

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 Coefficients (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. Lifting 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

- [Steps at a Glance](#)
- [Why do we do these things?](#)
- [What is the Mel scale?](#)
- [Implementation steps](#)
- [Computing the Mel filterbank](#)
- [Deltas and Delta-Deltas](#)
- [Implementations](#)
- [References](#)
- [Related pages on this site:](#)



LOUIS VUITTON

节日臻礼

为她挑选

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 perceived 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:

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

To go from Mels back to frequency:

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

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:

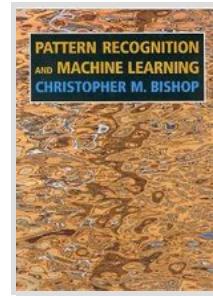
$$S_i(k) = \sum_{n=1}^N s_i(n)h(n)e^{-j2\pi kn/N} \quad 1 \leq k \leq K$$

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 coefficients.

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 from 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 coefficients. Once this is performed we are left with 26

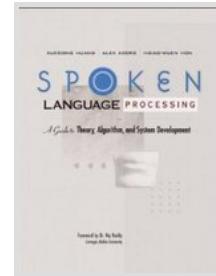


Pattern Recognition and Machine Learning

ASIN/ISBN: 978-0387310732

"The best machine learning book around"

[Buy from Amazon.com](#)

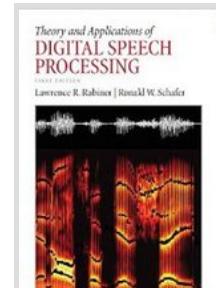


Spoken Language Processing: A Guide to Theory, Algorithm and System Development

ASIN/ISBN: 978-0130226167

"A good overview of speech processing algorithms and techniques"

[Buy from Amazon.com](#)



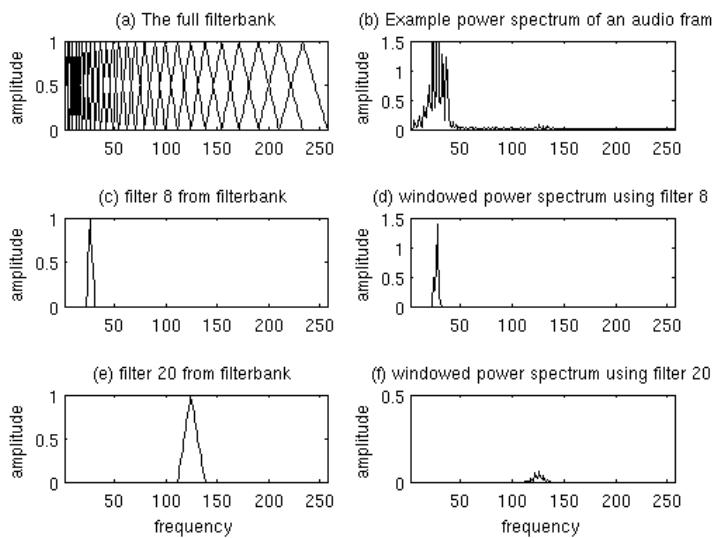
Theory and Applications of Digital Speech Processing

ASIN/ISBN: 978-0136034285

"A comprehensive guide to anything you want to know about speech processing"

[Buy from Amazon.com](#)

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 coefficients. 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 frequencies to fft bin numbers we need to know the FFT size and the sample rate,

$$f(i) = \text{floor}((nfft+1)*h(i)/sampleRate)$$

This results in the following sequence:

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

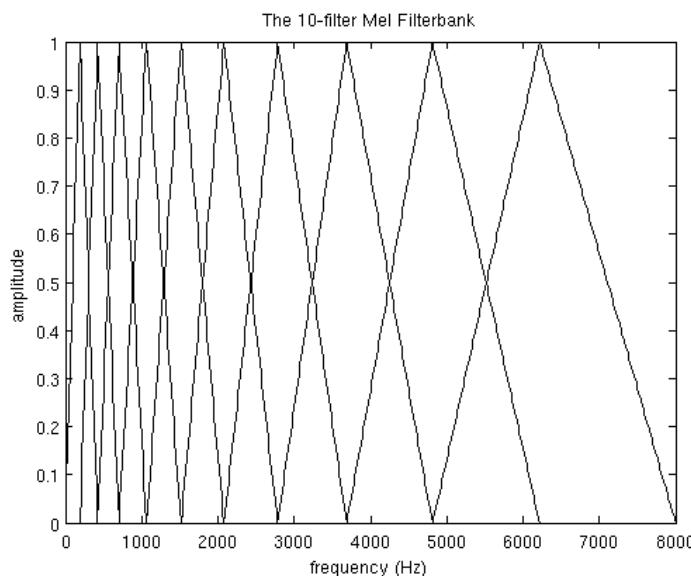
We can see that the final filterbank finishes at bin 256, which corresponds to 8kHz with a 512 point FFT size.

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:

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

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](#)

Disqus seems to be taking longer than usual. [Reload?](#)

comments powered by Disqus

ybl krq ibf kfnlh r kfsqyrdq mlxdqh mv trppvdqx
- tfqzstdsh

Copyright & Usage

Copyright James Lyons © 2009-2012
No reproduction without permission.

