Mel Frequency Cepstral Coefficient (MFCC) tutorial

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

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

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

Steps at a Glance

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

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

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

Why do we do these things?

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

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

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

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

Contents

Further reading

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

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

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

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

What is the Mel scale?

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

The formula for converting from frequency to Mel scale is:

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

To go from Mels back to frequency:

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

Implementation steps

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

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

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

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

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

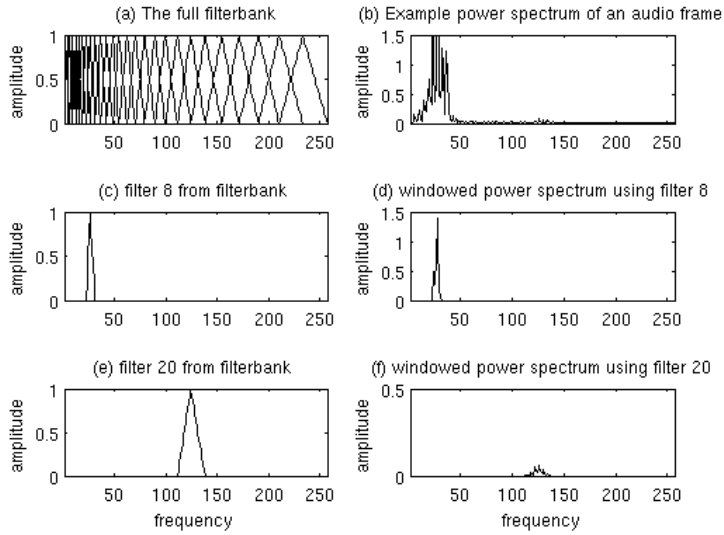
where $h(n)$ is an N sample long analysis window (e.g. hamming window), and K is the length of the DFT. The periodogram-based power spectral estimate for the speech frame $s_i(n)$ is given by:

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

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

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

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



Plot of Mel Filterbank and windowed power spectrum

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

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

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

Computing the Mel filterbank

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

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

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

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

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

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

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

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

   This results in the following sequence:

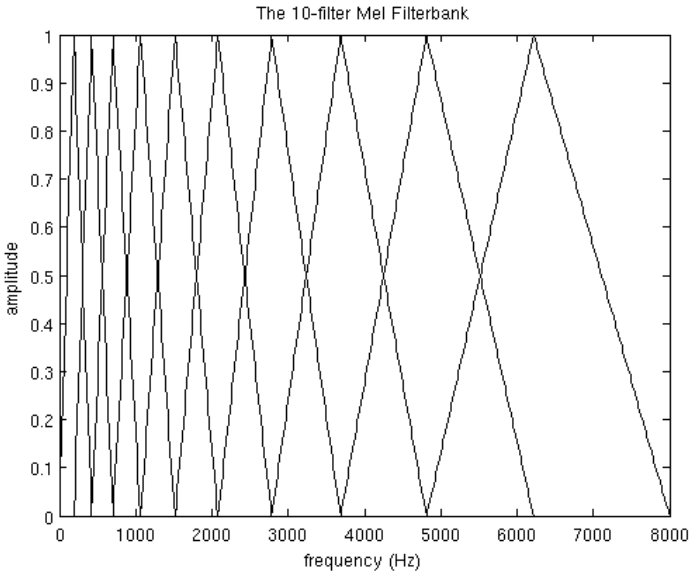
   ```
   f(i) =  9, 16,  25,   35,   47,   63,   81,  104,  132, 165,  206,  256
   ```

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

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

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

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



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

Deltas and Delta-Deltas

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

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

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

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

Implementations

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

There is a good MATLAB implementation of MFCCs over here.

References

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

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

Related pages on this site:

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

ybl krq ibf kfnlh r kfsqyrdq mlxdqh mv trppvdqx
- tfqzstdsh

Questions/Feedback

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

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

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

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

Setup

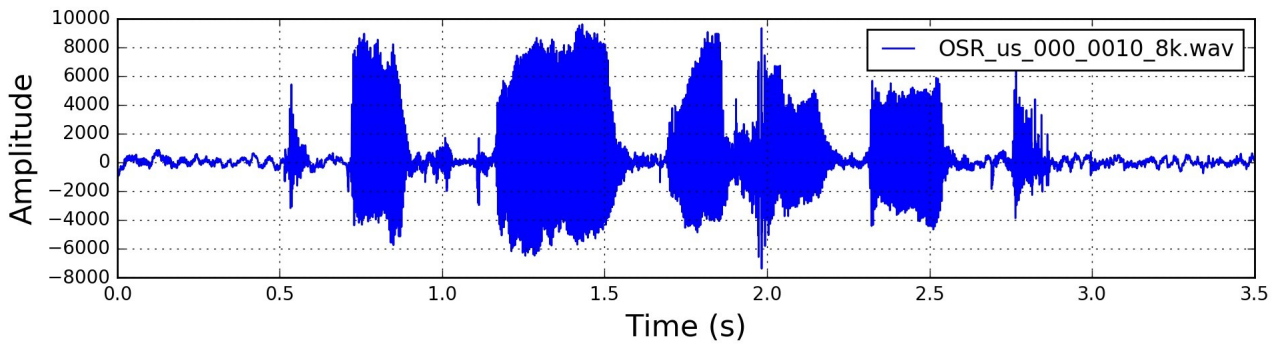
For this post, I used a 16-bit PCM wav file from here, called “OSR_us_000_0010_8k.wav”, which has a sampling frequency of 8000 Hz. The wav file is a clean speech signal comprising a single voice uttering some sentences with some pauses in-between. For simplicity, I used the first 3.5 seconds of the signal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

