Lab 4: MPEG Audio Signal Processing

Shane Kirkley

March 18, 2019

Table of Contents

Introduction

Audio compression is an important part of the efficient storage and sharing of digital audio. In this lab, we will implement MPEG 1 Audio Layer III (MP3) encoding and decoding. The MP3 codec breaks down a digital audio signal into frequency subbands. From the coefficients of these subbands, the original audio signal can be synthesized. First we will implement the MPEG analysis and synthesis functions, and examine the results on several audio samples. Finally, we will analyze compression of digital audio using MPEG, and optimize this compression by examining the balance of information loss and file size.

Background

The MP3 codec uses several digital signal processing techniques. Filter banks are used in both the analysis and synthesis procedures. Recall that a filter bank is an array of band-pass filters, which effectively subdivides a signal into multiple components, each containing a frequency sub-band. Up- and down-sampling are also employed in the MP3 codec. Downsampling, or decimation, by a factor of N effectively keeps every N^{th} sample from a signal. Conversely, upsampling in the codec is implemented by adding zeros between samples, producing an approximation of the signal had it been sampled at the upsample rate.

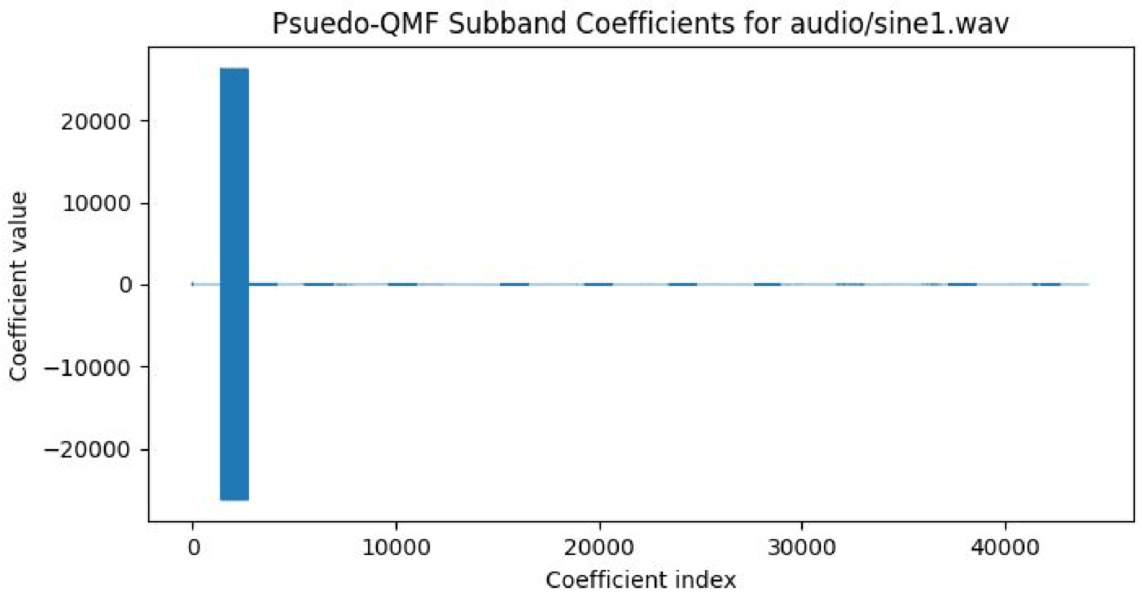
Part I: Encoding

a. Analysis Filter Bank

Encoding the audio signal first involves splitting the signal into 32 equally spaced frequency bands using an analysis filter bank. The analysis filter bank is made up of pseudo quadrature mirror filters (pseudo-QMF). The output of each filter is critically sampled by downsampling by a factor of 32. Because each filter output is critically sampled, each block of 32 samples from the input signal produces 32 output coefficients. If some of these subband coefficients were omitted, the result would be lossy compression - a smaller file size with a trade off of losing information for the omitted frequency subbands. Function `pqmf`, located in script `lab04_funcs.py` in the appendix, implements the subband coefficient calculation.