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

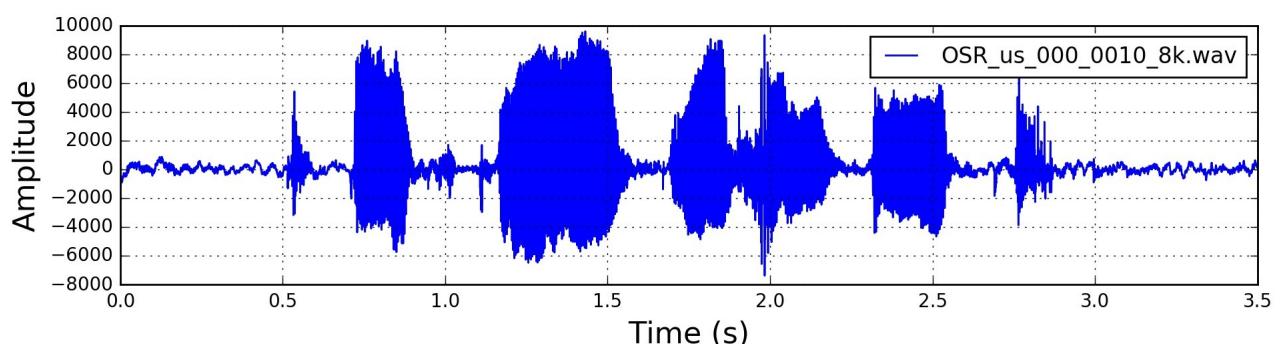
For this post, I used a 16-bit PCM wav file from [here](#), called “OSR_us_000_0010_8k.wav”, which has a sampling frequency of 8000 Hz. The wav file is a clean speech signal comprising a single voice uttering some sentences with some pauses in-between. For simplicity, I used the first 3.5 seconds of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

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

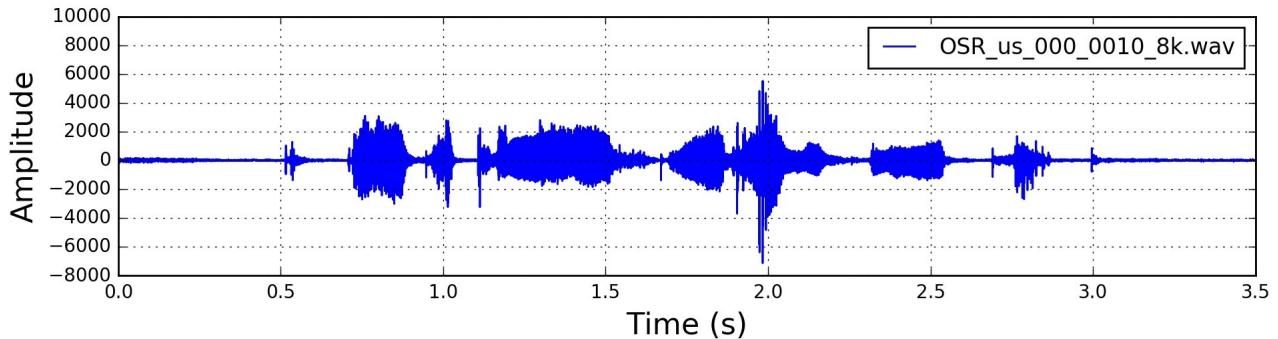
which can be easily implemented using the following line, where typical values for the filter coefficient (α)

