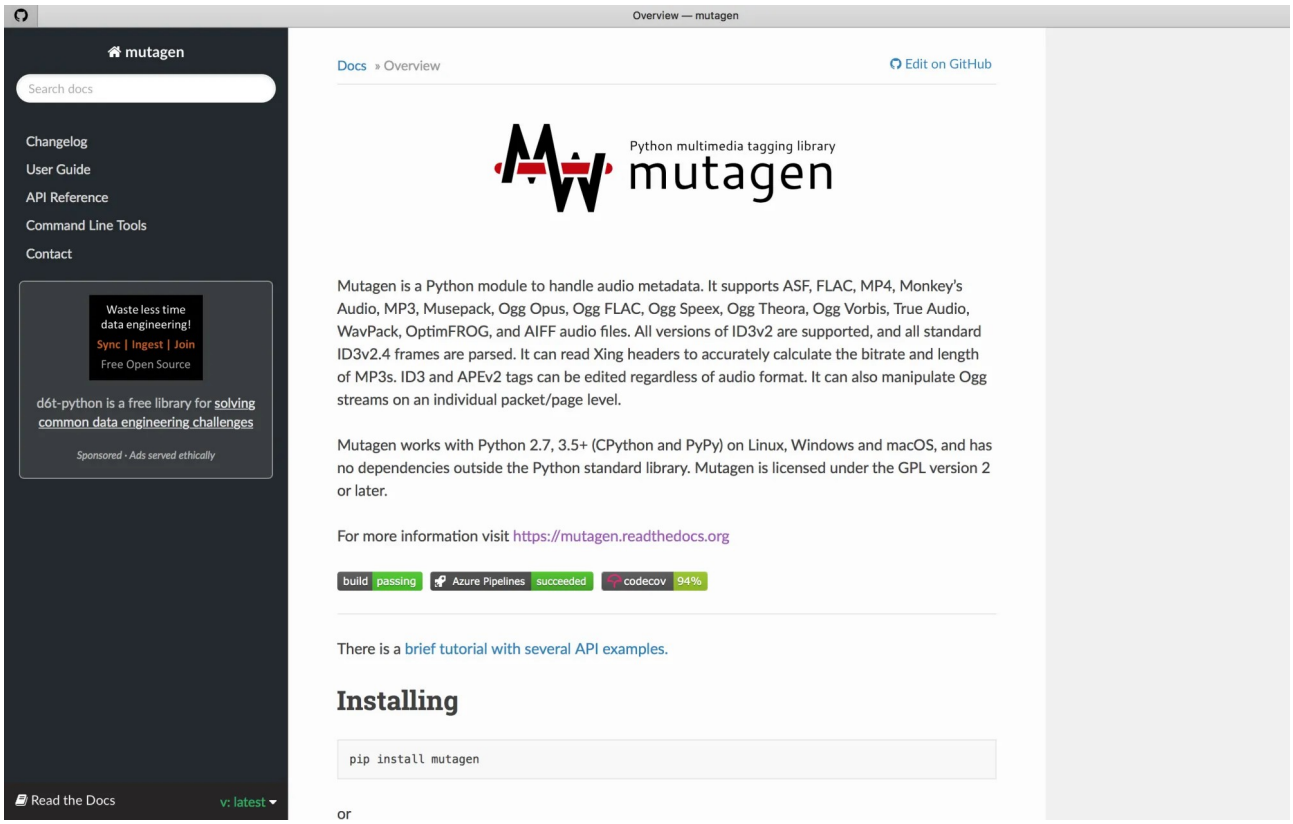
[3. TimeSide –]

It is a well design python framework for Audio Analysis . Specially for labelling , transcoding, streaming etc .It is more popular for audio processing in python with web .



Mutagen –

This is really one of the great python module for audio processing specially tagging ,and meta data extraction . Mutagen also provide command line interface .



Others –

Truely speaking ! To provide a particular name at this place will be injustice to others Python Audio Processing and Analysis Library . Hence I have decide to create a bucket for this . Here are a list of some more interesting Python Libraries for Audio Processing –

1.audiolazy

2. audioread

3.beats

Audio Processing and Machine Learning –

Audio processing is harder with Machine Learning .Actually before sending directly to Machine Learning Platform so many hidden tasks. Which are quite time taking but seems small . Like we have to load the sound . The imported or loaded audio sample may be of some different format . We have to first convert them into the required one. Now the above mention Library comes to the role . Few of them are coming with such features of format conversion .