b. Subband Coefficient Results

The script `test_pqmf.py`, visible in the appendix, was run on several audio samples, producing plots of subband coefficients as shown in the following pages. Coefficients were calculated over the first 5 seconds of audio for each sample, then plotted in order of low frequency to high frequency subbands. These plots reveal some interesting insights into the frequency composition of each audio sample. Figure 1 shows the subband coefficients for pure sine waves. Because these audio samples have a single fixed frequency, only a single subband is needed to contain the information necessary to reconstruct the time-domain signal. We see in the plots that `sine1.wav` is only non-zero in the second subband, and `sine2.wav` is only non-zero in the 1st subband. Additionally, these subband coefficients are constant, indicating that the frequency occurs uniformly throughout the samples.



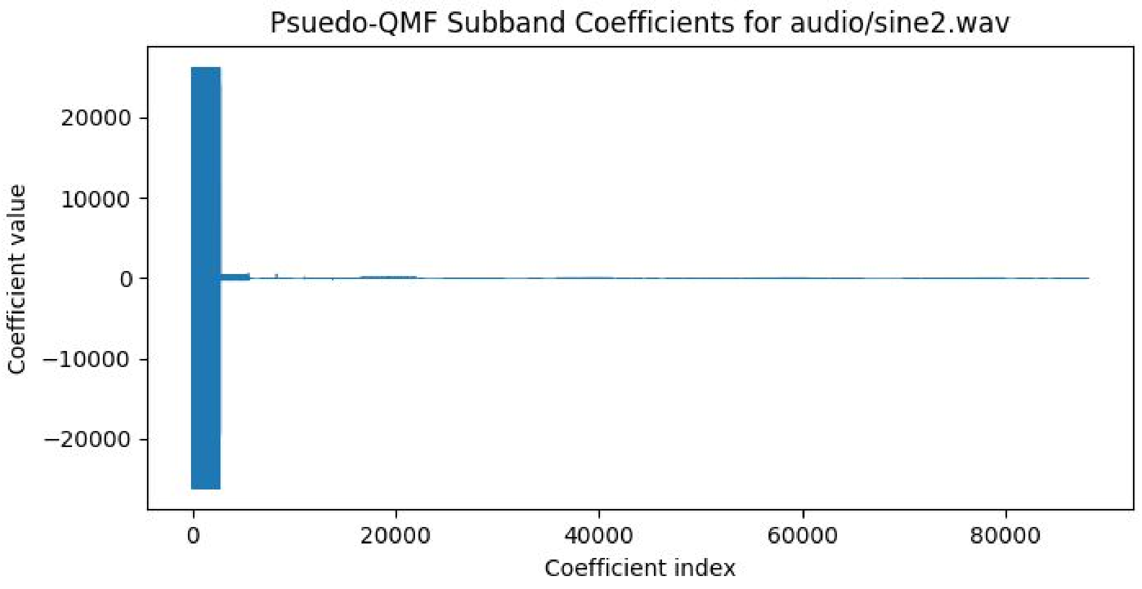


Figure 1 - Subband coefficients for sine1.wav and sine2.wav.

Figure 2 shows the coefficients for a classical piano piece. We note the magnitude of coefficients is higher at low frequency subbands, with some noise at higher subbands. Figure 3 shows the plot of subband coefficients for audio sample `cast.wav`, which contains percussion sounds. We see a wider spread of high magnitude coefficients throughout most of the subbands, indicating high frequency sounds occur in the sample.