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

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

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

The raw signal has the following form in the time



Pre-Emphasis

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

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

using the first order filter in the following equation:

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

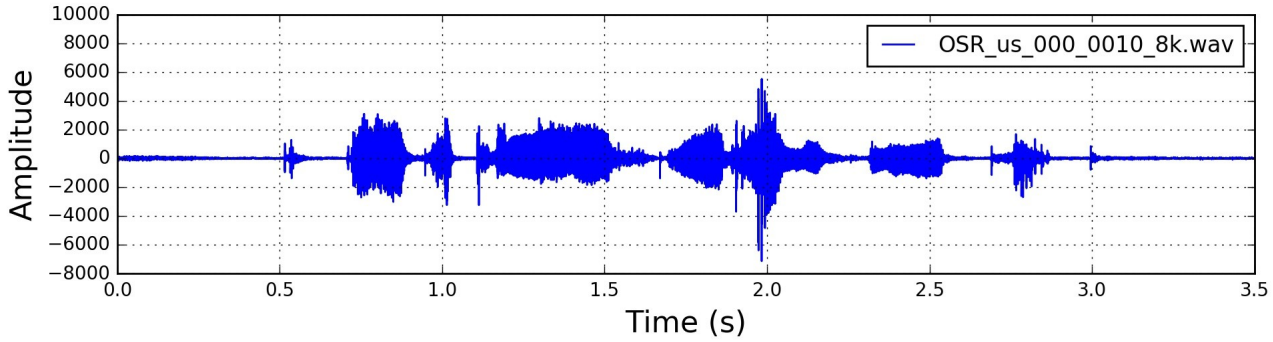
which can be easily implemented using the following line, where typical values for the filter coefficient (α

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

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

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

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



Framing

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

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

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

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

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

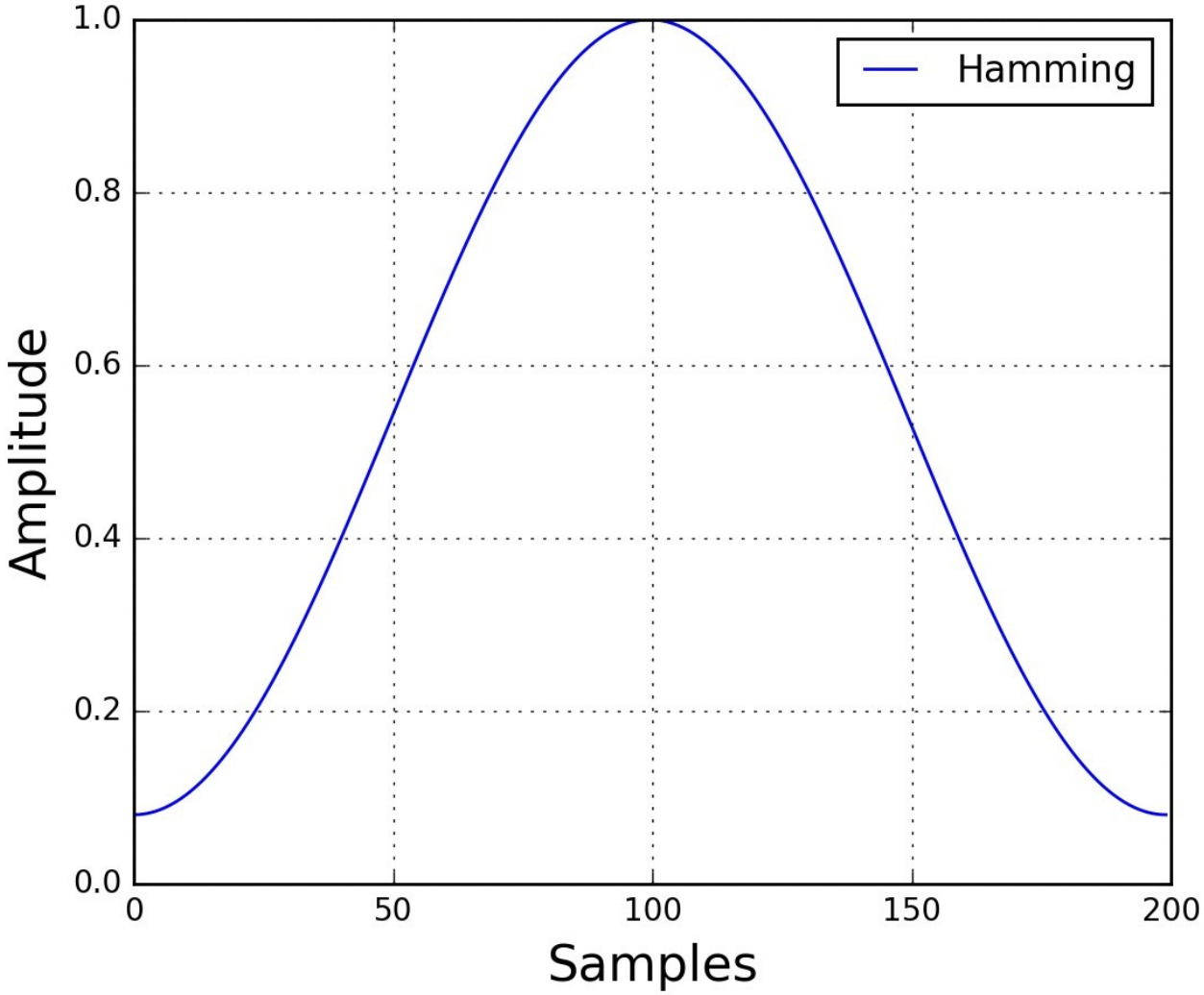
Window

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

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

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

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


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

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

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

Fourier-Transform and Power Spectrum

We can now do an N

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

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

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

where, xi

is the ith frame of signal x

. This could be implemented with the following lines:

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

Filter Banks

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

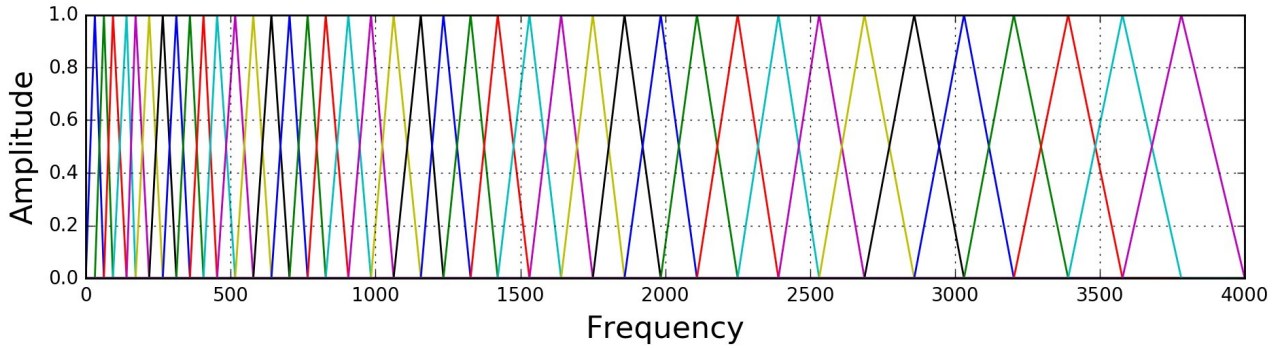
) and Mel (m

) using the following equations:

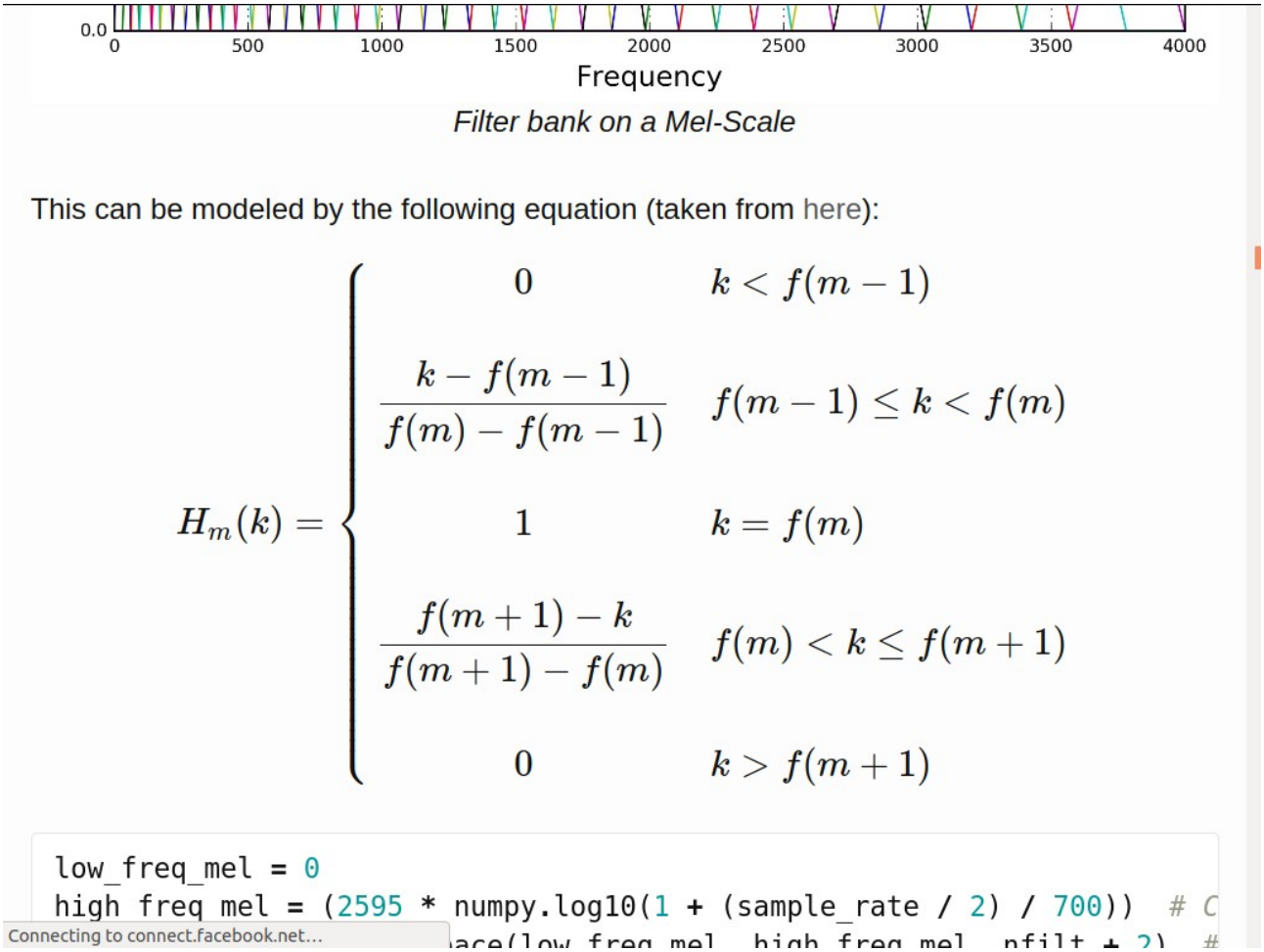
$$m=2595\log10(1+f700)$$

$$f=700(10m/2595−1)$$

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



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



Filter bank on a Mel-Scale

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

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

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

Connecting to connect.facebook.net...

