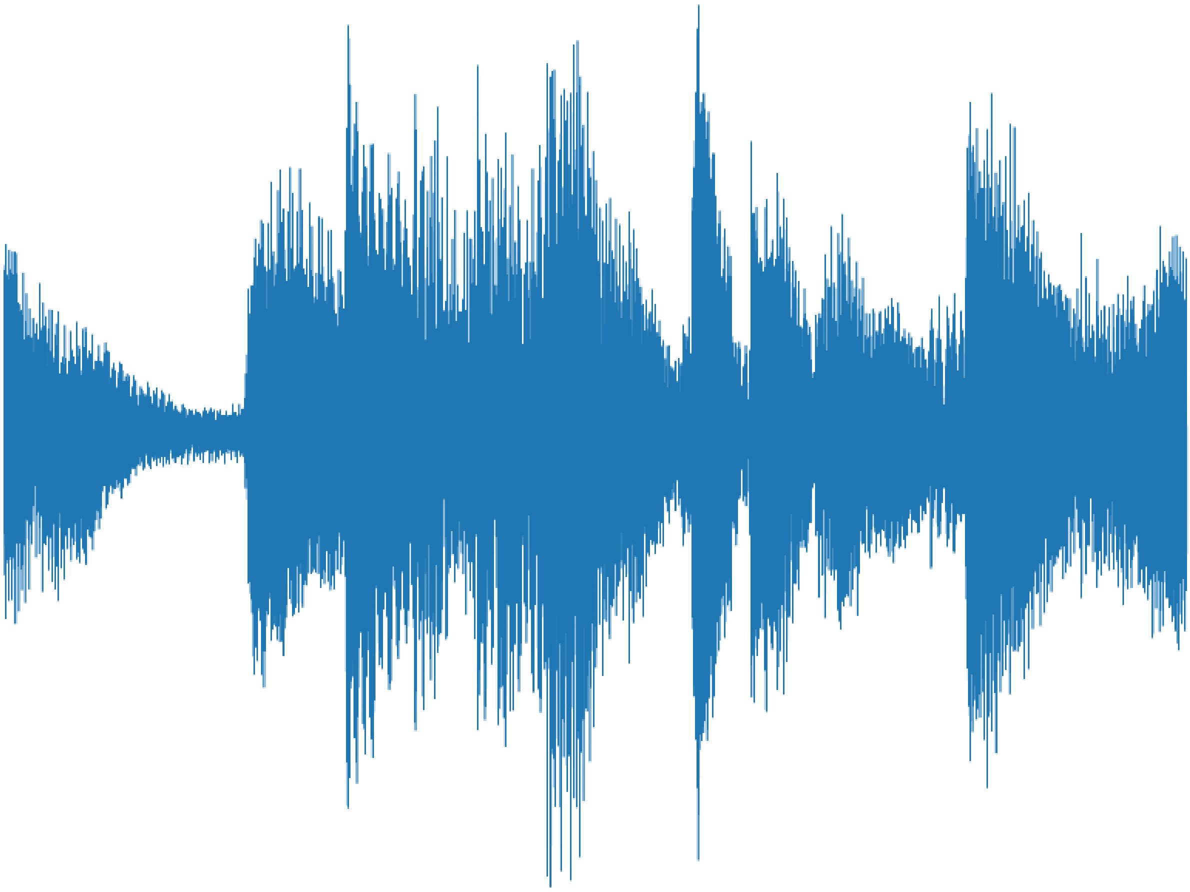
ECEN 4532 - Lab 4: MPEG Audio Signal Processing (MP3)

Andrew Teta

March 18, 2019



Contents

1 Introduction

In this lab, we explore signal processing in the context of audio compression. Specifically, we will be constructing the basis of the MPEG 1 Layer III codec, commonly known as MP3. The general idea is to implement a series of sub-band filters to decompose and reconstruct input audio ”perfectly” and later, experiment with the exclusion of certain frequency bands as a method of data compression. We will implement the polyphase pseudo-QMF filter bank, which filters the audio signal into 32 frequency bands. The two main procedures in this processing scheme are analysis and synthesis.

2 Background

2.1 Analysis

Analysis is the process of decomposing a signal into frequency components and filtering it into sub-bands. MP3 encoding splits the audio signal $x(n)$ into 32 equally spaced frequency bands. This is accomplished with 32 parallel filters with impulse response h_k such that,

$$s'_k(n) = h_k(n) * x(n), \quad k = 1, ..., 32. \tag{1}$$

By downsampling the output of each filter by a factor of 32, we achieve a critically sampled analysis filter. Thus,

$$s_k(n) = ((h_k * x) \downarrow 32)(n) = s'_k(32n). \tag{2}$$

The effect of this procedure (and the idea of critical sampling) is an efficient filter, where for an input block of size N samples of $x(n)$, a total of N output samples are produced.

2.2 Synthesis

Reconstruction is performed by upsampling the downsampled signals with zero values and filtering with a synthesis filter bank, g_k.

$$s_k \uparrow 32) * g_k(n), \quad k = 1, ..., 32. \tag{3}$$

The synthesized output, $\tilde{x}$ is found by the sum of all the synthesis filters,

$$\tilde{x} = \sum_{k=1}^{32} (s_k \uparrow 32) * g_k(n). \tag{4}$$

The MP3 filter bank has nearly (but not exactly) perfect reconstruction and fortunately the error is not an issue. The filter design is greatly simplified by using a prototype filter, modified only slightly for each of the 32 analysis filters, h_k.

3 Cosine Modulated Pseudo Quadrature Mirror Filter: Analysis

This section describes the mathematical description of MP3 analysis and further derives a fast algorithm to compute both convolution and decimation together, producing the output signal s_k for each of the 32 sub-bands.