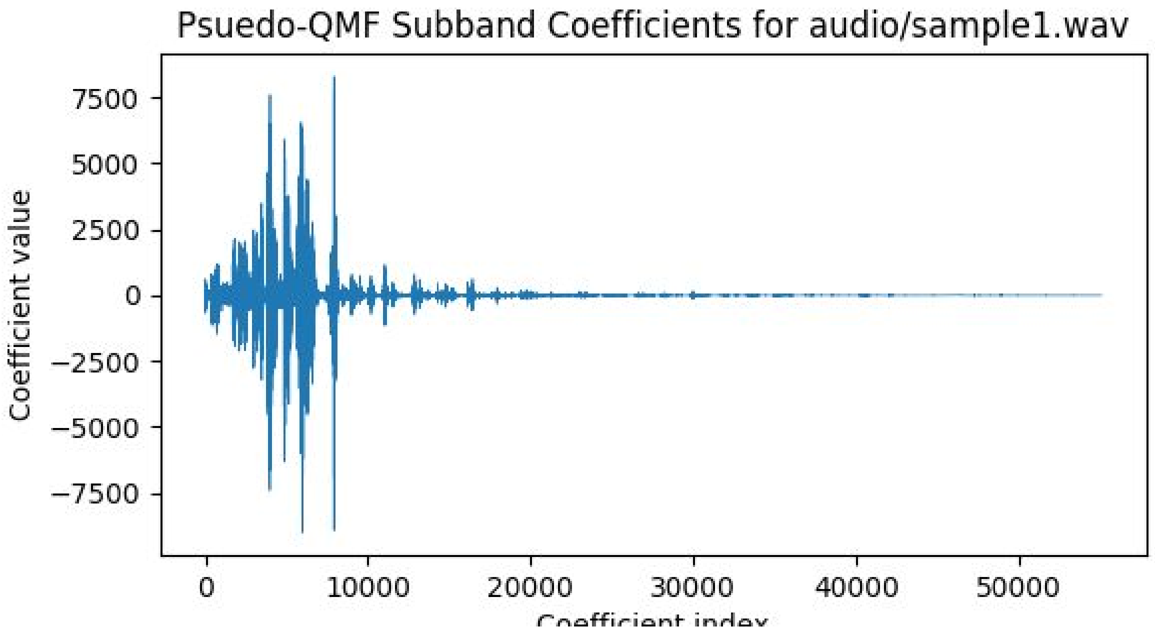


Figure 2 - Subband coefficients for piano piece, sample1.wav.

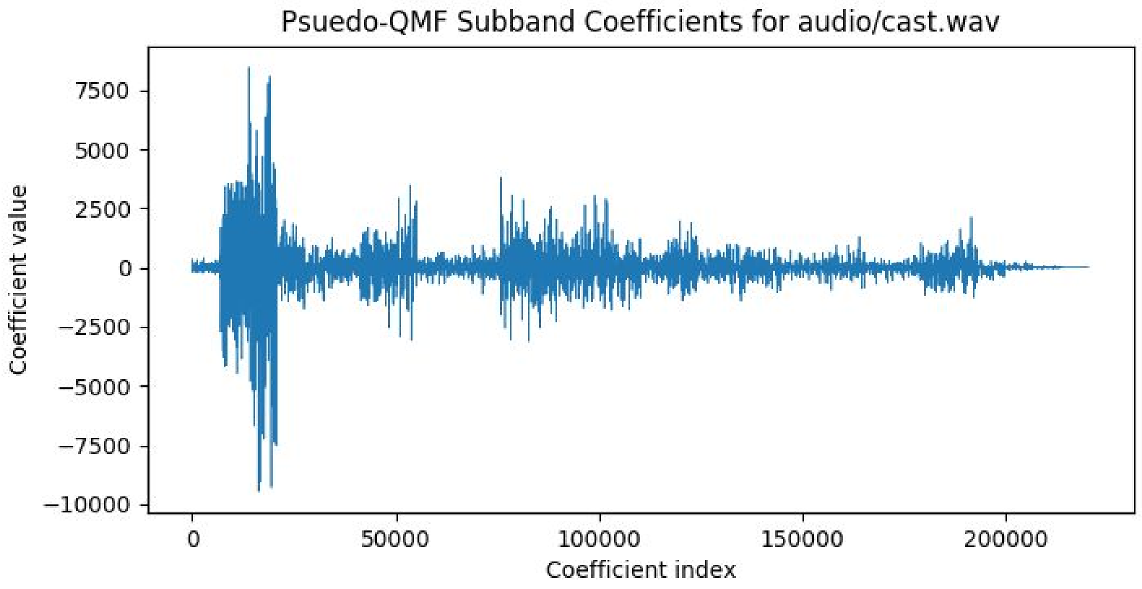


Figure 3 - Subband coefficients for percussion piece, cast.wav.

