University of Colorado at Boulder ECEN 4532

Instructor: Mike Perkins

MPEG Audio Signal Processing (MP3)

Lab 4
Lab report due on March 18, 2019

This is a Python only lab. Each student must turn in their own lab report and programs.

1 Introduction

This lab focuses on the signal processing required to implement the MPEG 1 Layer III audio codec, also known as MP3. In this lab you will implement several sub-band filters to decompose and reconstruct the original audio signal. There are two types of filtering: one sub-divides the frequency axis into critically sampled sub-bands, while the other decomposes the output of each critically sampled filter using sinusoidal functions (a transform similar to the DFT).

We will implement the polyphase pseudo-QMF filter bank that breaks the audio signal into 32 frequency bands. You will write code that decomposes and reconstructs the audio signal, and investigate the effects of leaving out certain sub-bands when synthesizing the output signal.

1.1 Analysis filter bank

The MP3 encoding scheme first splits the audio signal $x(n)$ into 32 equally spaced frequency bands using a bank of 32 "analysis" filters with impulse response h_k,

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

The output of each analysis filter is then critically sampled by downsampling by a factor of 32,

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

As a result, each block of size N samples in the incoming signal, $x(n)$, produces a total of N otuput samples (this is the meaning of critically sampled: the net number of samples is not changed by this processing).

1.2 Synthesis filter bank

The reconstruction is performed by first upsampling the downsampled signals, s_k, (that is inserting 31 zeros between the samples),

$$(s_k \uparrow 32)(n), \tag{3}$$

and then filtering with "synthesis" filters, g_k,

$$(s_k \uparrow 32) * g_k(n), \quad k = 1, \ldots, 32, \tag{4}$$

Finally, the output of each synthesis filter is summed to create the reconstructed signal $\tilde{x}[n]$,

$$\tilde{x}(n) = \sum_{k=1}^{32} (s_k \uparrow 32) * g_k(n). \tag{5}$$

In the case of MP3, the filter bank has nearly (but not exactly) perfect reconstruction: the magnitude of the Fourier transform of the reconstructed signal is slightly attenuated at certain frequencies. In practice, this is not a problem. The advantage of not requiring perfect reconstruction is that the filter design is greatly simplified: the same prototype filter is used for each of the 32 analysis filters, h_k.

2 Cosine modulated pseudo Quadrature Mirror Filter: "analysis"

In this section we give the detailed mathematical description of the analysis filter banks. We will manipulate the equations that combine convolution and decimation to arrive at a fast algorithm to compute the output signal, s_k, for each of the 32 subbands. In section 2.1 we further describe the algorithmic implementation.

Consider the 512 tap filter h_k; the output of the combined filtering and decimation for this filter is given by

$$s_k(n) = \sum_{m=0}^{511} h_k(m) x(32n - m), \tag{6}$$

where

$$h_k(m) = p_0(m) \cos\left(\frac{(2k+1)(m-16)\pi}{64}\right) \quad k = 0, \ldots, 31, \quad m = 0, \ldots, 511, \tag{7}$$

and p_0 is a prototype lowpass filter (see Fig. 1). The frequency response of h_k is clear: the modulation of $p_0(m)$ by $\cos\left(\frac{(2k+1)(m-16)\pi}{64}\right)$ shifts the lowpass filter's frequency response to be centered around the frequency $(2k+1)\pi/64$, and thus h_k will become a bandpass filter that selects frequencies around $(2k+1)\pi/64$, for $k = 0, \ldots 31$, with a nominal bandwidth of $\pi/32$, (see Fig. 1).

Equation (6) requires $32 \times 512 = 16,384$ combined multiplications and additions to compute the 32 outputs $s_1, \ldots, s_{32}$ for each block of 32 samples of the incoming signal x. In the following, we develop a faster algorithm to compute this operation.