3.1 The math

Consider a 512 tap filter h_k such that

$$s_k(n) = \sum_{m=0}^{511} h_k(m)x(32n - m) \tag{5}$$

and

$$h_k(m) = p_0(m)cos\left(\frac{(2k+1)(m-16)\pi}{64}\right) \quad k = 0, ..., 31, \quad m = 0, ..., 511. \tag{6}$$

p_0 is a prototype lowpass filter (see Fig. 1)and the multiplication with $cos\left(\frac{(2k+1)(m-16)\pi}{64}\right)$ shifts the center frequency of p_0 to be centered around $(2k+1)\pi/64$. Thus, h_k becomes a bandpass filter, selecting frequencies around $(2k+1)\pi/64$, for $k = 0, ..., 31$, and a nominal bandwidth of $\pi/32$.

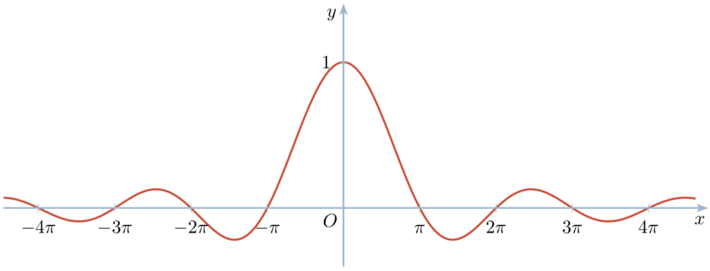


Figure 1: A cartoon representation of p_0, the prototype filter used for analysis of a signal in MP3 encoding. This plot has no relevance to scale and is only a reference to the general shape of p_0 and its lowpass filter construction.

3.2 Derivation of a faster way

We begin with (5) and decompose the summation by letting $m = 64q + r$

$$s_k(n) = \sum_{m=0}^{511} h_k(m)x(32n - m) = \sum_{q=0}^{7}\sum_{r=0}^{63} h_k(64q + r)x(32n - 64q - r) \tag{7}$$

and since

$$h_k(m) = p_0(m)cos\left(\frac{(2k+1)(m-16)\pi}{64}\right), \tag{8}$$

we can write

$$\begin{aligned} h_k(64q + r) &= p_0(64q + r)cos\left(\frac{(2k+1)(64q - r - 16)\pi}{64}\right) && (9) \\ &= p_0(64q + r)cos\left(\frac{(2k+1)(r-16)\pi}{64} + (2k+1)q\pi\right) && (10) \\ &= \begin{cases} p_0(64q + r)cos\left(\frac{(2k+1)(r-16)\pi}{64}\right) & \text{if } q \text{ is even} \\ -p_0(64q + r)cos\left(\frac{(2k+1)(r-16)\pi}{64}\right) & \text{if } q \text{ is odd} \end{cases} && (11) \end{aligned}$$

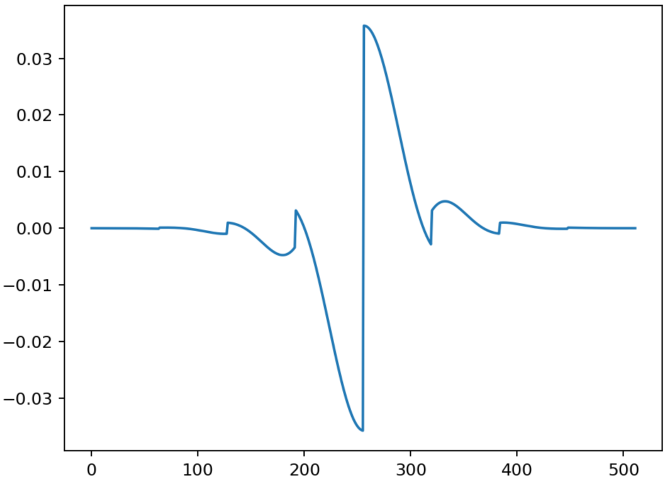


Figure 2: Plot of a 512 tap analysis filter, C, used in MP3 encoding.

Let us define (see Fig. 2),

$$c(m) = \begin{cases} p_0(m) & \text{if } q = \lfloor m/64 \rfloor \text{ is even} \\ -p_0(m) & \text{if } q = \lfloor m/64 \rfloor \text{ is odd,} \end{cases} \tag{12}$$

then

$$h_k(64q + r) = c(64q + r)cos\left(\frac{(2k+1)(r-16)\pi}{64}\right). \tag{13}$$

Using the notation of the MP3 standard, we define

$$\boxed{M_{k,r} = cos\left(\frac{(2k+1)(r-16)\pi}{64}\right), \quad k = 0, ..., 31, \quad r = 0, ..., 64,} \tag{14}$$

so,

$$h_k(64q + r) = c(64q + r)M_{k,r}, \tag{15}$$

and

$$s_k(n) = \sum_{q=0}^{7}\sum_{r=0}^{63} M_{k,r}c(64q + r)x(32n - 64q - r) \tag{16}$$

Expressing the analysis in the form of (16), we can efficiently compute sub-band sample coefficients. For every n, we compute:

$$z(64q + r) = c(64q + r)x(32n - 64 - r), \quad r = 0, ..., 63, \quad q = 0, ..., 7, \tag{17}$$

sum over q

$$y(r) = \sum_{q=0}^{7} z(64q + r), \quad r = 0, ..., 63, \tag{18}$$

and compute one sample output for each sub-band by taking a sum over r

$$s_k = \sum_{r=0}^{63} M_{k,r}y(r), \quad k = 0, ..., 31. \tag{19}$$

3.3 Frequency inversion

Downsampling has the effect of moving the output of each filter down to the baseband. However, the periodic nature of the discrete-time Fourier transform (DTFT) representation, causes a ”frequency inversion” to occur, where higher frequency components appear to be to the left of the y-axis. This can be illustrated visually in Fig. 3.