The first 5 seconds of `gilberto.wav` contains only percussion. In Figure 4, we see the gradual tapering of subband coefficients towards zero with higher frequency subbands. Figure 5 shows the subband coefficients for `handel.wav`, a sample containing classical violin with vocals. Here we notice sharp peaks in each subband, indicating that those subband frequencies don't occur uniformly throughout the sample.

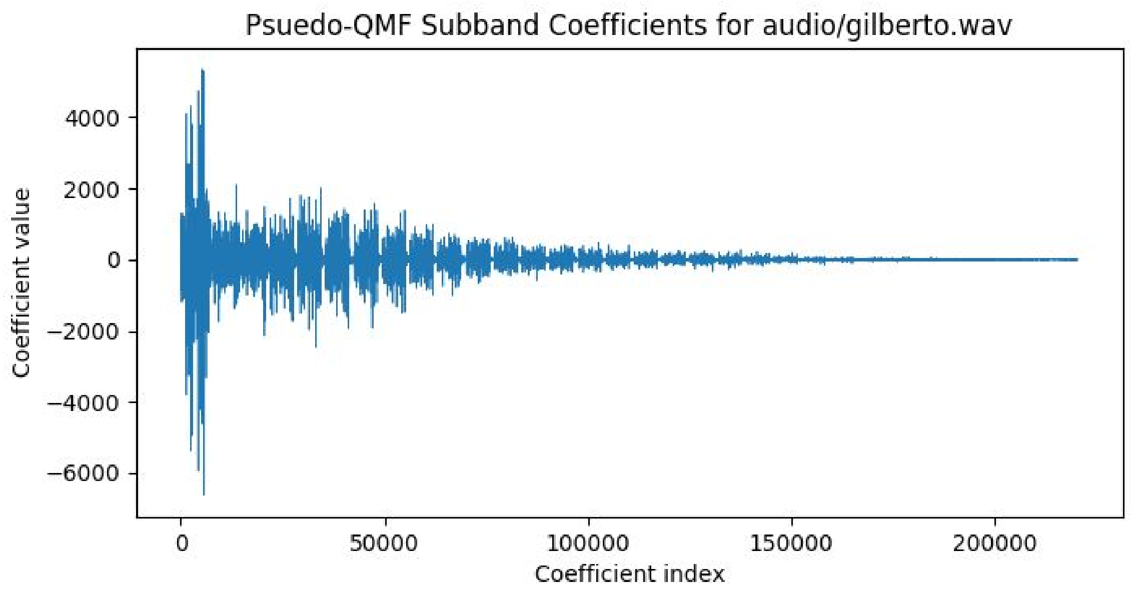


Figure 4 - Subband coefficients for percussion in gilberto.wav.