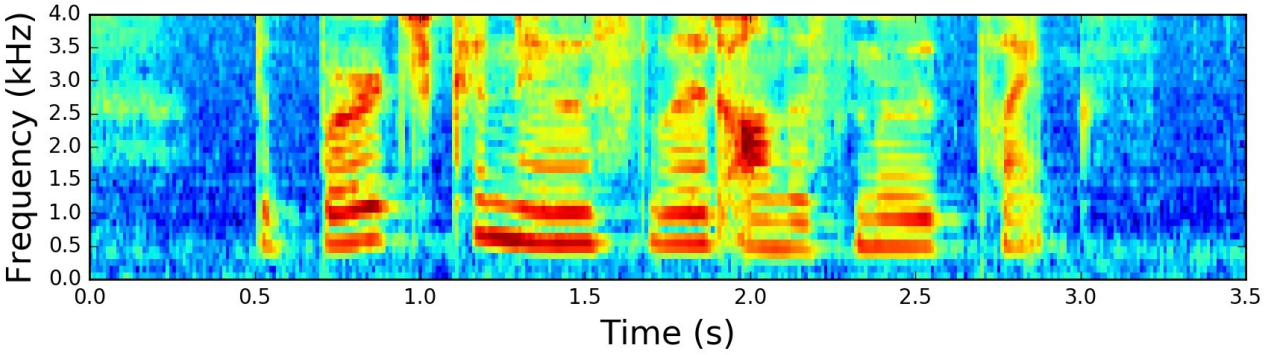
```
low_freq_mel = 0
high_freq_mel = (2595 * numpy.log10(1 + (sample_rate / 2) / 700))  # Convert Hz
to Mel
mel_points = numpy.linspace(low_freq_mel, high_freq_mel, nfilt + 2)  # Equally
spaced in Mel scale
hz_points = (700 * (10**(mel_points / 2595) - 1))  # Convert Mel to Hz
bin = numpy.floor((NFFT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

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

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

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

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



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

Mel-frequency Cepstral Coefficients (MFCCs)

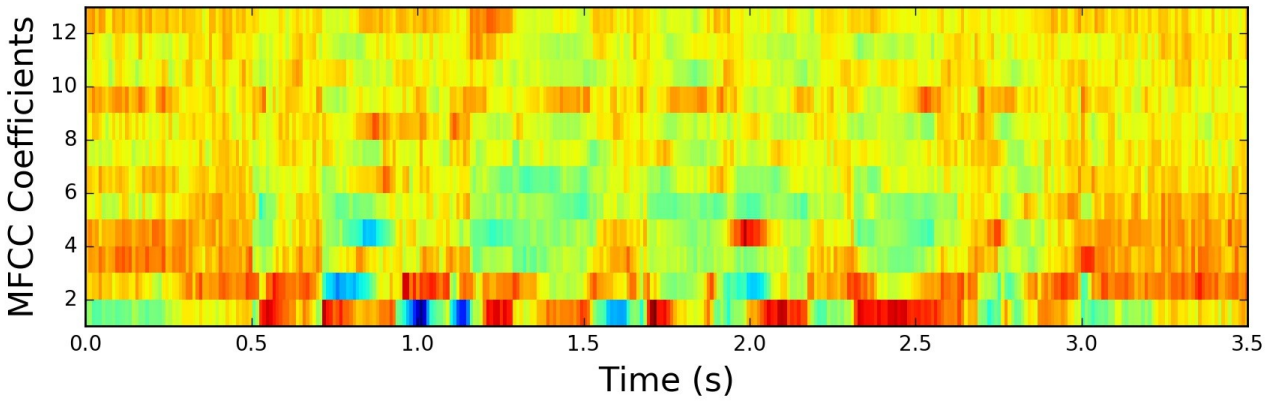
It turns out that filter bank coefficients computed in the previous step are highly correlated, which could be problematic in some machine learning algorithms. Therefore, we can apply Discrete Cosine Transform (DCT) to decorrelate the filter bank coefficients and yield a compressed representation of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

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

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

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

The resulting MFCCs

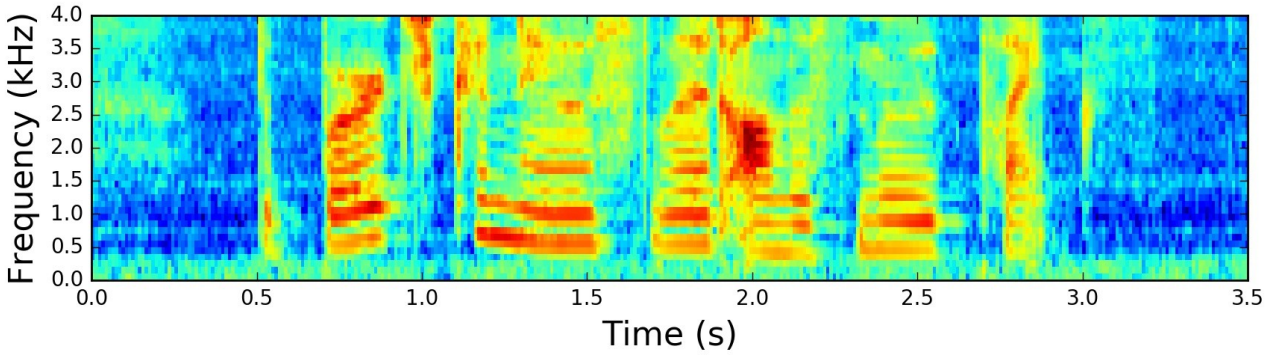


Mean Normalization

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

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

The mean-normalized filter banks

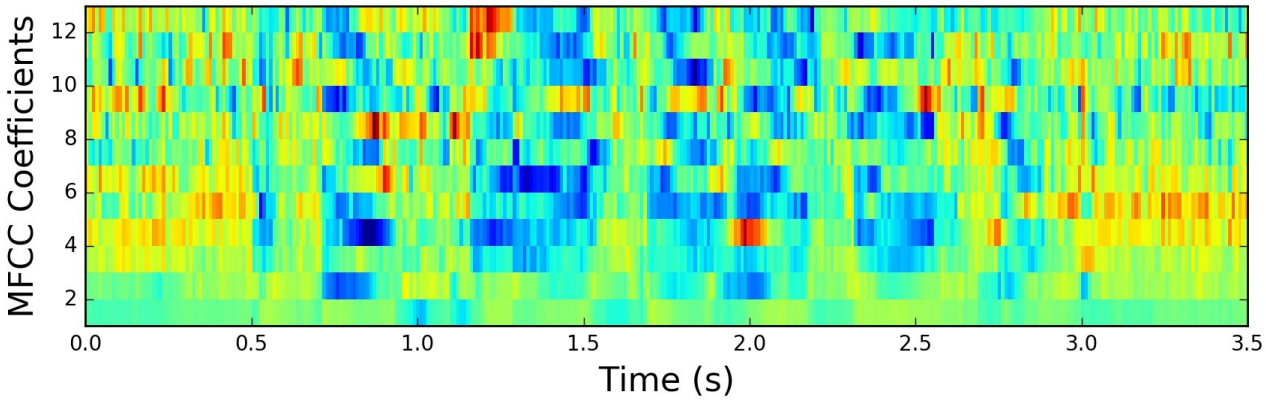


Normalized Filter Banks

and similarly for MFCCs:

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

The mean-normalized MFCCs:



Normalized MFCCs

Filter Banks vs MFCCs

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

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

Conclusion

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

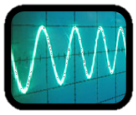