Considering the first sub-band filter to be index 0, then as it works out, the *odd* sub-bands are inverted. The MP3 codec further decomposes the filter bank output using the modified discrete cosine transform (MDCT), so it is desirable to undo the inversion. In complex exponential form, this can be accomplished by multiplying the signal by $e^{j\pi n}$, but in Python and other analysis tools where only the exponential coefficients are considered, we can simply multiply the sample output of every odd sub-band by -1. This operation has the effect of translating the spectral signal by π, restoring its orientation because the DTFT is periodic in 2π.

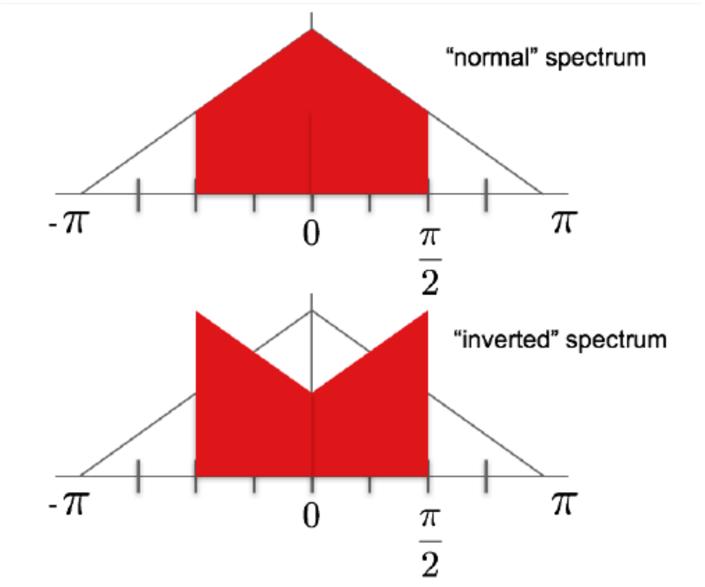


Figure 3: Spectrum inversion

3.4 Implementation

Our analysis was done in Python, however the basic procedure of implementation will be the same for any implementation.

Each processing cycle works on packets of 32 audio samples and the filtering is performed on a buffer X of size 512. The buffer is shifted to the right by 32 samples and the available indexes at the left end of the buffer are filled with new audio samples. 32 sub-band coefficients are computed and this process is repeated until the length of the input signal is reached.

In this lab, the window filter coefficients, c, were calculated prior and provided in the form of a text file. The 512 coefficients were read into a `numpy` array at the beginning of analysis to be used in calculation later. Audio was provided as `.wav` tracks. We used `scipy.io.wavfile.read()` to convert these files to `numpy` arrays. Further analysis was performed on the first 5 seconds of each track.

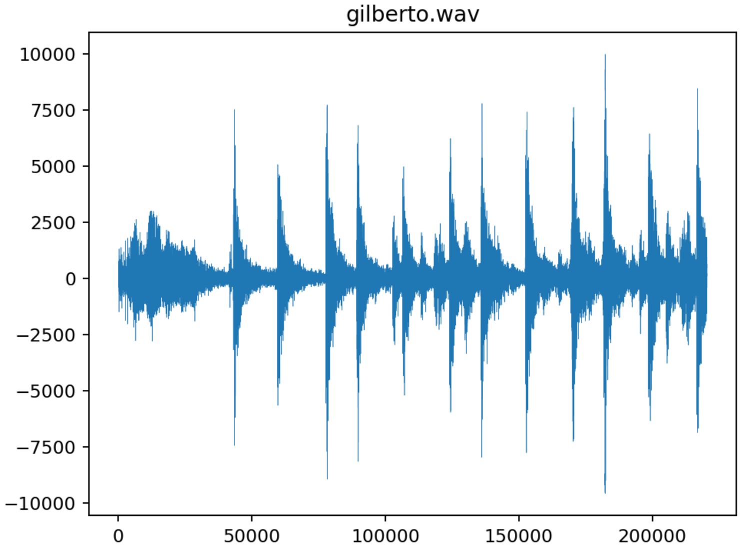


Figure 4: Plot of samples vs. magnitude for first 5 seconds of a jazz song, used as input to our analysis.

We begin calculation of sub-band coefficients by reshaping the audio sample array to have dimension (-1,32), where -1 means the first dimension is implied. I implemented a quick calculation to append zeros to the end of the array if it did not have the correct number of elements to be reshaped directly. Before beginning to loop over the rows of this new matrix of audio samples, we can build M one time to save processing cycles. A temporary buffer, X of length 512, an output matrix of `np.zeros_like` the (-1,32) input sample data, and a couple helper vectors which allow vector operations instead of sums, are all declared outside of the loop. One helper vector, `fInvert` is used for frequency inversion correction.

$$fInvert = [1, -1, 1, \dots, 1] \quad \text{length} = 32$$

As well as `zFlat`, a length 8 vector of ones which replaces the sum in (18).

Now for the actual analysis. We loop over each row, effectively performing operations on a "packet" of 32 samples for each loop iteration. For each packet,

1. Shift every value in buffer X right by 32
2. Fill X[0:32] with packet (flipped to perform convolution)
3. Compute $Z = C * X$, windowing X by the analysis filter taps (eq. (17))
4. Reshape Z into (8,64) and calculate $Y = zFlat \cdot Z$ (eq. (18))
5. Compute $S = M \cdot Y$ (eq. (19))
6. Store length 32 output vector, S in a row of output matrix

The output of this sequence should be a matrix of sub-band coefficients organized like so (input matrix of shape (N,32)):

$$A = \begin{bmatrix} sb_{0,0} & sb_{0,1} & sb_{0,2} & \dots & sb_{0,31} \\ sb_{1,0} & sb_{1,1} & sb_{1,2} & \dots & sb_{1,31} \\ \dots & & & & \\ sb_{N,0} & sb_{N,1} & sb_{N,2} & \dots & sb_{N,31} \end{bmatrix} \tag{20}$$

and will also have shape (N,32).