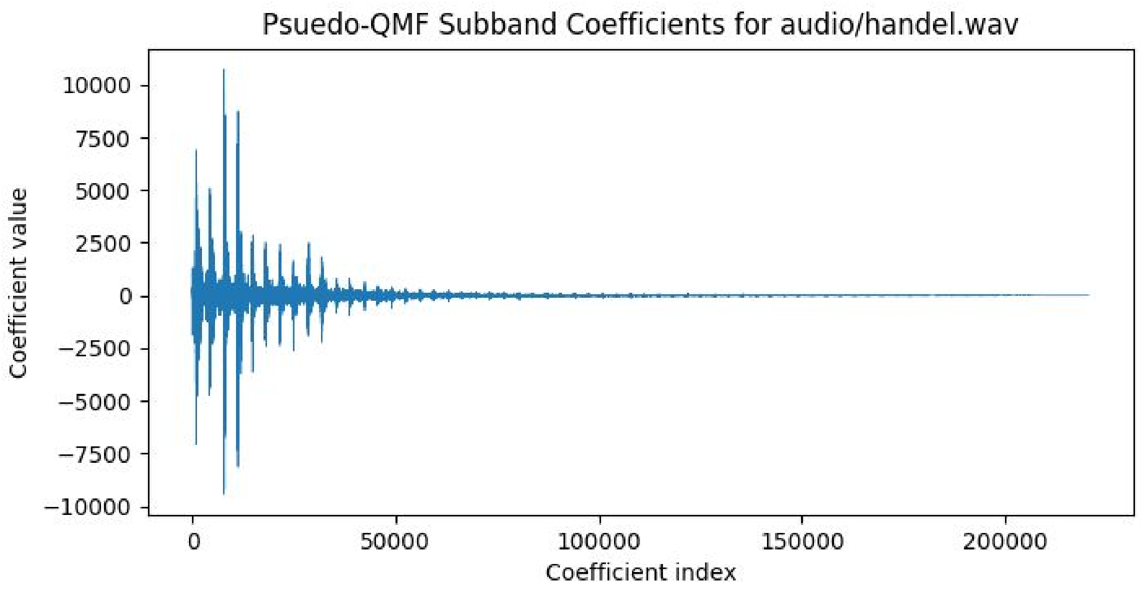


Figure 5 - Subband coefficients for classical sample, handel.wav.

Finally, Figure 6 shows subband coefficients for an electronic piece, `sample2.wav`. We notice the high magnitude of low frequency subband coefficients from the bass beat, plus higher frequency subband coefficients from the synthesizer melody, as well as noise.

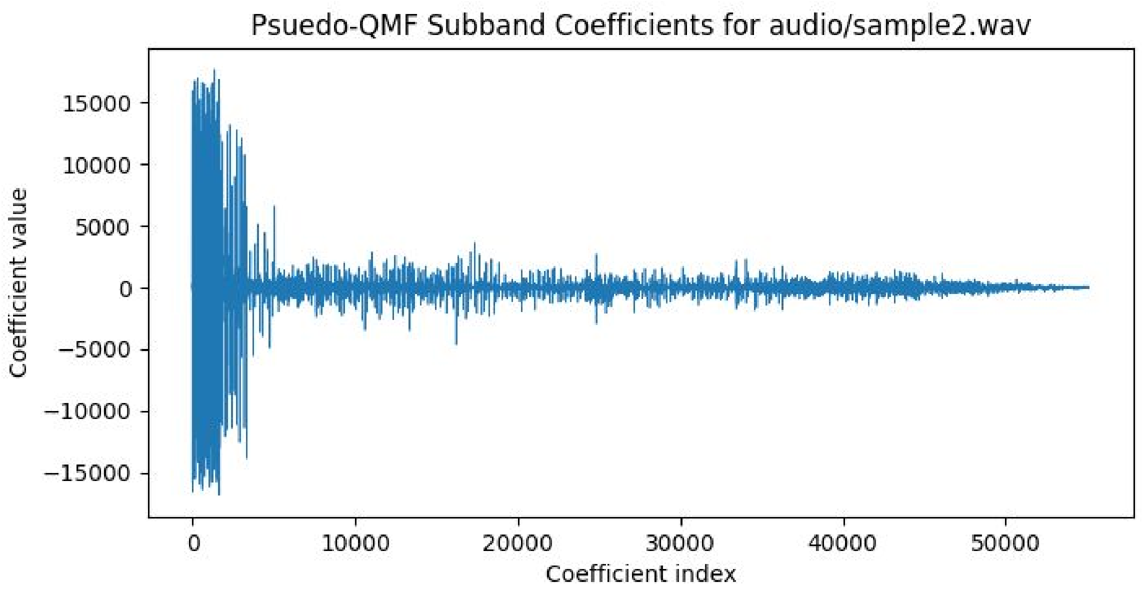


Figure 6 - Subband coefficients for electronic music, sample2.wav.