tl;dr: Use Mel-scaled filter banks if the machine learning algorithm is not susceptible to highly correlated input. Use MFCCs if the machine learning algorithm is susceptible to correlated input.

---

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

Discrete Cosine Transformations

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



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

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

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

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

JPEG

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

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

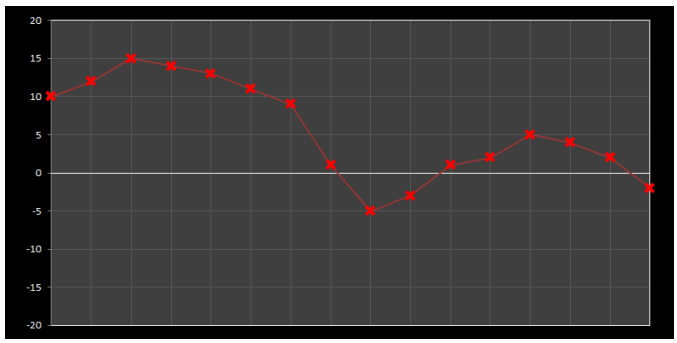
How is this compression achieved? By using a DCT transform, the image is shifted into the frequency domain. Then, depending on how much compression is required, the higher frequency coefficients of the signal are masked off and removed (the digital equivalent of applying a *low-pass analog filter*).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

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

Example: One Dimension

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

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

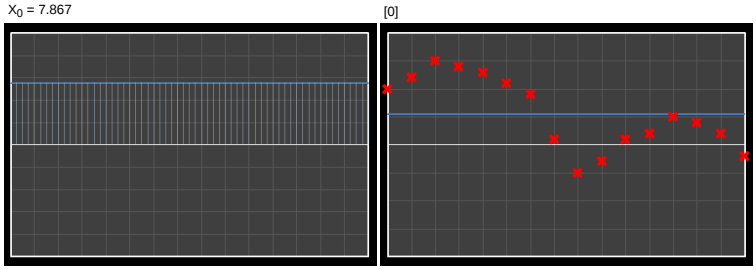


Using DCT, we can break this curve into a series of Cosine waves of various frequencies. It is by the superposition (adding together) of these fundamental waves that we recreate the original waveform.

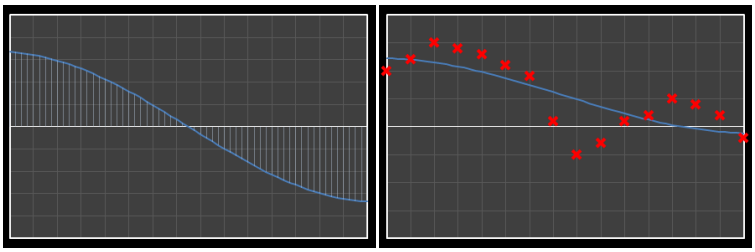
For the above sixteen points, I've broken down the data using DCT. The graphs on the left below show the cosine function of that frequency component and the image on the right shows the superposition of this to the running total (this component added to all those above it). The lowest frequency components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

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