We start with (6) and decompose the summation as follows (think of this as letting $m = 64q + r$):

$$s_k(n) = \sum_{m=0}^{511} h_k(m)x(32n - m) = \sum_{q=0}^{7} \sum_{r=0}^{63} h_k(64q + r)x(32n - 64q - r) \tag{8}$$

Next, we observe that since

$$h_k(m) = p_0(m) \cos\left(\frac{(2k + 1)(m - 16)\pi}{64}\right) \tag{9}$$

we can write

$$\tag{10}$$

$$h_k(64q + r) = p_0(64q + r) \cos\left(\frac{(2k + 1)(64q + r - 16)\pi}{64}\right) \tag{11}$$

$$= p_0(64q + r) \cos\left(\frac{(2k + 1)(r - 16)\pi}{64} + (2k + 1)q\pi\right) \tag{12}$$

$$= \begin{cases} p_0(64q + r) \cos\left(\frac{(2k+1)(r-16)\pi}{64}\right) & \text{if } q \text{ is even} \\ -p_0(64q + r) \cos\left(\frac{(2k+1)(r-16)\pi}{64}\right) & \text{if } q \text{ is odd} \end{cases} \tag{13}$$

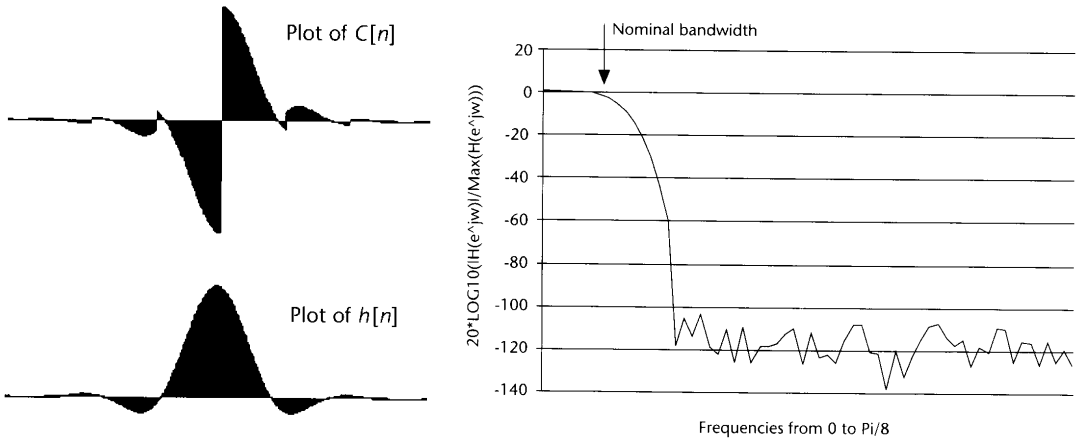

Figure 1: Left: prototype filter $p_0(m)$, (in the diagram above it is the bottom graph on the left and is labelled $h(n)$). Right: the frequency response of the prototype filter p_0.

Let us define (see Fig. 1),

$$c(m) = \begin{cases} p_0(m) & \text{if } q = \lfloor m/64 \rfloor \text{ is even} \\ -p_0(m) & \text{if } q = \lfloor m/64 \rfloor \text{ is odd,} \end{cases} \tag{14}$$

then

$$h_k(64q + r) = c(64q + r) \cos\left(\frac{(2k+1)(r-16)\pi}{64}\right) \tag{15}$$

Using the notations of the standard, we further define

$$\boxed{M_{k,r} = \cos\left(\frac{(2k+1)(r-16)\pi}{64}\right), \quad k = 0, \ldots, 31,\ r = 0, \ldots, 63,} \tag{16}$$

then

$$h_k(64q + r) = c(64q + r) M_{k,r}, \tag{17}$$

and

$$s_k(n) = \sum_{q=0}^{7} \sum_{r=0}^{63} M_{k,r} c(64q + r) x(32n - 64q - r) \tag{18}$$