Now once it is converted into the required format , we have to perform the preprocessing like noise removal and all . After it the last and the most important step comes where we have to extract the feature from the audio sample . Finally it becomes c a typical machine learning stuff after the feature engineering .

Conclusion –

In this article we tried to cover the Audio Processing stuffs with Python Library . You may solve most of Audio processing stuffs using this libraries . So friends I hope this article **” Top 5 Audio Analysis Library for Python : Must for Data Scientist ” ,** must clear your doubt .Anyways if you want to discuss some more on it , Please write back to us . Audio Processing and Analysis is little different then text and image processing . If you think you may contribute some more on this topic , Data Science Learner’s Team always appreciate such efforts as guest posting .

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

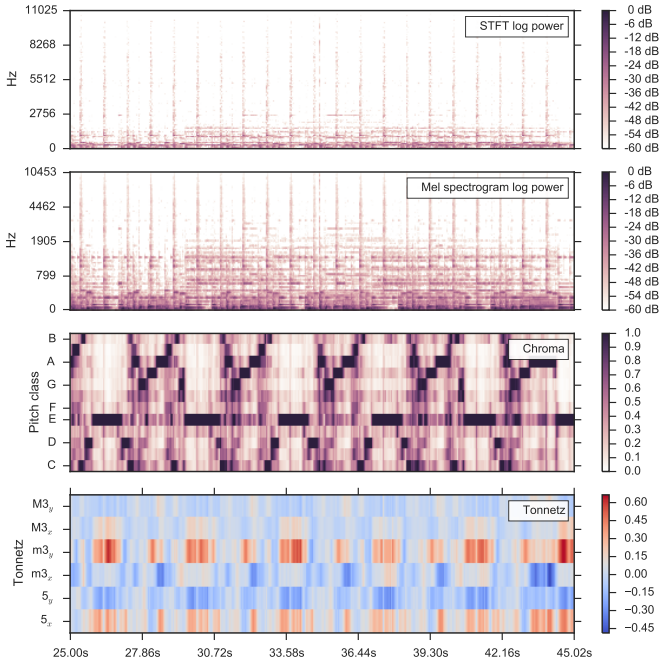


Fig. 1: *First: the short-time Fourier transform of a 20-second audio clip (`librosa.stft`). Second: the corresponding Mel spectrogram, using 128 Mel bands (`librosa.feature.melspectrogram`). Third: the corresponding chromagram (`librosa.feature.chroma_cqt`). Fourth: the Tonnetz features (`librosa.feature.tonnetz`).*