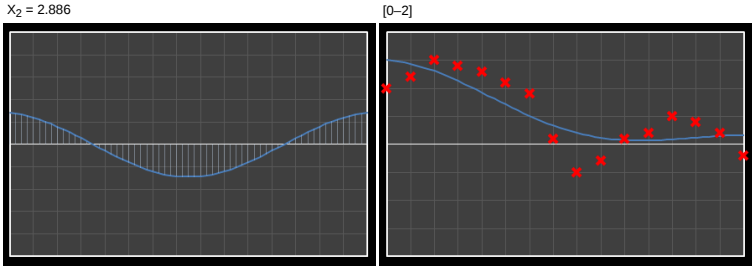
$X_0 = 7.867$ [0]



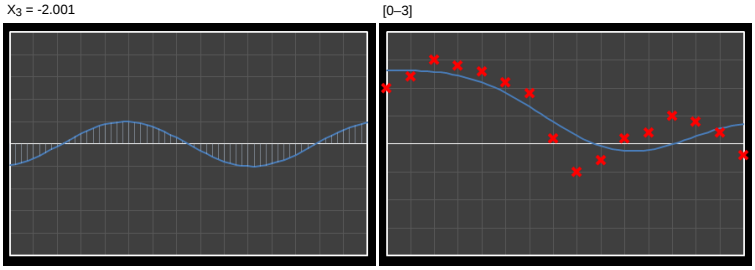
$X_1 = 6.742$ [0–1]



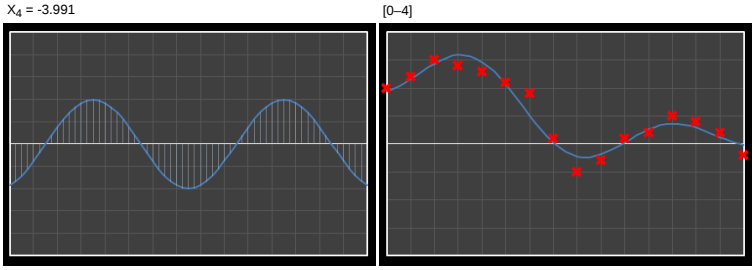
X_2 = 2.886 [0–2]



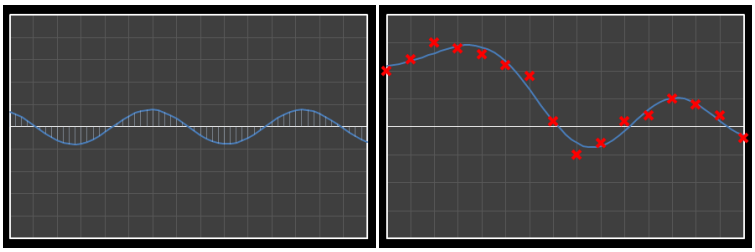
X_3 = -2.001 [0–3]



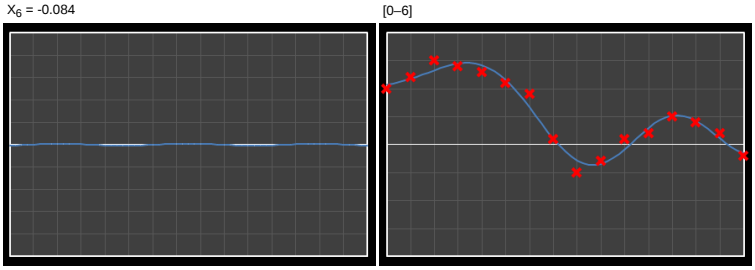
X_4 = -3.991 [0–4]



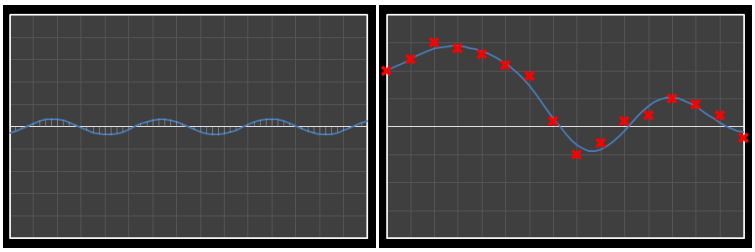
X_5 = 1.546 [0–5]



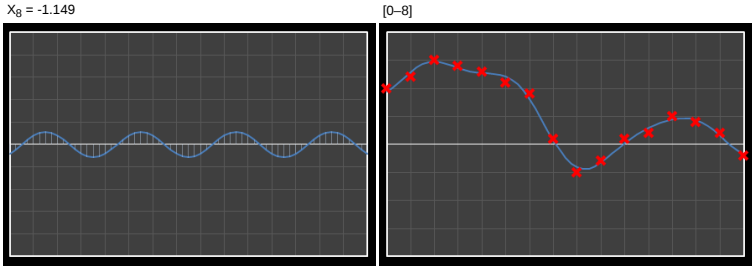
X_6 = -0.084 [0–6]



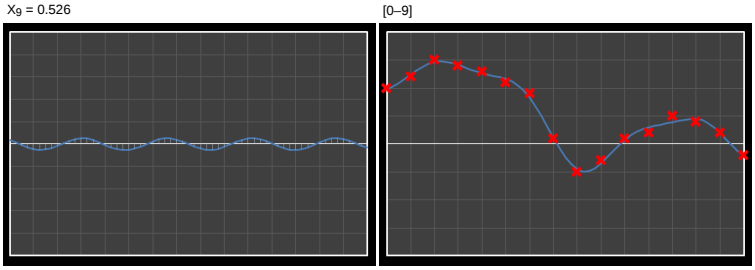
X_7 = -0.703 [0–7]



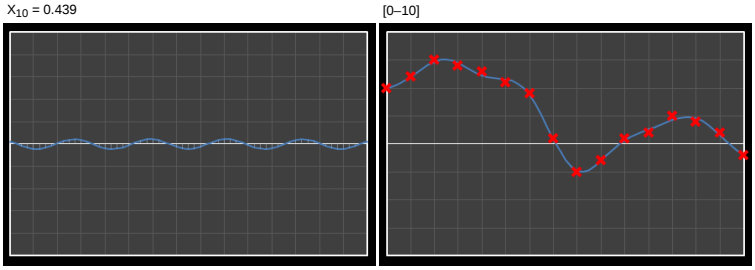
X_8 = -1.149 [0–8]



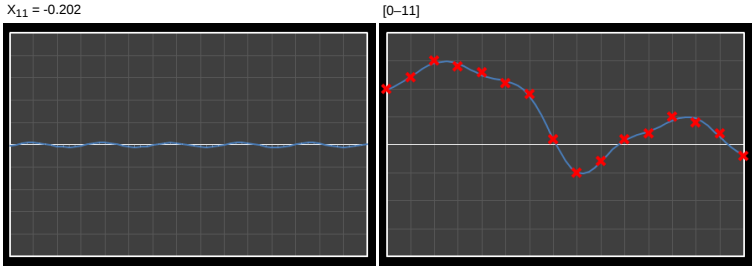
X_9 = 0.526 [0–9]



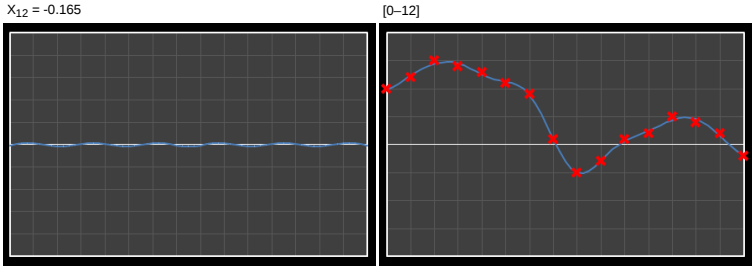
X_{10} = 0.439 [0–10]



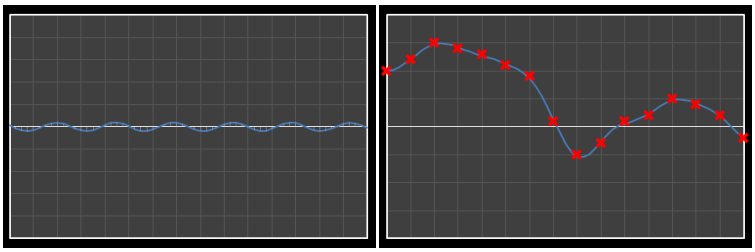
X_{11} = -0.202 [0–11]



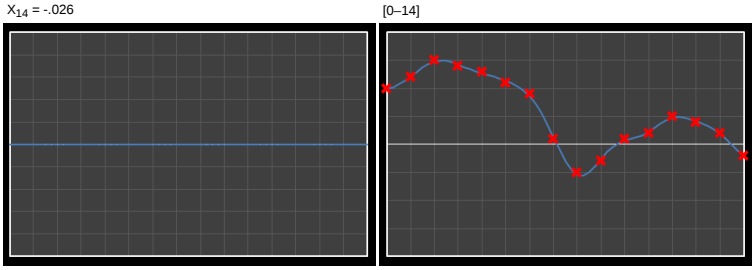
X_{12} = -0.165 [0–12]



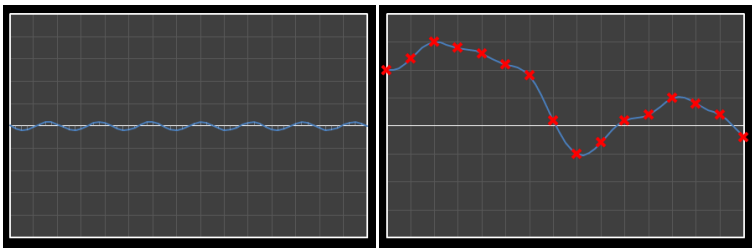
X_{13} = 0.382 [0–13]



X_{14} = -.026 [0–14]

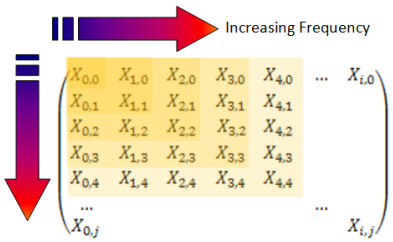


X_{15} = 0.368 [0–15]



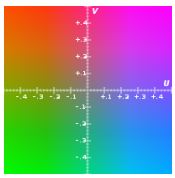
Example: Two Dimensions (and the basics of JPEG compression)

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



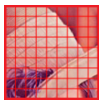
In JPEG compression, the block size used is 8 x 8, resulting in a similar size matrix of coefficients. However, for my examples, I'm going to select a larger block size of 16 x 16 (I think it is easier to visualize things at this size.)

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



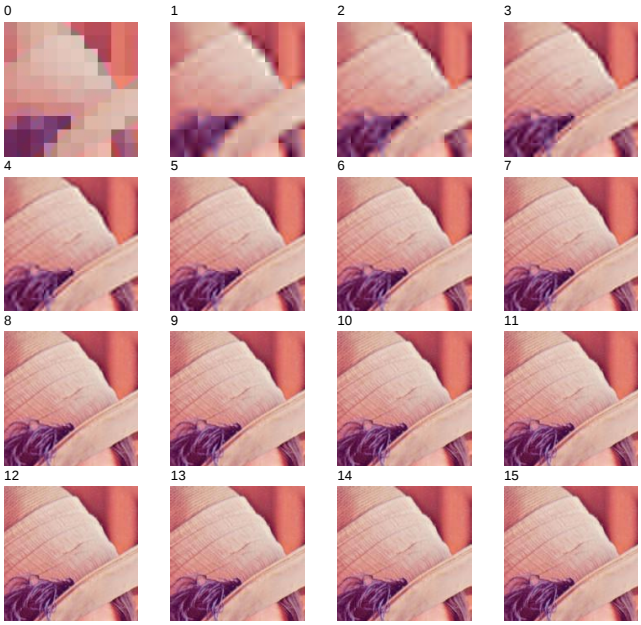
Our eyes are more sensitive to changes in brightness than changes in shade, and this can be exploited for more compression by converting RGB colors in YUV space, then UV channels can be sub-sampled (quantized) and reduced in dynamic range – Because of lower senstitivity you need *fewer distinct levels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

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