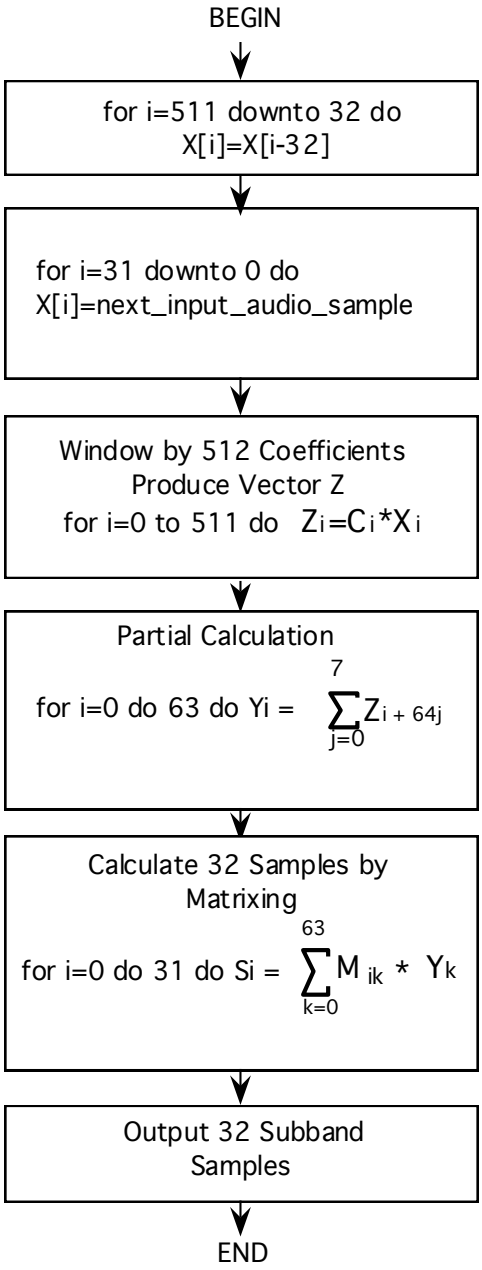

Figure 2: Encoding flow chart: the buffer X receives 32 new audio data samples, and 32 subband coefficients $S_0, \ldots, S_{31}$ are computed.

As a result of expressing the convolution of equation (6) in the form of equation (18), we can efficiently compute the sub-band samples. For every n, we use the following three steps:

First, compute

$$z(64q + r) = c(64q + r)x(32n - 64q - r), \quad r = 0, \ldots, 63, q = 0, \ldots, 7 \tag{19}$$

Next, sum out the dependency on q via

$$y(r) = \sum_{q=0}^{7} z(64q + r), \quad r = 0, \ldots, 63; \tag{20}$$

Finally, compute one sample output for each subband via

$$s_k = \sum_{r=0}^{63} M_{k,r}\, y(r), \quad k = 0, \ldots, 31. \tag{21}$$

This algorithm is described in the block diagram in Fig. 2. In its most naive implementation, this algorithm requires $512 + 32 \times 64 = 2,560$ multiplications, and $64 \times 7 + 32 \times 63 = 2,464$ additions. Further speedup can be obtained by using a fast DCT algorithm to compute the matrix-vector multiplication in 21.

Note that the output $s_k(n)$ is maximally decimated: for every 32 new input samples, the filter bank generates 32 outputs $s_1(n), \ldots s_{32}(n)$. In other words, for each of the 32 sub-bands k, there is a single output for every 32 new input samples.

2.1 Implementation of the analysis filter bank

Figure 2 shows the flow chart of the filtering of the audio data. The filtering is based on the following ideas.

1. Each processing cycle works on a packet of 32 audio samples.
2. The filtering is performed on a buffer X of size 512, using equations (19)-(21).
3. During each processing cycle, the buffer X is shifted to the right, and samples $X[480:]$ are discarded. This leaves 32 empty slots to fill with new data: the 32 new audio samples are saved in $X[:32]$.
4. 32 subband coefficients, $s_i, i = 0, \ldots 31$ are computed using (21).
5. At startup, the buffer X is filled with zeros. After processing 512 audio samples, the buffer X is always full.