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)]
```

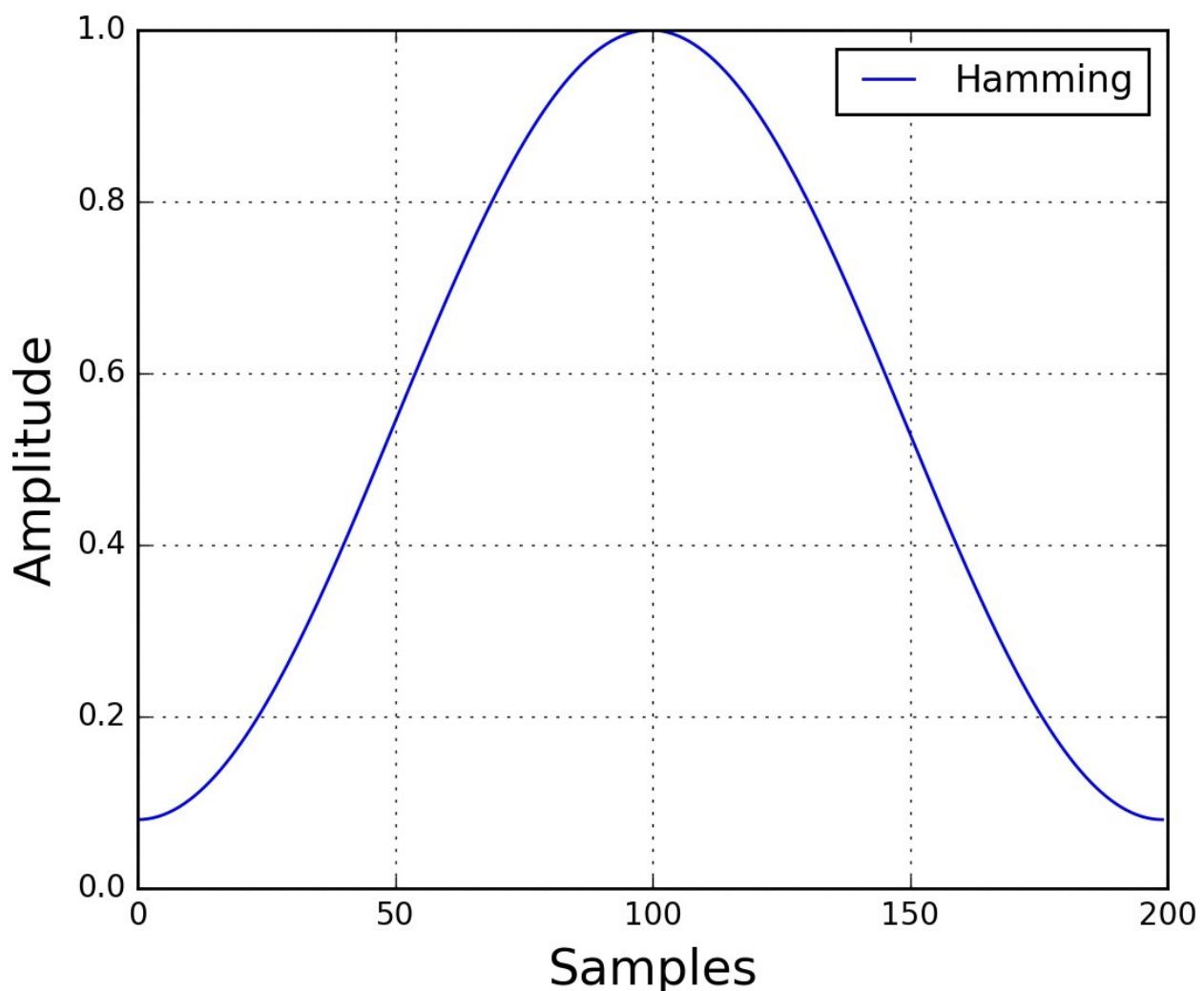
Window

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:

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

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

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



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, $\text{NFFT} = 512$; and then compute the power spectrum (periodogram) using the following equation:

$$P = |FFT(x_i)|^2 N$$

where, x_i

is the i th 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, $\text{nfilt} = 40$ 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

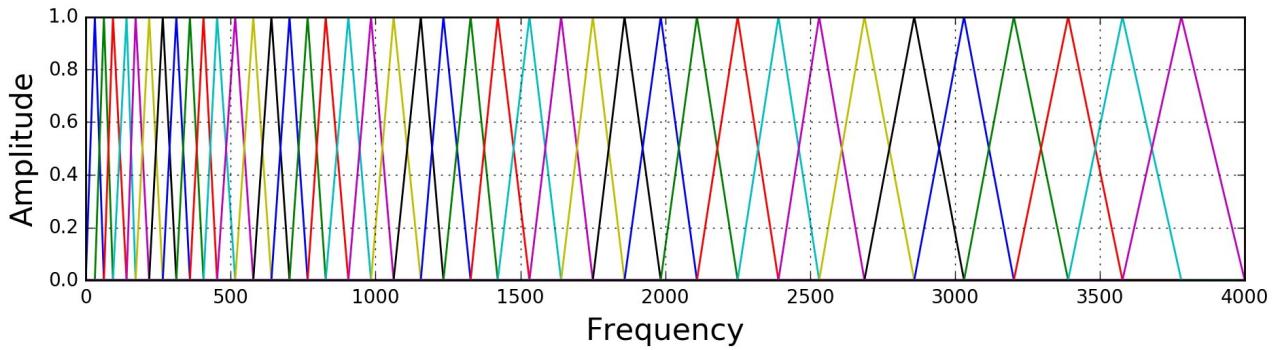
) and Mel (m

) using the following equations:

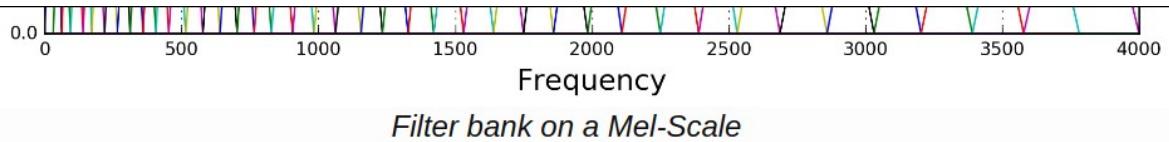
$$m = 2595 \log_{10}(1 + f/700)$$

$$f = 700(10m/2595 - 1)$$

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](#)):



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

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

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

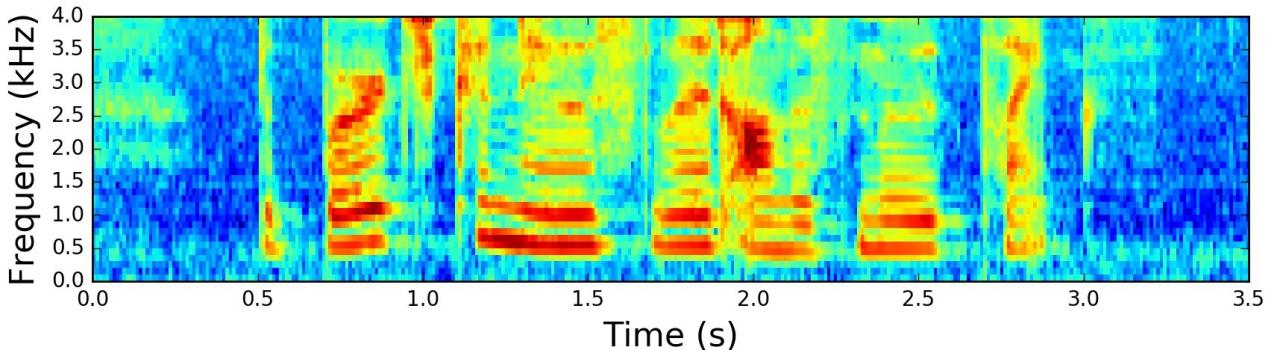
```

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)