Transposing and flattening A, we can plot the sub-band analysis output.

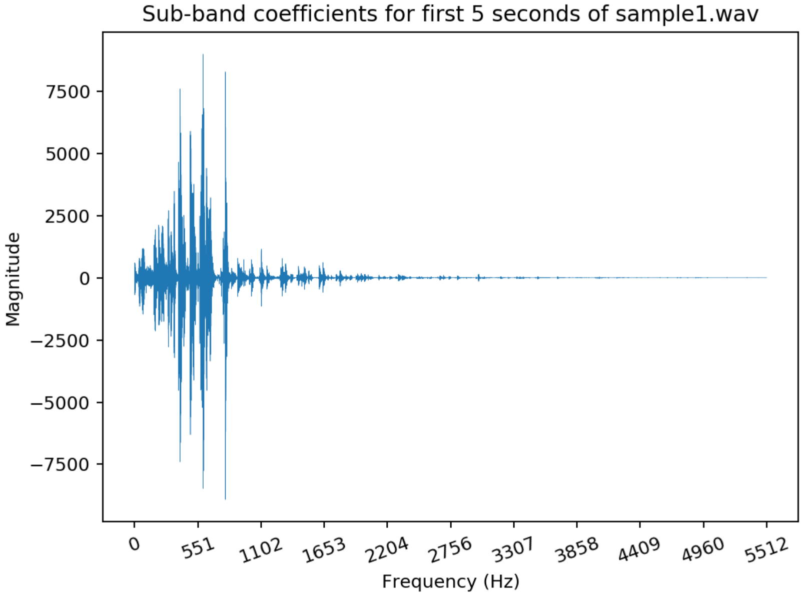


Figure 5: Spectral content of a piano melody. We see a small magnitude of very low frequencies, most of the energy concentrated in the low-mid frequency range, and almost no content at high frequency.

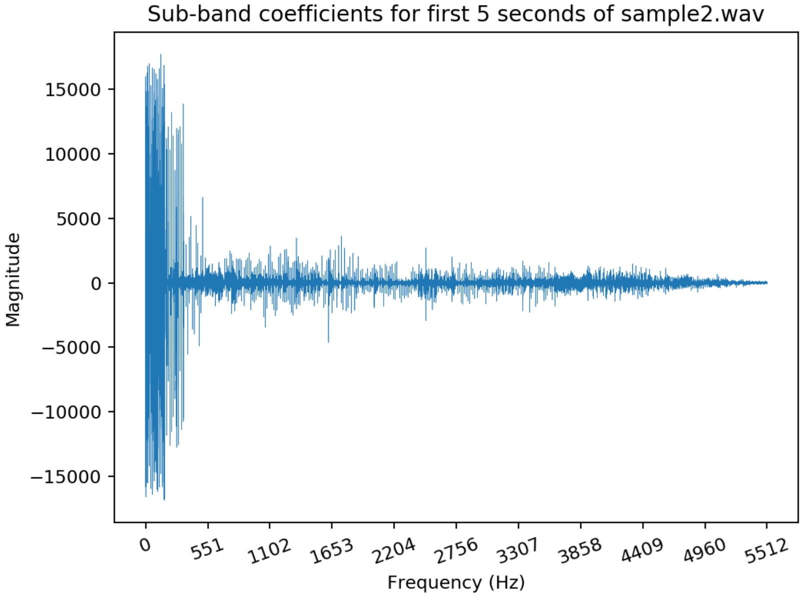


Figure 6: Spectral content of a dance floor beat. We see high energy density at very low frequencies and lots of noise at higher frequencies.

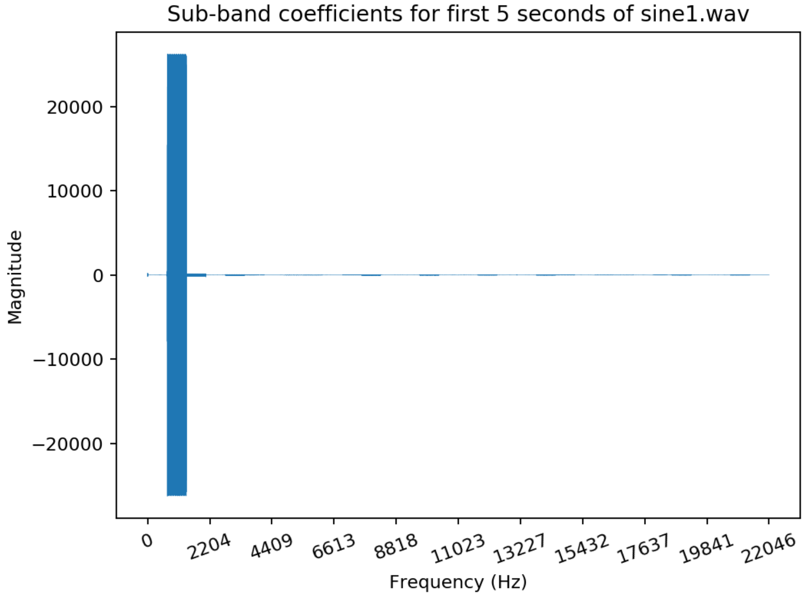


Figure 7: Spectral content of a sine wave. We see energy only between about 750 Hz and 1250 Hz.