We assume that we only process monophonic audio and that we do so in frames. Each frame will contain $576 = 18 \times 32$ audio samples. The output of a frame will be a sequence of 18 vectors of sub-band coefficients, where each vector has size 32. The following Python code provides a skeleton for the subband filtering in layer III.

```
# Let audio be a numpy array holding the audio samples
...
frameSize = 576
nFrames = np.floor(len(audio) / frameSize);

# chunk the audio into blocks of 576 samples
for frame in range(nFrames):
    # "address" of the frame
    offset = frame * frameSize;
    # Loop over 18 non overlapping blocks of size 32
    for index in range(18):
        # process a block of 32 new input samples
        # see flow chart in Fig. 2
        ...
        ...


        # Undo the frequency inversion here
        ...

     end
end
```

The 512 coefficients of the filter c (see Fig. 3) are assigned using the values in the file `c.dat` in canvas.

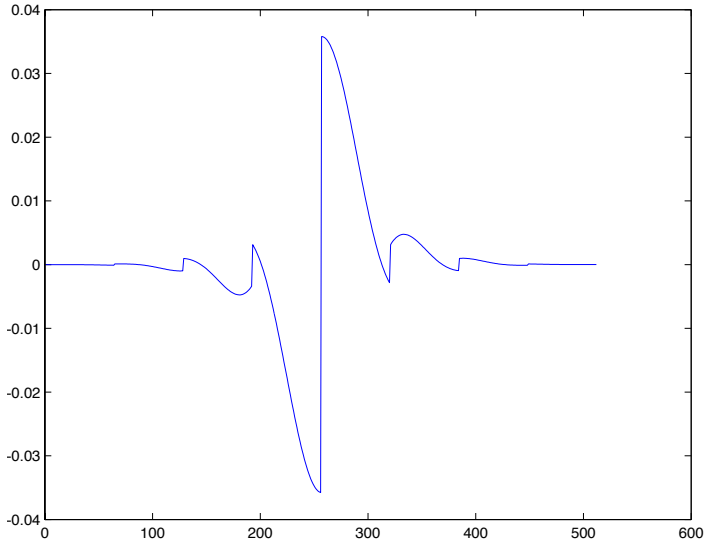

Figure 3: Analysis window c.

2.2 Frequency inversion

Downsampling the output of each bandpass filter has the effect of moving the spectrum of the filter's output down to baseband (i.e. to 0 frequency). Another way of saying this is that the bandpass filter's sub-sampled output is a set of samples representing the bandpass filter's output translated to baseband. However, because of the mathematics of the MP3 downsampling, every other subband has its spectrum "frequency inverted". Frequency inversion means that the baseband spectrum of the filter's output has its higher frequencies to the left of zero and its lower frequencies to the right of zero. The following diagram illustrates spectral inversion:

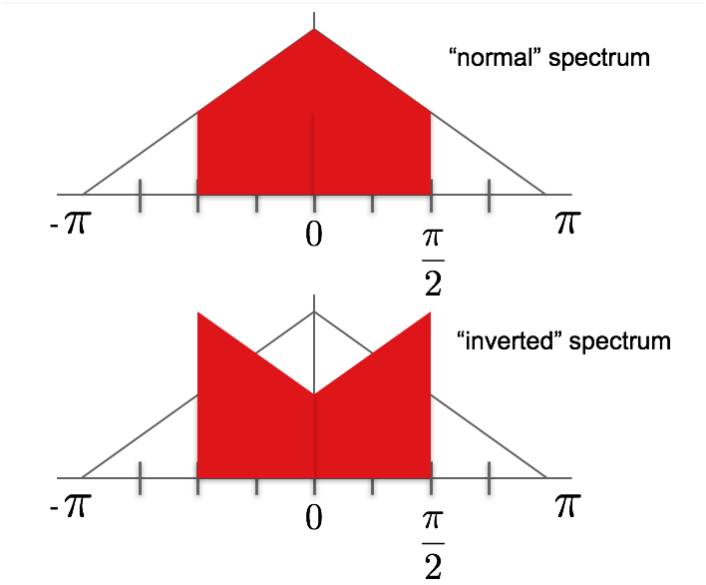

Figure 4: Spectrum inversion

If we begin subband numbering with 0, as in Python, then the *odd* subbands have their spectra inverted, while the even subbands do not. Because the MP3 algorithm further decomposes each filterbank's output into additional frequency bands using the MDCT, it is desirable to undo the frequency inversion. This can be done by multiplying the signal by $e^{i\pi n}$; the modulation property of the DTFT means that this multiplication has the effect of "sliding" the spectrum to the right by π, thereby restoring the natural spectral orientation (remember, the DTFT is periodic with period 2π).

In summary, to undo the frequency inversion, multiply every odd sample of every odd subband by -1.

Assignment

1. Write the Python function `pqmf` that implements the analysis filter bank described in equations (19-21). The function will have the following template:

   ```
   coefficients = pqmf(input)
   ```

   where `input` is a buffer (numpy array) that contains an integer number of frames of audio data. The output array `coefficients` has the same size as the buffer `input`, and contains the subband coefficients.

   The array coefficients should be organized in the following manner:

   $$\texttt{coefficients} = \left[S_0[0] \quad \ldots \quad S_0[N_S-1] \quad \ldots S_{31}[0] \ldots S_{31}[N_S-1]\right] \tag{22}$$

   where $S_i[k]$ is the coefficient from subband $i = 0, \ldots, 31$ computed for the packet k of 32 audio samples. Also N_S is the total number of packets of 32 samples:

   $$N_S = \texttt{nSamples} = 18 * \texttt{nFrames} \tag{23}$$

   The organization of `coefficients` is such that the low frequencies come first, and then the next higher frequencies, and so on. Figure 5 displays the array coefficients for the first 5 seconds of the piano sonata in `sample1.wav`. You notice the absence of very low frequencies, and few high frequencies. In comparison, figure 6 displays the array coefficients for the first 5 seconds of the dance floor piece in `sample2.wav`. You immediately notice the presence of very low frequencies (the thumping bass), and also significant high frequencies. These high frequencies may be interpreted as noise, and could be removed in an effort to compress the signal (more about that in the rest of the lab).

2. Analyze the first 5 seconds of the following tracks, and display the array `coefficients`, as in Fig. 5,
   - sample1.wav, sample2.wav
   - sine1.wav, sine2.wav
   - handel.wav
   - cast.wav
   - gilberto.wav

   Comment on the visual content of the arrays `coefficients`.

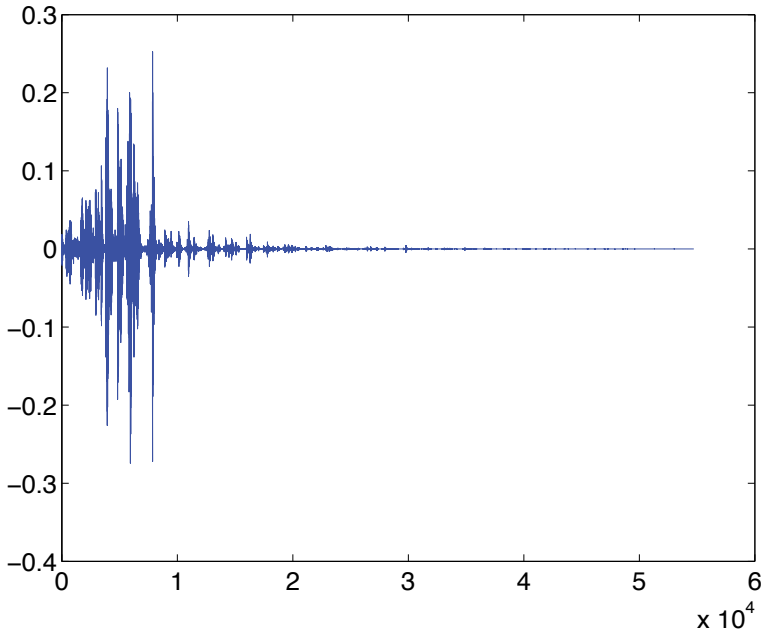

Figure 5: The array `coefficients` for the first 5 seconds of the audio sample 1.