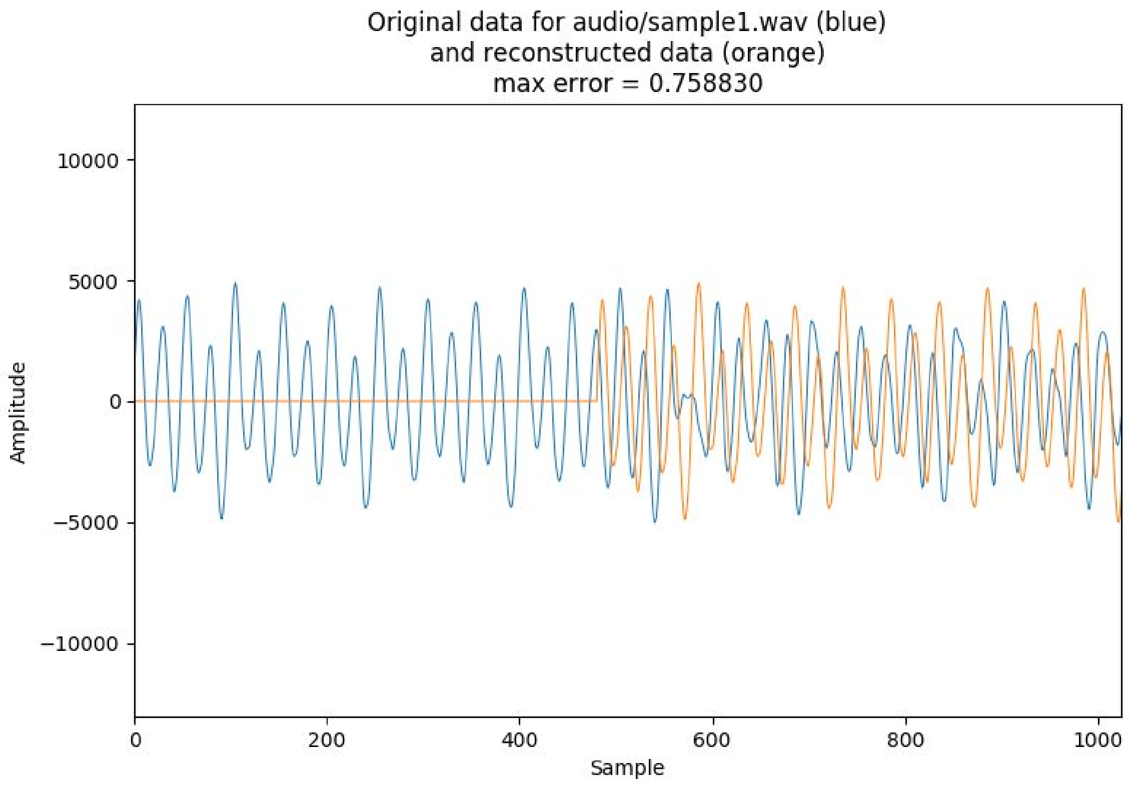
Part II: Decoding

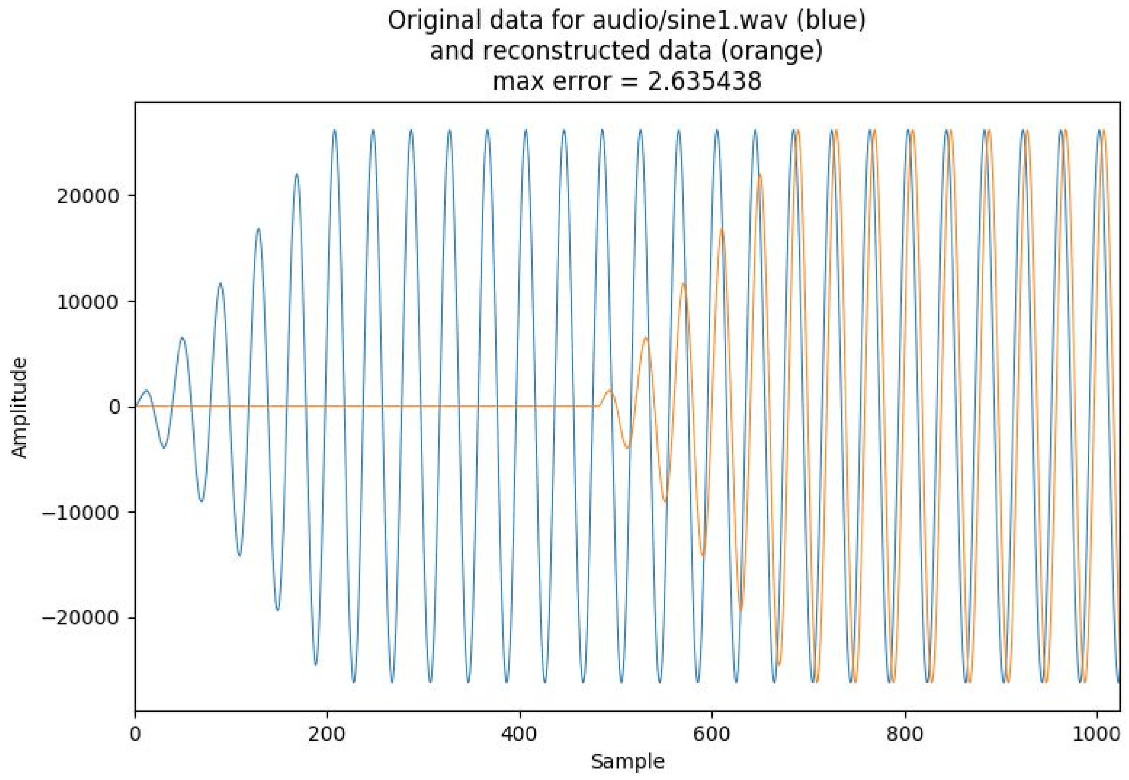
a. Synthesis Filter Bank

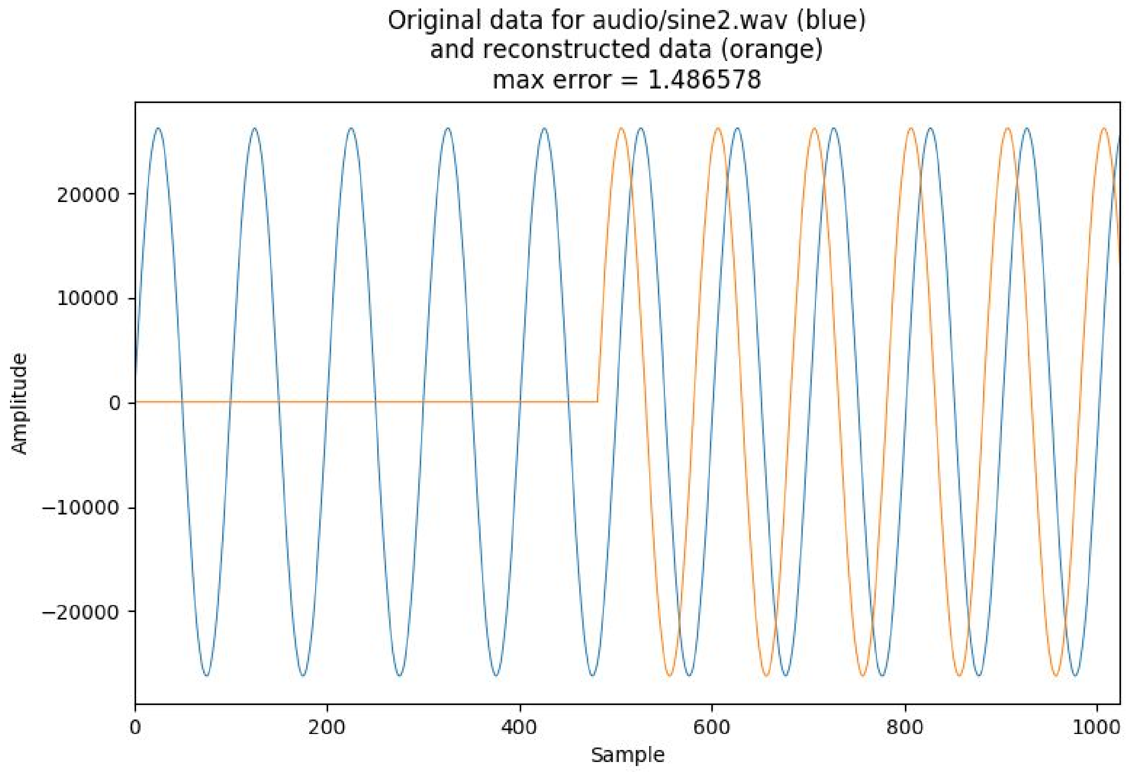
The audio signal can be reconstructed from the subband coefficients in a similar manner to the encoding step, essentially undoing the steps performed in encoding. Subband coefficients are upsampled, then passed through a synthesis filter bank. This filter bank uses a cosine transform which differs from the analysis filter bank. The output of each filter is then summed to create the output signal. Function ipqmf in script lab04_funcs.py, located in the appendix, implements MP3 synthesis. A small error will be present in the reconstructed audio signal - this error will grow depending on which subbands were omitted in the compression algorithm. This will be examined further in part c.