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

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

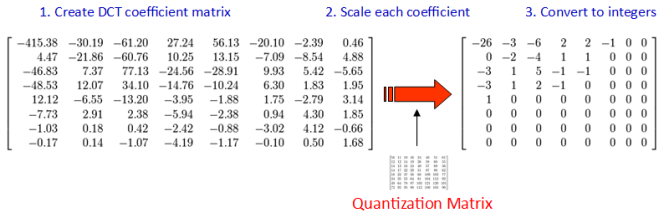


JPEG Advanced

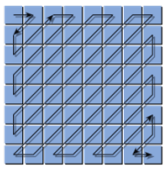
The above paragraphs explain the concept of how JPEG compression works (reduction of the higher frequency components), in reality, it is a little more complicated.

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

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



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



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

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

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

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

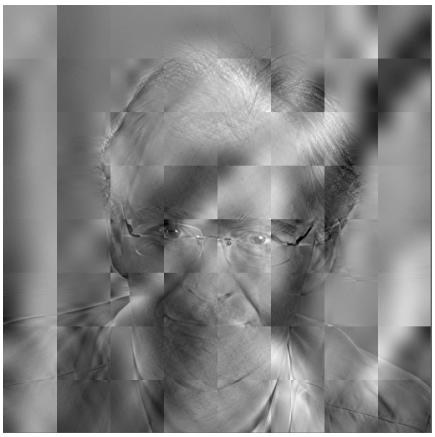
It is possible to merge two images with different spatial frequencies.

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

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

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

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



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

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

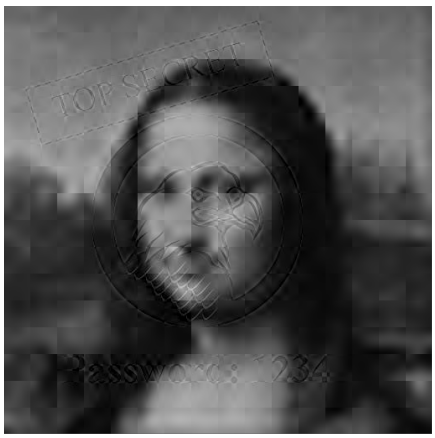
You can also experience the same effect by squinting at the hybrid image. Squinting artificially reduces the size of your pupil creating a Pinhole effect, increasing the depth of field for your eyes.

Hidden Letters

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

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

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

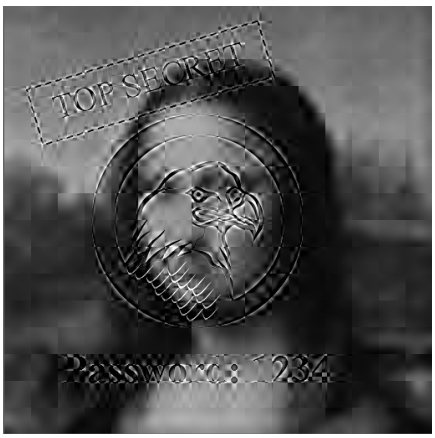


Need more convincing?

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

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

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



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

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