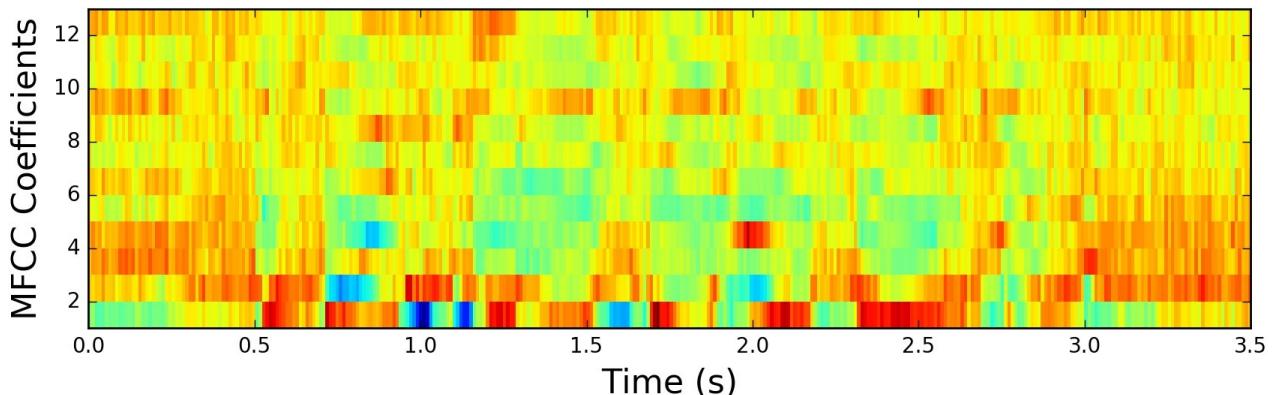
It turns out that filter bank coefficients computed in the previous step are highly correlated, which could be problematic in some machine learning algorithms. Therefore, we can apply Discrete Cosine Transform (DCT) to decorrelate the filter bank coefficients and yield a compressed representation of 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

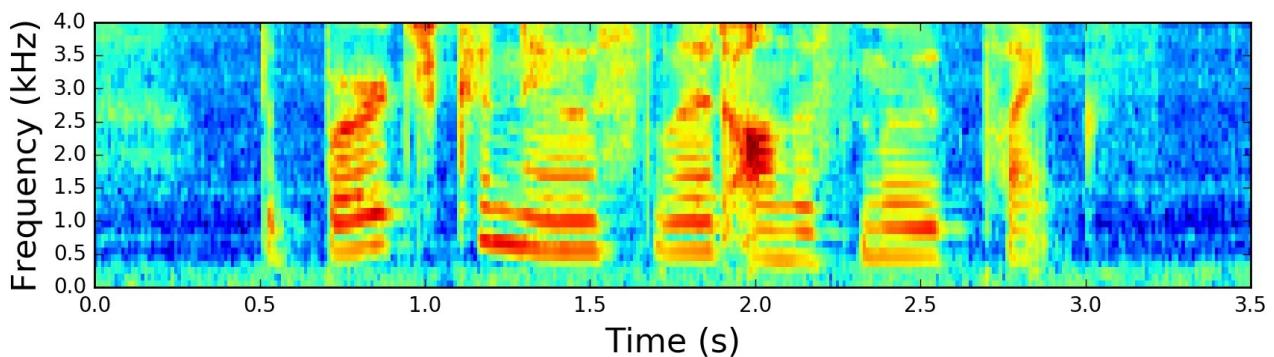


Mean Normalization

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

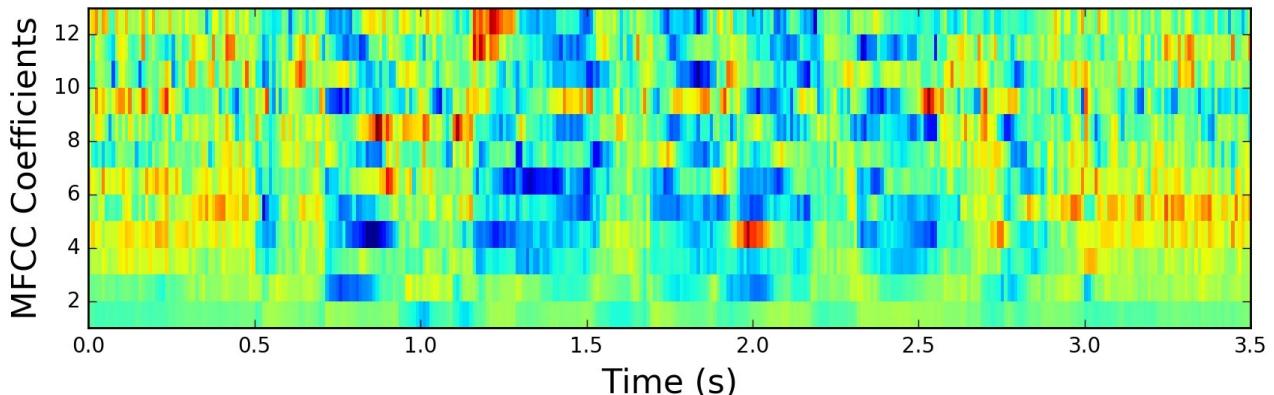


Normalized Filter Banks

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 *spectral* and *cepstral* with *filtering* and *liftering* respectively. [←](#)
 2. An excellent discussion on this topic is in [this thesis](#). [←](#)

DataGenetics

[Home](#) [Blog](#) [About Us](#) [Work](#) [Content](#) [Contact Us](#)

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.



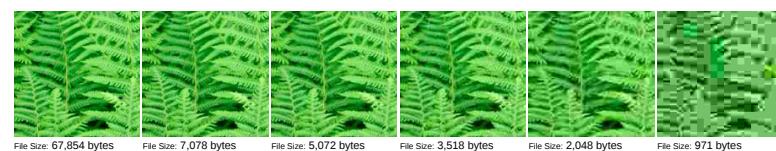
$$= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

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 compression, we have to be careful 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 (those 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.



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.

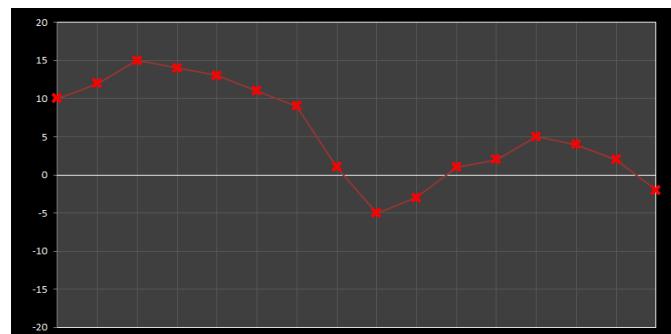
Notice the 'blocky' appearance of the image on the far right? This is an artifact of how the DCT compression is calculated. Again, we'll look at this later.