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

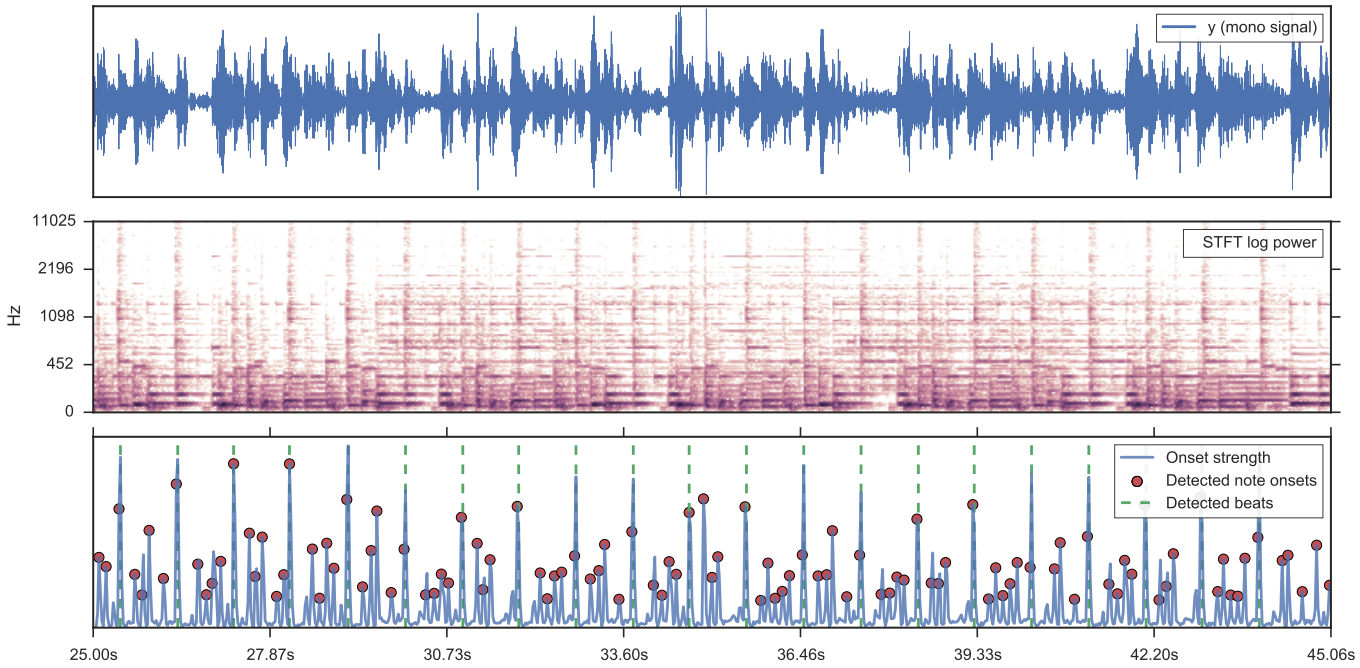


Fig. 2: *Top: a waveform plot for a 20-second audio clip* `y`*, generated by* `librosa.display.waveplot`*. Middle: the log-power short-time Fourier transform (STFT) spectrum for* `y` *plotted on a logarithmic frequency scale, generated by* `librosa.display.specshow`*. Bottom: the onset strength function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

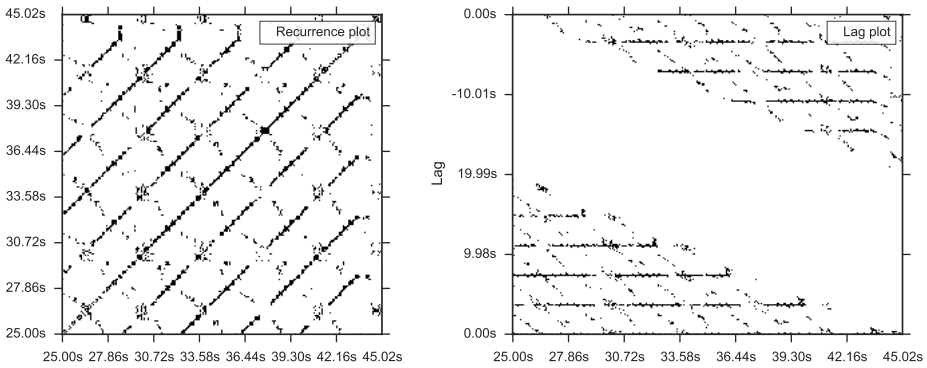


Fig. 3: *Left: the recurrence plot derived from the chroma features displayed in Figure 1. Right: the corresponding time-lag plot.*

```
>>> 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.:

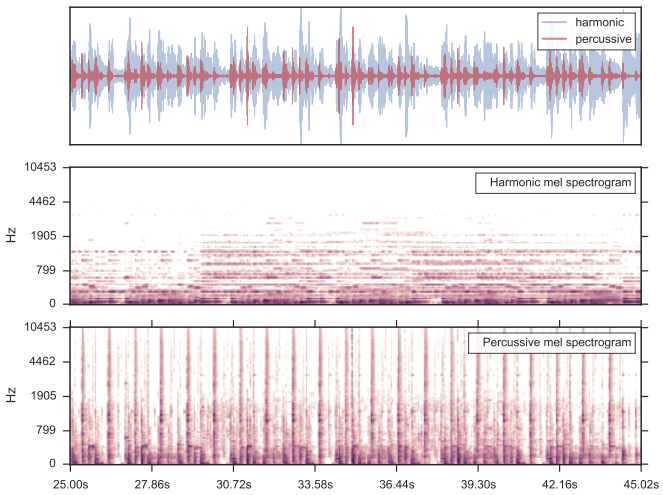


Fig. 4: *Top: the separated harmonic and percussive waveforms. Middle: the Mel spectrogram of the harmonic component. Bottom: the Mel spectrogram of the percussive component.*

```
>>> 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

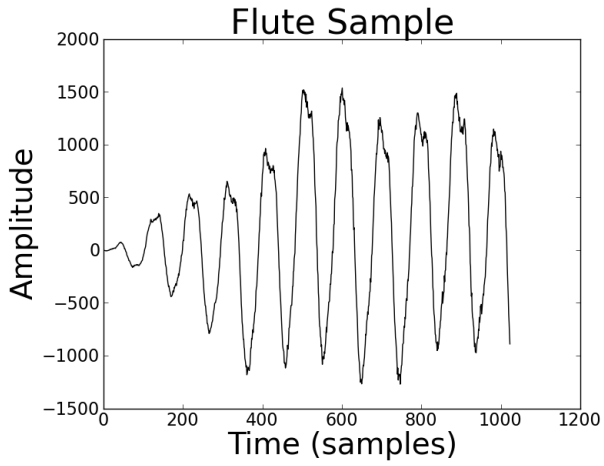


Figure 1: Plot of audio samples, generated by the code given in Listing 1.