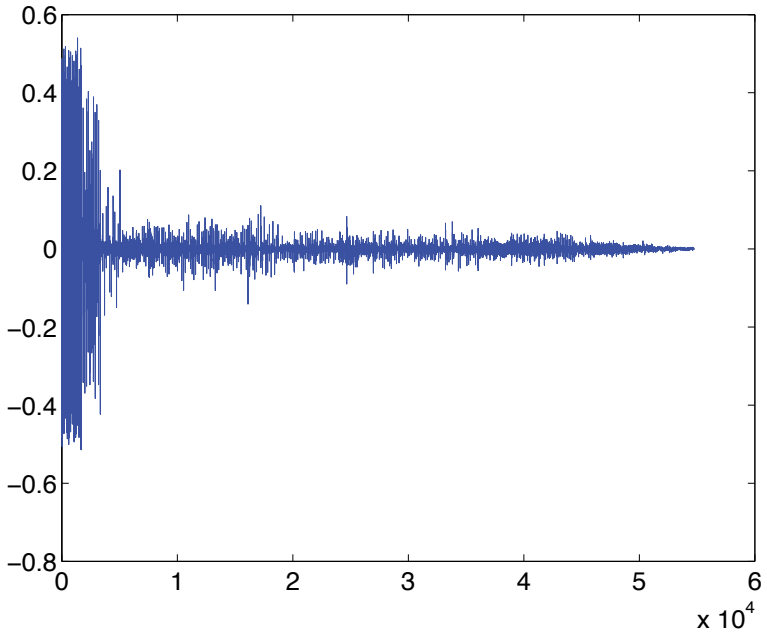

Figure 6: The array `coefficients` for the first 5 seconds of the audio sample 2.

2.3 Synthesis filter bank

The synthesis, or reconstruction, from the subband samples is performed in a very similar manner. The following (pseudo code) equations yield the reconstruction of 32 audio samples from 32 subband coefficients (NOTE: the reconstruction algorithm assumes frequency inversion is *present*, so the un-inverted spectral components are simply re-inverted first by applying the exact same step as above. It's clear this puts things back in their original state since $(-1)(-1) = 1$)

for $i = 1023$ down to 64 do

$$v[i] = v[i - 64] \tag{24}$$

for $i = 63$ down to 0 do

$$v[i] = \sum_{k=0}^{31} N_{i,k}\ s[k], \tag{25}$$

for $i = 0$ to 7 do

for $j = 0$ to 31 do

$$u[64\ i + j] = v[128\ i + j] \tag{26}$$

$$u[64\ i + j + 32] = v[128\ i + j + 96] \tag{27}$$

for $i = 0$ to 511 do

$$w[i] = d[i]\ u[i], \tag{28}$$

for $j = 0$ to 31 do

$$x[j] = \sum_{j=0}^{15} w[j + 32i]. \tag{29}$$

where

$$\boxed{N_{i,k} = \cos\left(\frac{(2k+1)(16+i)\pi}{64}\right), \quad i = 0, \ldots, 63,\ k = 0, \ldots, 31.} \tag{30}$$

Figure 8 shows the flow chart of the filtering of the audio data. The filtering is based on the following ideas.

1. The filtering is performed on a buffer V of size 1024, using equations (25)-(30).
2. Each processing cycle works on a packet of 32 audio samples.

3. During each processing cycle, the buffer V is shifted to the right, and $V(959:1023)$ are discarded (24). This leaves 64 empty slots that are filled with new data: the 64 numbers that are obtained from 32 coefficients using (28).