Advertisement

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.

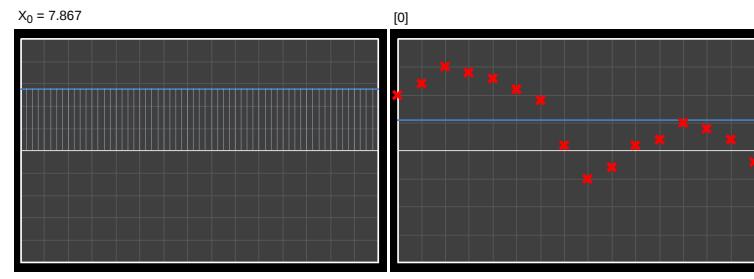
The math required to perform DCT is not very complicated, but it's also not rated **PG-13**. I'm opening with the concepts, then I'm going to branch straight past the implementation steps and move onto the results. For those that like to code, and want to experiment with this, let me give the standard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

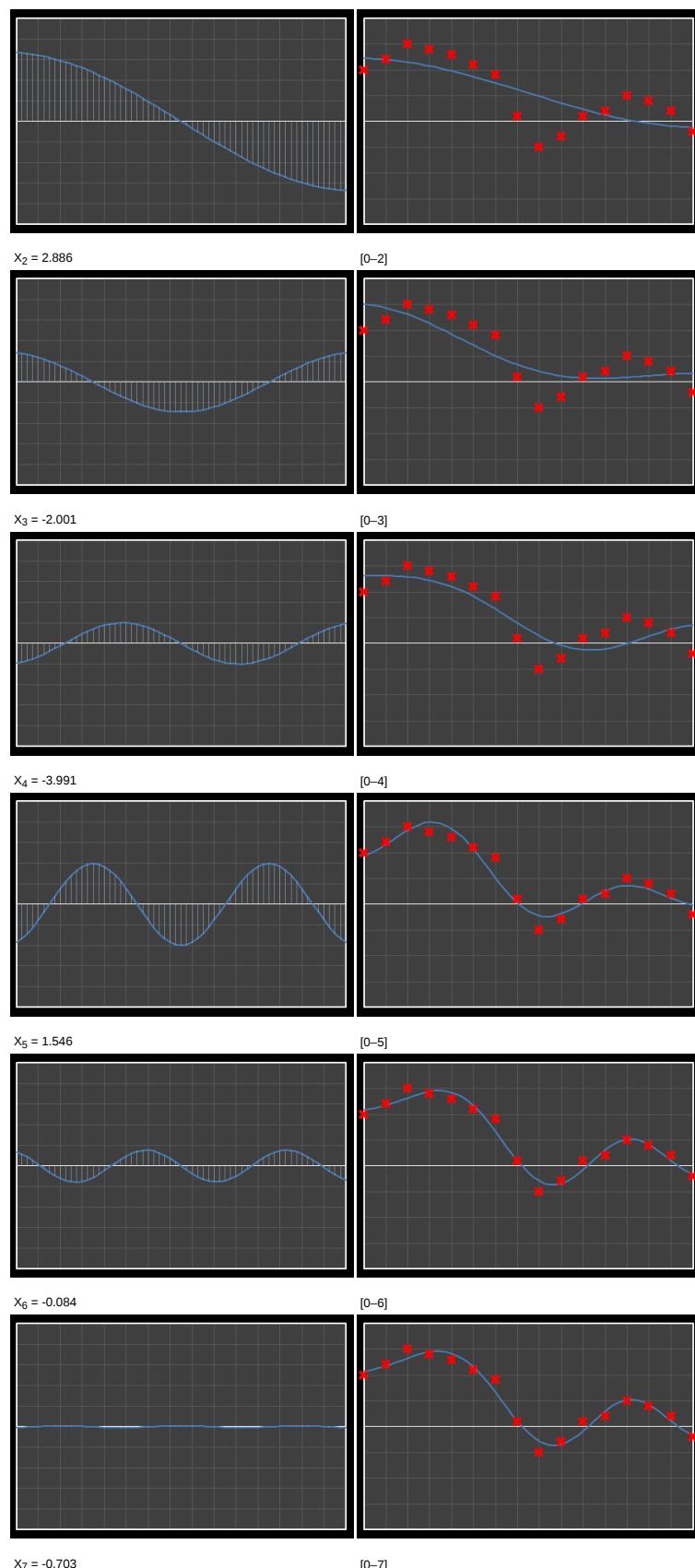


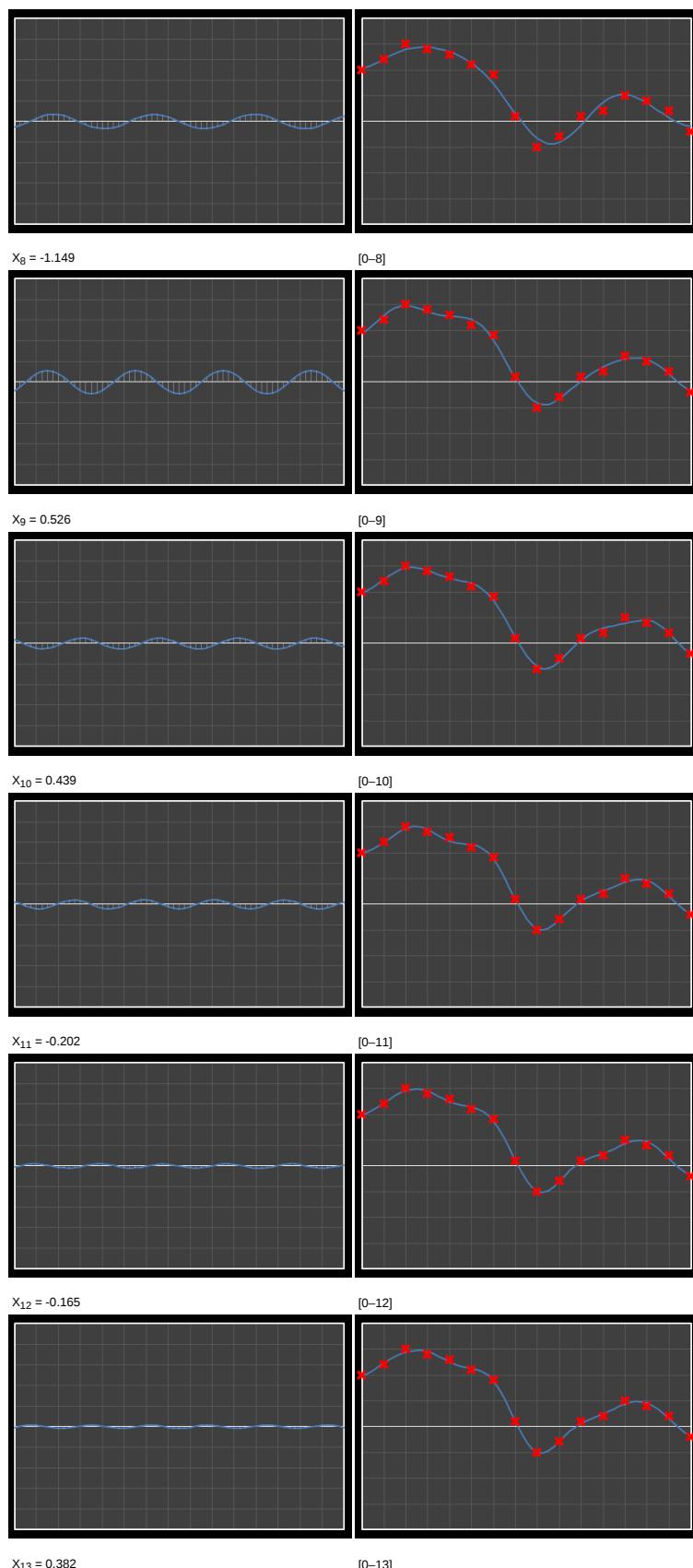
Using DCT, we can break this curve into a series of Cosine waves of various frequencies. It is by the superposition (adding together) of these fundamental waves that we recreate the original waveform.

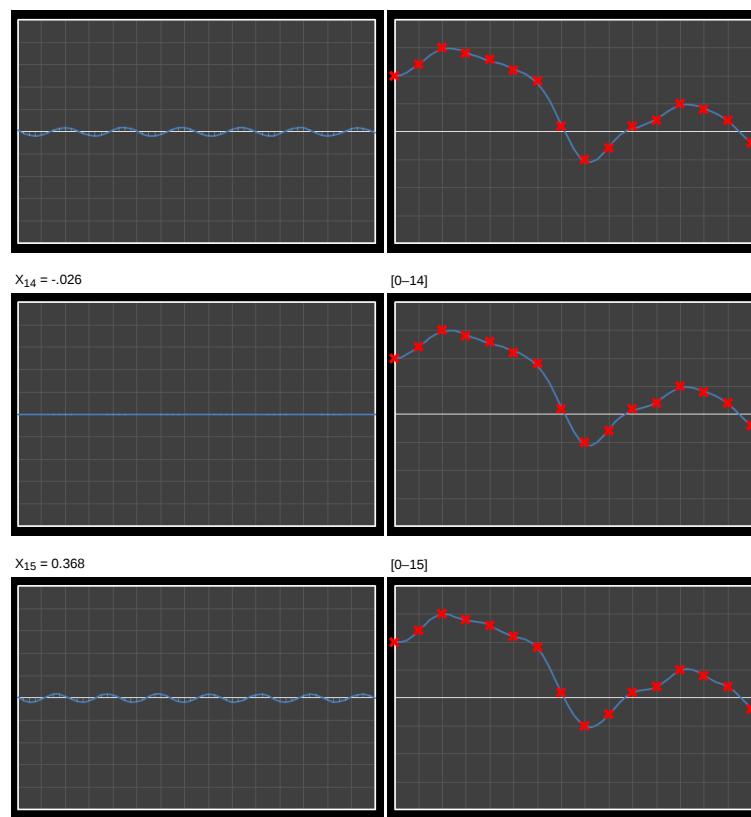
For the above sixteen points, I've broken down the data using DCT. The graphs on the left below show the cosine function of that frequency component and the image on the right shows the superposition of this to the running total (this component added to all those above it). The lowest frequency components are shown first, and as we move down the page, the higher frequency components are added. Above this 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.



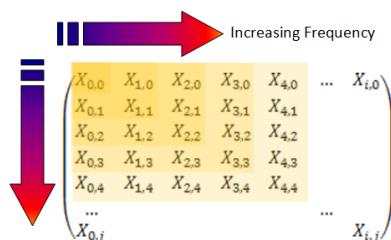






Example: Two Dimensions (and the basics of JPEG compression)

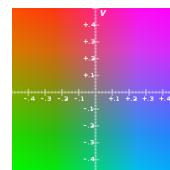
We can apply just the same technique in two dimensions, this time breaking the image into blocks of pixels and looking at the harmonics in each block. The result of this analysis is a matrix of coefficients. Moving down and to the right are the coefficients with increasingly higher frequency components.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



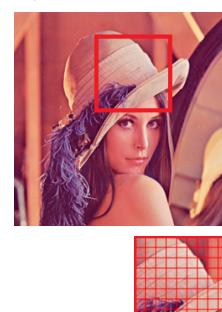
In JPEG compression, the block size used is 8×8 , resulting in a similar size matrix of coefficients. However, for my examples, I'm going to select a larger block size of 16×16 (I think it's easier to visualize things at this size.)