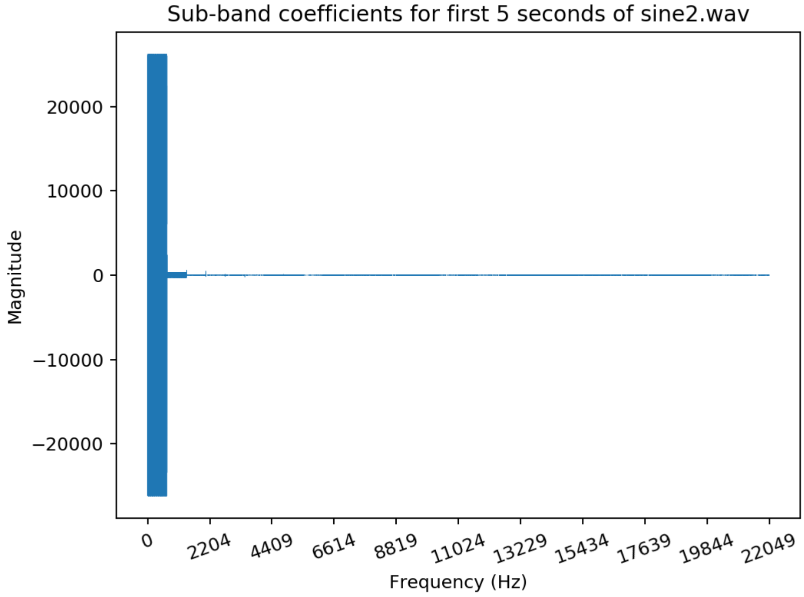


Figure 8: Spectral content of a sine wave. We see energy only between 0 Hz and 750 Hz.

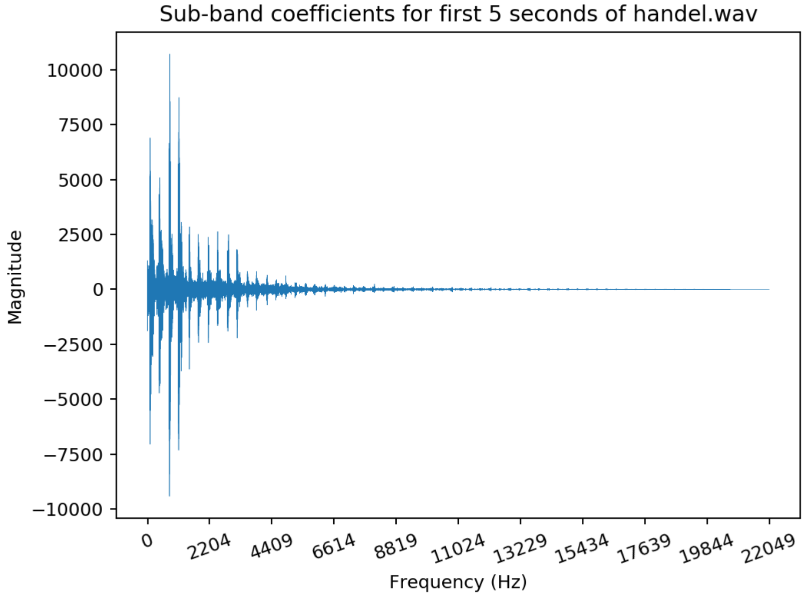


Figure 9: Spectral content of a classical track. There is good contribution from spectral bands at 0 Hz all the way up to about 5 kHz.

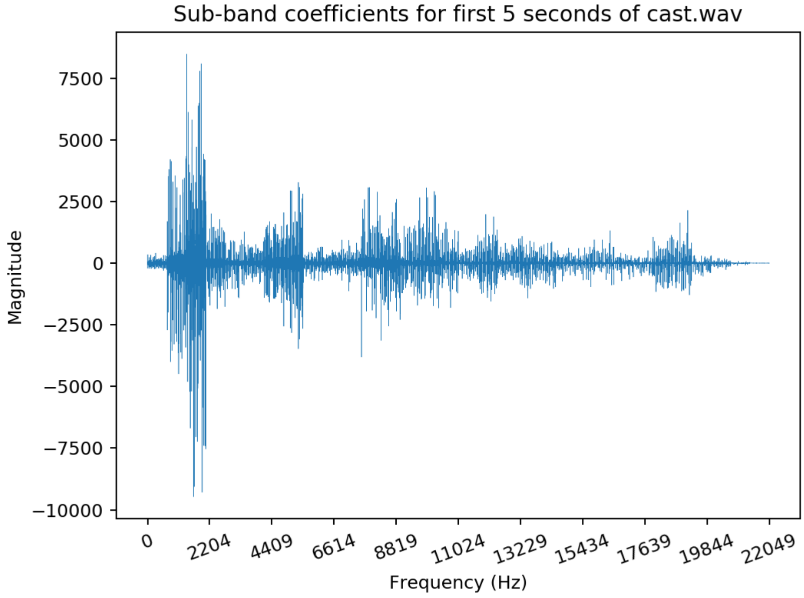


Figure 10: Spectral content of a percussive recording. The energy is distributed fairly evenly across the frequency spectrum here.