4. The audio samples are reconstructed using (29).

5. The cosine transform is different from the analysis filter, and is given by (30).

6. The prototype filter for the synthesis is $d[i]$. The 512 coefficients of the synthesis window d (see Fig. 7) are assigned using the file `d.dat` in canvas.

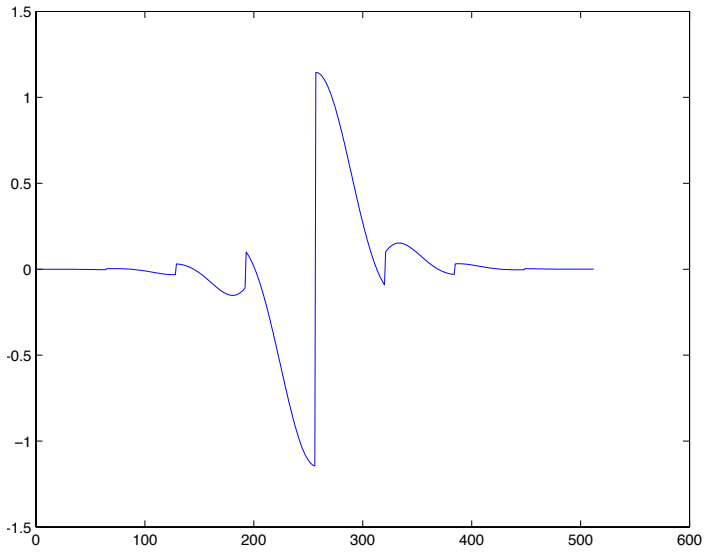

Figure 7: Analysis window d.