As you probably know, colors in computer images are often described 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 sensitivity you need fewer distinct levels of shade of a color than the brightness of a color to still maintain a smooth image.

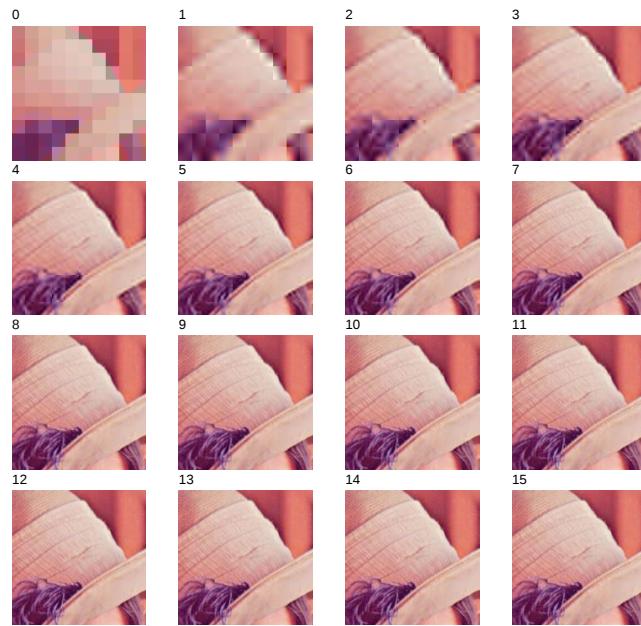


Because we are working on image processing, we have to roll out *Lenna*, the "The First Lady of the Internet". This image is probably the most widely used test image in computer history (click on the link for background details).



We're going to look at a selection of 16×16 pixel blocks around the hat. Below are sixteen versions 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 8×8 block, to produce an 8×8 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.

1. Create DCT coefficient matrix

$$\begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.13 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.88 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & 5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.08 \end{bmatrix}$$

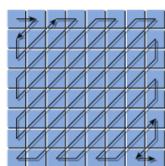
2. Scale each coefficient

3. Convert to integers

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Quantization Matrix

The results of this division are then converted to integers (from floating point), causing further quantization. A typical output matrix of this process is shown 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.

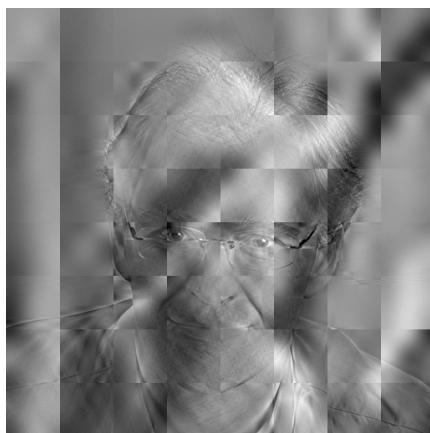


To make a hybrid image containing two images we use the digital equivalent of a high-pass and low-pass filter. In the example below, I've taken the Lena image and passed it through DCT to convert it to the frequency domain. (To make this a little clearer, for this example, I've also converted the image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

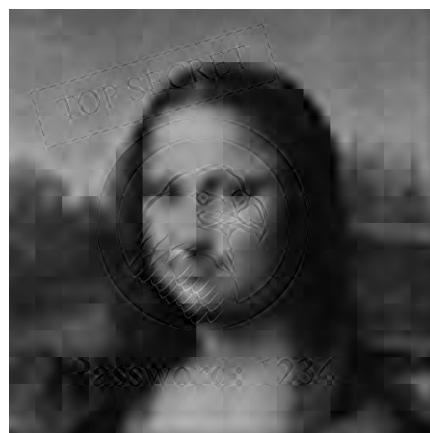
Hidden Letters

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 eavesdropper 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?

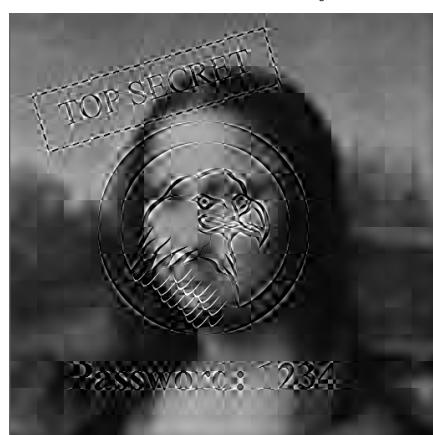


Can't see it? Here's a version where the "Hidden Text" is even higher contrast.

Need more convincing?

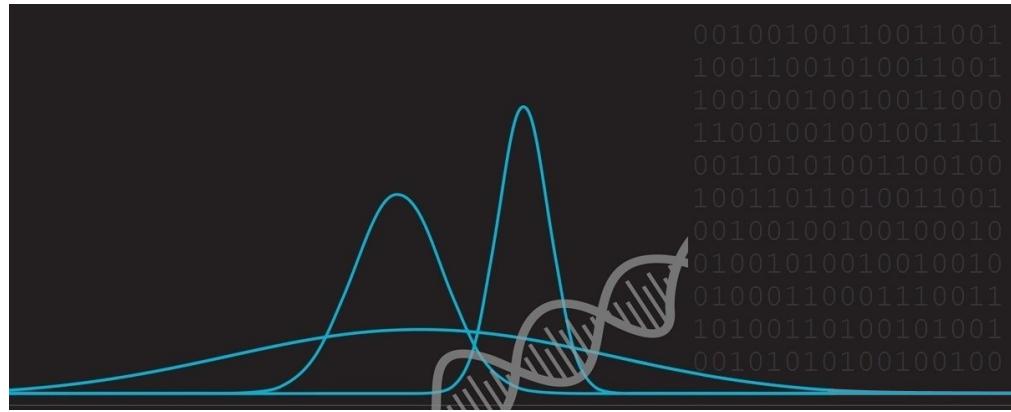
The image below is the same image as the one to the left. All I've done, if you *View Source* for the page, is override the width of the image from the default of 512 pixels to 128 pixels. Your browser has scaled down the image using an appropriate algorithm, and has averaged out the pixels, reducing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

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



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.