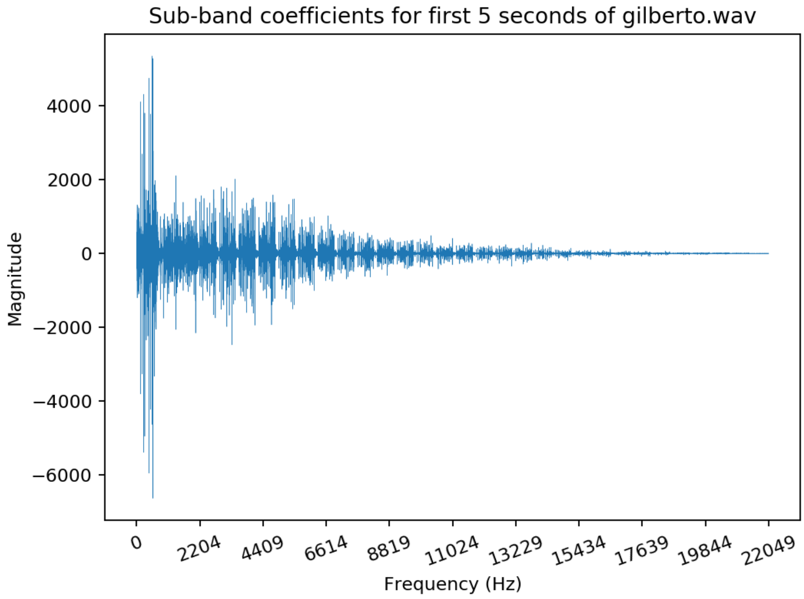


Figure 11: Spectral content of a jazz track. The first five seconds are very percussion heavy, most likely leading to the large spread of spectral content. Above about 16 kHz there is mostly noise.

4 Reconstruction: Synthesis

Synthesis, or reconstruction of the MP3 codec is very similar to analysis. For each set of 32 sub-band coefficients, 32 audio samples are reconstructed. The algorithm will need to reverse the frequency inversion correction, so we will actually invert the signal again.

This time, we will perform modulation with a matrix N where,

$$N_{i,k} = cos\left(\frac{(2k+1)(16+i)\pi}{64}\right), \quad i = 0, ..., 63, \quad k = 0, ..., 31. \tag{21}$$

First, we load reconstruction filter tap coefficients from a text file to obtain the window in Fig. 12

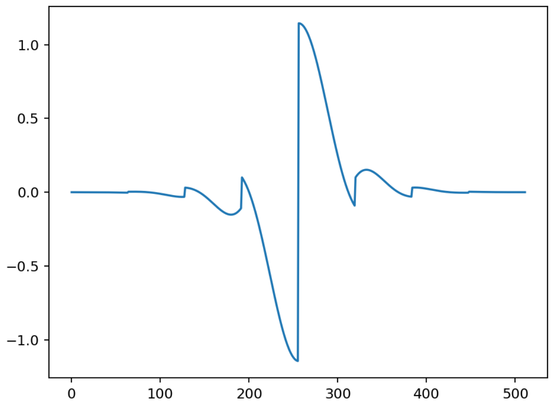
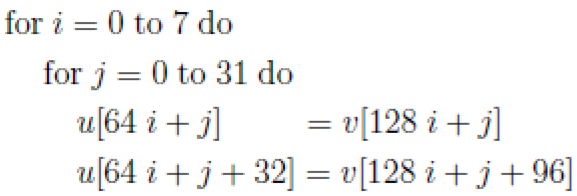


Figure 12: Plot of reconstruction window D.

We again operate on "packets" of length 32 samples and loop over vectors of this length through to the end of the track. This time our buffer vector V is of length 1024, we declare length 512 vectors U and W, an output vector S and two more helping vectors, `fInvert` as before and a length 16 vector of ones, `wFlat`.

We will build U during each cycle and use it to compute the reconstruction. V is constructed over multiple packet cycles and so the content of U is also updated. We define



```
for $i = 0$ to 7 do
    for $j = 0$ to 31 do
        $u[64\,i + j] \quad\quad\;\; = v[128\,i + j]$
        $u[64\,i + j + 32] = v[128\,i + j + 96]$
```

The steps in reconstruction go as follows:

1. Shift every value in a buffer V right by 64
2. Calculate modulation and re-invert: $V[0:64] = N \cdot (fInvert * packet)$
3. Compute U as shown above
4. Window the samples vector by $W = U * D$
5. Reshape W into (16,-1) and store $wFlat \cdot W$ in output matrix

Plotting the output of this reconstruction procedure, we can compare it to the original signal.

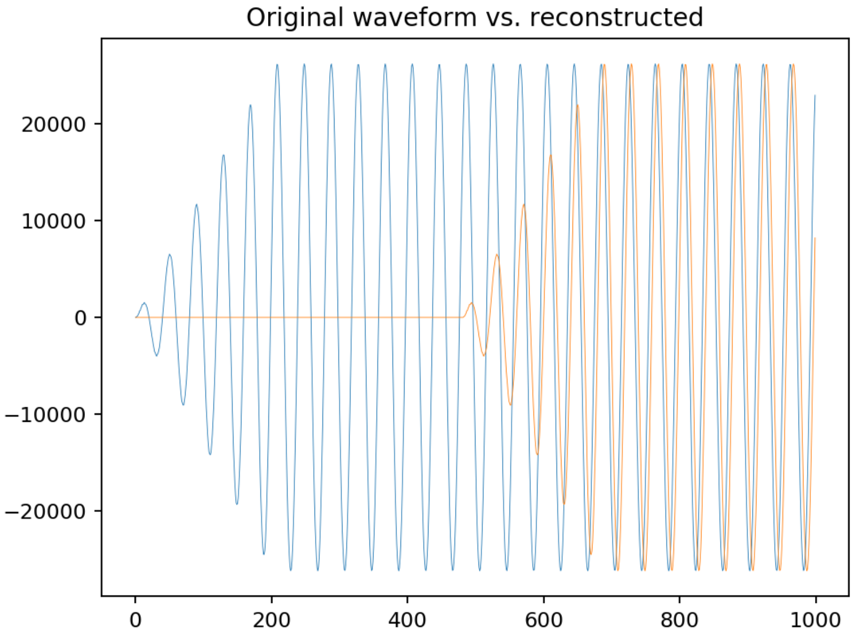


Figure 13: Plot of first 1000 samples of a simple sine wave input signal, sine1.wav. Observe the reconstructed signal is delayed by some amount.