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

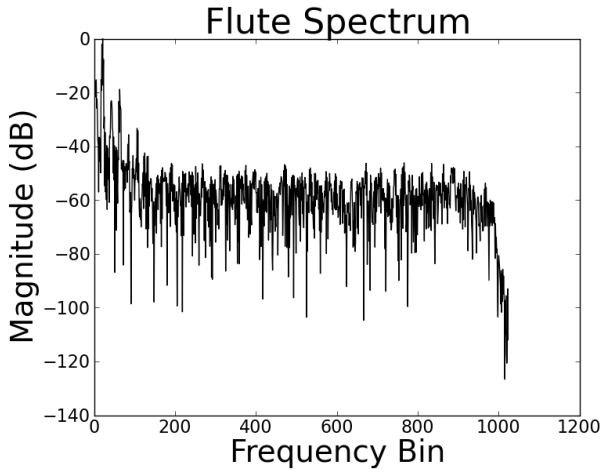


Figure 2: Flute magnitude spectrum produced from code in Listing 2.

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

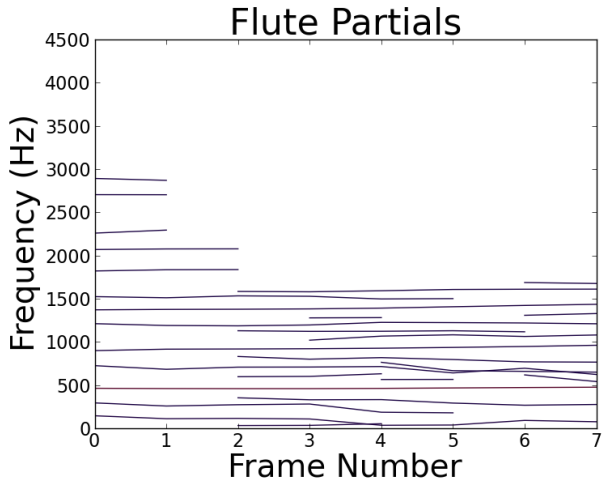


Figure 3: Partials detected in the first 8 frames of a flute sample, produced by the code in Listing 3. Darker colours indicate lower amplitude partials.

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

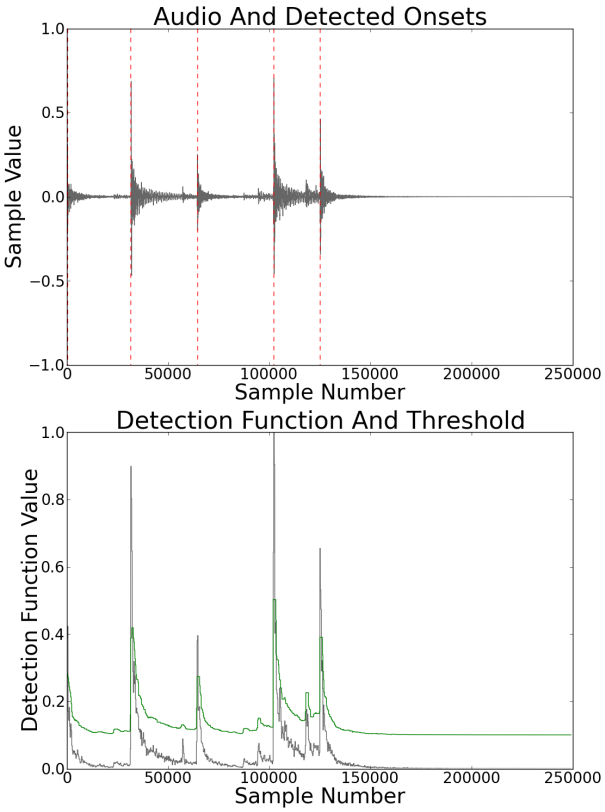


Figure 4: The upper plot shows an audio sample with detected onsets indicated by dashed red lines. The lower plot shows the detection function that was created from the audio file (in grey) and the peak picking threshold (in green).

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).