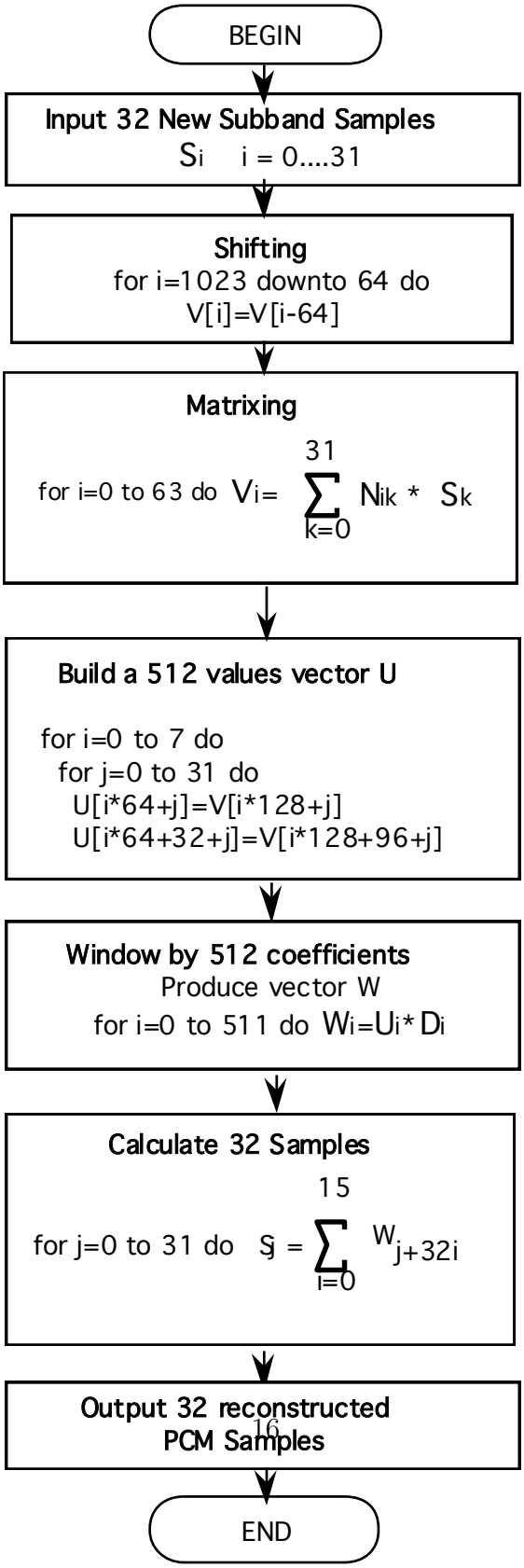

Figure 8: Decoding flow chart: the buffer V receives 32 new PQMF coefficients, $S_0, \ldots, S_{31}$, and 32 new audio samples $X_0, \ldots, X_{31}$ are reconstructed.

Assignment

3. Write the Python function `ipqmf` that implements the synthesis filter bank described in equations (25-(30). The function will have the following template:

   ```
   recons = ipqmf(coefficients)
   ```

   where `coefficients` is a buffer that contains the coefficients computed by `pqmf`. The output array `recons` has the same size as the buffer `coefficients`, and contains the reconstructed audio data. Don't forget that the first step in this function must be re-inverting the required spectral components.
4. Reconstruct the first 5 seconds of the following tracks, and display the signal `recons` on top of the signal `input` as in Fig. 9. You should observe that the reconstructed signal is slightly delayed. This is due to the fact that the processing assumes that a buffer of 512 audio samples is immediately available.
   - sample1.wav, sample2.wav
   - sine1.wav, sine2.wav
   - handel.wav
   - cast.wav
   - gilberto.wav
5. Compute the maximum error between the reconstructed signal and the original, taking into account the delay. The error should be no more than 10^{-5}. Explain how you estimate the delay.

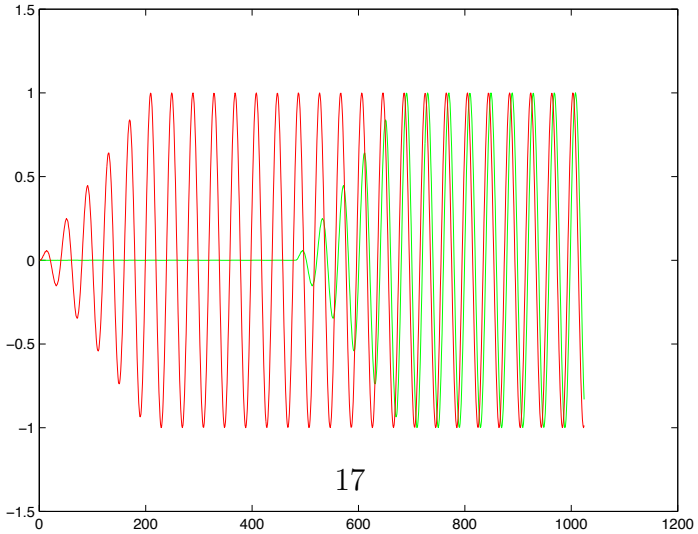

Figure 9: First 1024 samples from the reconstructed signal (green) and the original signal (red), for `sine1.wav`.

Assignment

6. Modify your code to reconstruct an audio signal using only a subset of bands. This is the beginning of compression. Your function prototype should look like this:

   ```
   recons = ipqmf (coefficients, thebands)
   ```

   where `thebands` is an array of 32 integers, such that `thebands[i]` =1 if band `i` is used in the reconstruction, and `thebands[i]` = 0 if band `i` is not used.
   Experiment with the files
   - sample1.wav, sample2.wav
   - sine1.wav, sine2.wav
   - handel.wav
   - cast.wav
   - gilberto.wav

   and describe the outcome of the experiments, when certain bands are not used to reconstruct. If you define the compression ratio as

   $$\frac{\text{number of bands used to reconstruct}}{32}, \tag{31}$$

   explain what is a good compression ratio, and a good choice of bands for each audio sample. Note that the psychoacoustic model of MP3 performs this task automatically.