Through trial and error, attempting to minimize the error between the original signal and reconstruction, I found the delay to be equal to 481 samples. This is caused by having an empty buffer when we begin synthesis. The it is only after 15 packets of information, that the buffer is full (with 512 values) and so the first 15 cycles produce a delay of available information. We start with 32 samples and index from 0, thus we have a delay of 481 samples.

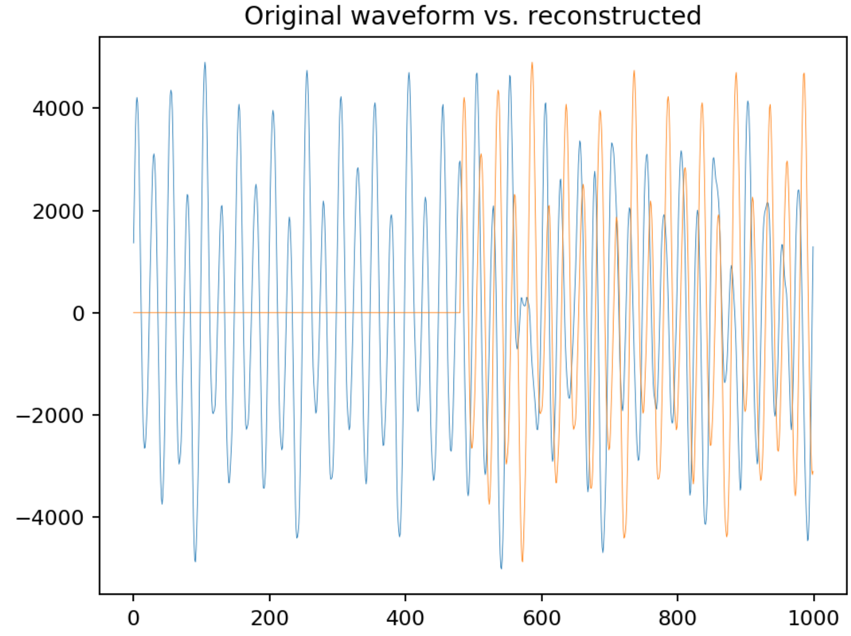


Figure 14: Reconstruction and delay shown for sample1.wav