© 2009-2013 DataGenetics



» [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 musical 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. Its 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 its 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 primitives that are designed to be 'cascaded' to create 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 its 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, demultiplexing, 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 developed 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.

- » [!\[\]\(5072da0ee386d5ec63594f37d40bf9e0_img.jpg\) sounddevice](#) - This module provides bindings for the [!\[\]\(6b16540009b3a94452f5491f79c2ff15_img.jpg\) PortAudio](#) library (using [!\[\]\(47d14c1fbb974814228950ba352b262b_img.jpg\) CFFI](#)) and a few convenience functions to play and record [!\[\]\(ac53eb1d163c06dc639ae945974ab81f_img.jpg\) NumPy](#) arrays containing audio signals.

Community

- » [!\[\]\(1b278f2714738b87a10bf7ee8dcbf246_img.jpg\) 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

- » [!\[\]\(b26dbeaa7c1303ba114d81ed21a7f7a6_img.jpg\) 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. [!\[\]\(981b0e016292b10ea699c3c13f25a5c0_img.jpg\) Csound on Sourceforge](#); [!\[\]\(ff66999a005fb117b29c2a98915c79d7_img.jpg\) Csound-Python](#) and [!\[\]\(ed4e22ee2fee41518c51cd39fa33b805_img.jpg\) Csound](#) (some brief tutorials on the OLPC Wiki)
- » [!\[\]\(d6fbbc5d821c24f549f89a53836e9e06_img.jpg\) Csound Routines](#) - set of routines to manipulate and convert csound files
- » [!\[\]\(2d824c0ab4898acb4693fbebd89a4b5b_img.jpg\) PMask](#) - Python implementation of CMask, a stochastic event generator for Csound.

MP3 stuff and Metadata editors

- » [!\[\]\(db3b6a4105578dfb1724b2b4978e8fc8_img.jpg\) 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.
- » [!\[\]\(b9d7305e6f9d90cf8adfdd82454a9645_img.jpg\) 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 structure.
- »  **KaaMetadata** Successor 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 originally 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 bugs 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.
- »  **portmidizer0** - portmidizer0 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/permuations. 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 as 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), programmatically (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 this 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 parameters every 500 iterations.

Finally, you can generate a full composition after training is complete. The function `gen_adaptive` in main.py will generate a piece and also prevent long empty gaps by increasing note probabilities 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 progress at the moment.

Right now, there is no separate validation step, because my initial goal was to produce interesting music, not to assess the accuracy of this method. It does, however, print out the cost on the 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", "wb"))
```
```

and if you want to load them, you can do

```
```python
m.learned_config = pickle.load(open("path_to_weight_file.p", "rb"))
```
```

```
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_learned_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, Layer, 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 systems  
    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):  
    """  
        Initializes the recurrence relation with an initial hidden state  
        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 already 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_note)
        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_parallel, 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=[time_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, batch/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. Starts with [0,0], doesn't include last note.
        # In (note, batch/time, 2) format
        # Shape of start is thus (1, N, 2), concatenated with all 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 versa.
        note_inputs = T.concatenate( [time_final, note_choices_inputs], axis=2 )
        num_timebatch = note_inputs.shape[1]

```

```

        # apply dropout
        if self.dropout > 0:
            pitch_masks = theano_lstm.MultiDropout( [(num_timebatch, 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=[note_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, batch/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 likelihood of the events all happening.
        # For the purposes of training, if the ouputted probability 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 probabilities given all previous decisions, we can just
        # multiply the likelihoods, or, since we are logging them, add the logs.

        # Note that we mask out the articulations for those notes 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_notes], axis=3)

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

        updates, _, _, _, _ = create_optimization_updates(self.cost, 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_st
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 python-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/repos/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_LIBRARY_PATH=/usr/local/cuda/lib64" >> .bashrc
sudo reboot
sudo apt-get update
sudo apt-get -y dist-upgrade

cuda-install-samples-7.0.sh ~/
cd NVIDIA_CUDA-7.0_Samples/
cd 1_Utils/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[nvcc]\nfastmath=True\n[cuda]\nnroot=/usr/local/cuda\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(12)]
        return part_position + part_pitchclass + [0]*66 + [1]
    return [noteSentinel(note) for note in range(upperBound-lowerBound)]


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, (time//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 in range(len(state))]

def noteStateMatrixToInputForm(statematrix):
    # NOTE: May have to transpose this or transform it in some way to make Theano like it
    #[startSentinel()] +
    inputform = [noteStateSingleToInputForm(state, time) for time, state in enumerate(statematrix)]
    return inputform
Copyright (c) 2016, Daniel Johnson
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```
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.resolution / 8):
        # Crossed a note boundary. Create a new state, defaulting 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 bounds (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 time signatures. Bail early!
                    # print "Found time signature event {}".format(evt)
                    Bailing!
            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-lastcmdtime)*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,division_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_width)])
    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])), axis=0), 'output/sample{}'.format(i))
            pickle.dump(model.learned_config, open('output/params
{}{}.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(thought) ] + sequence

        return (np.concatenate(sequence, axis=1 ) *255).astype('uint8')

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[time][note_idx])
                for time in range(len(probs))
        ]
        for note_idx in range(len(probs[0]))
    ])

def thoughtsAndPastToStackedArray(thoughts, probs, succs, len_past):
    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], thoughts[-1], 0),
            vert_spacer,
            probAndSuccessToImgArray(thoughts[-3], thoughts[-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(internalMatrixToImgArray(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')
```