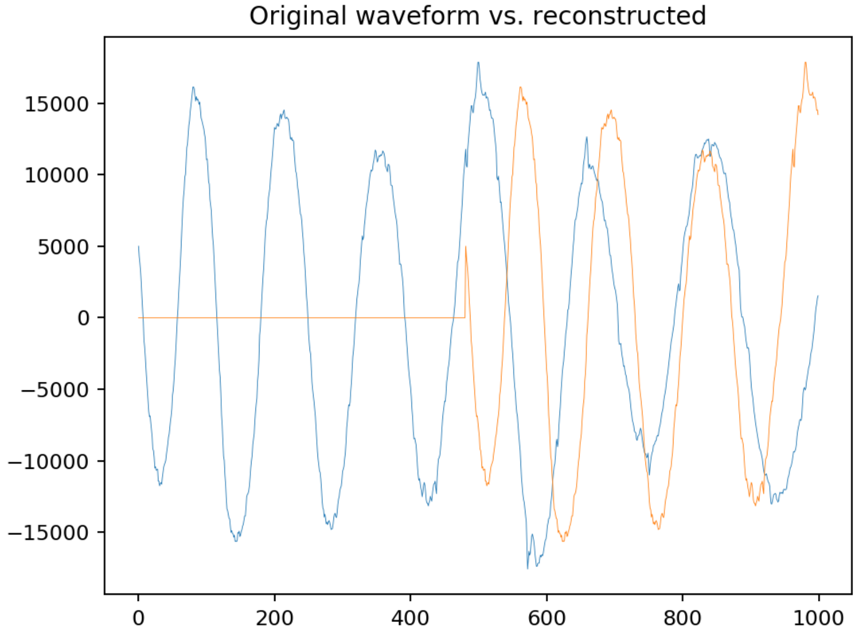


Figure 15: Reconstruction and delay shown for sample2.wav

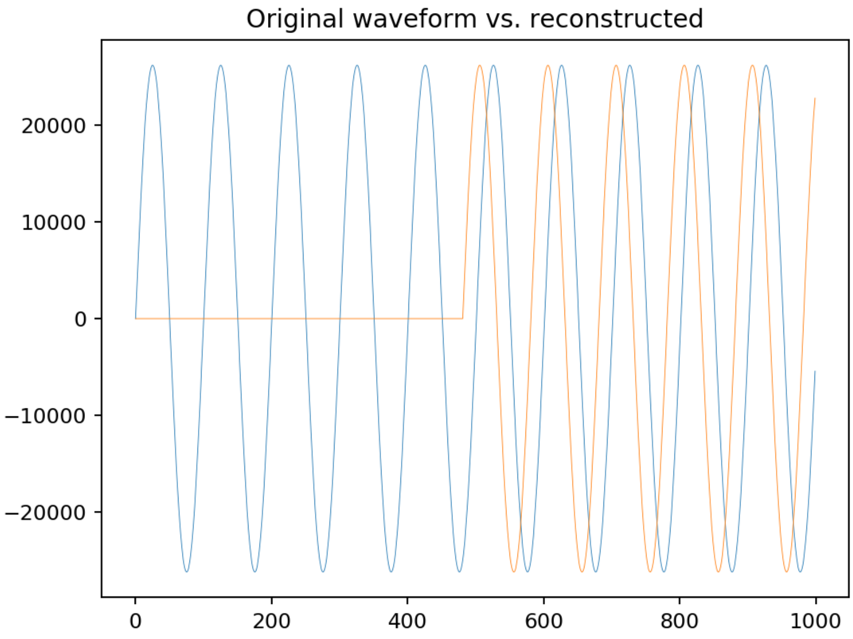


Figure 16: Reconstruction and delay shown for sine2.wav

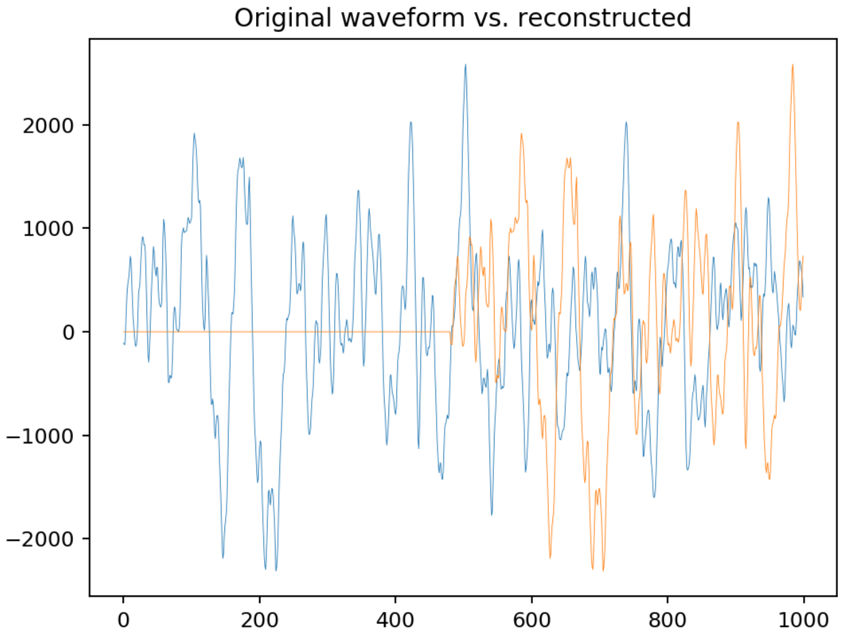


Figure 17: Reconstruction and delay shown for handel.wav

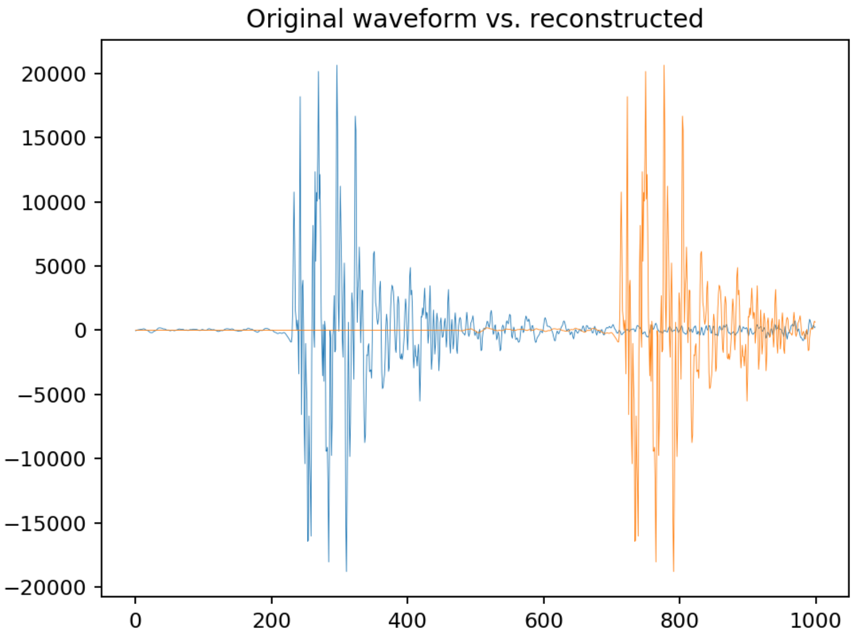


Figure 18: Reconstruction and delay shown for cast.wav

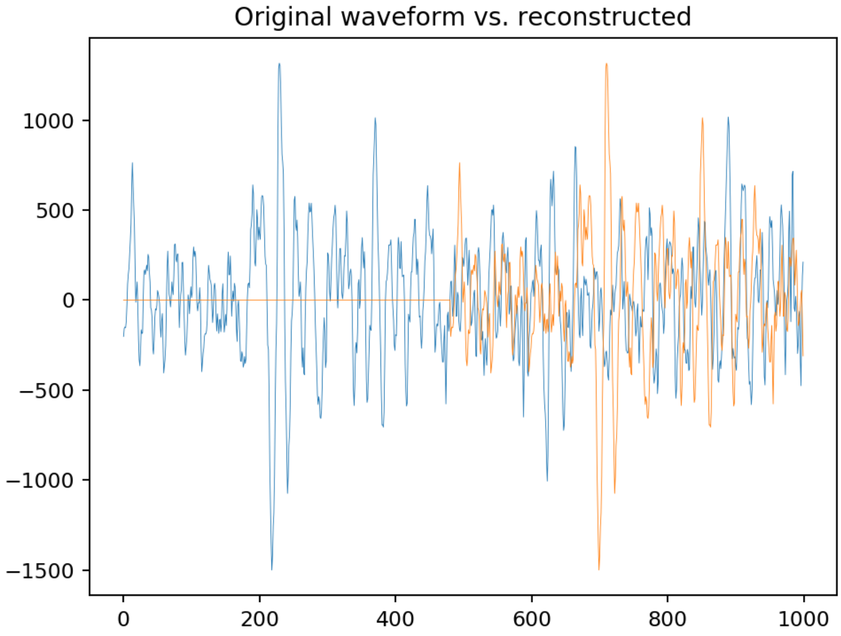


Figure 19: Reconstruction and delay shown for gilberto.wav