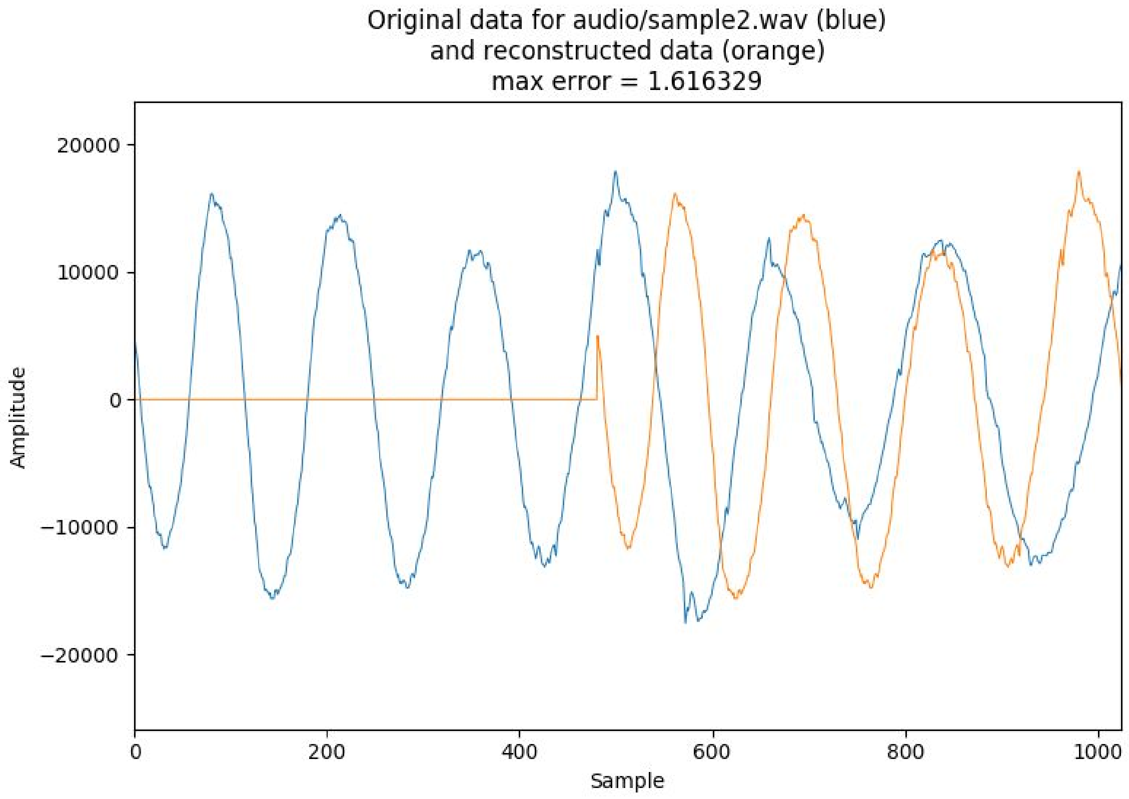
b. Reconstruction

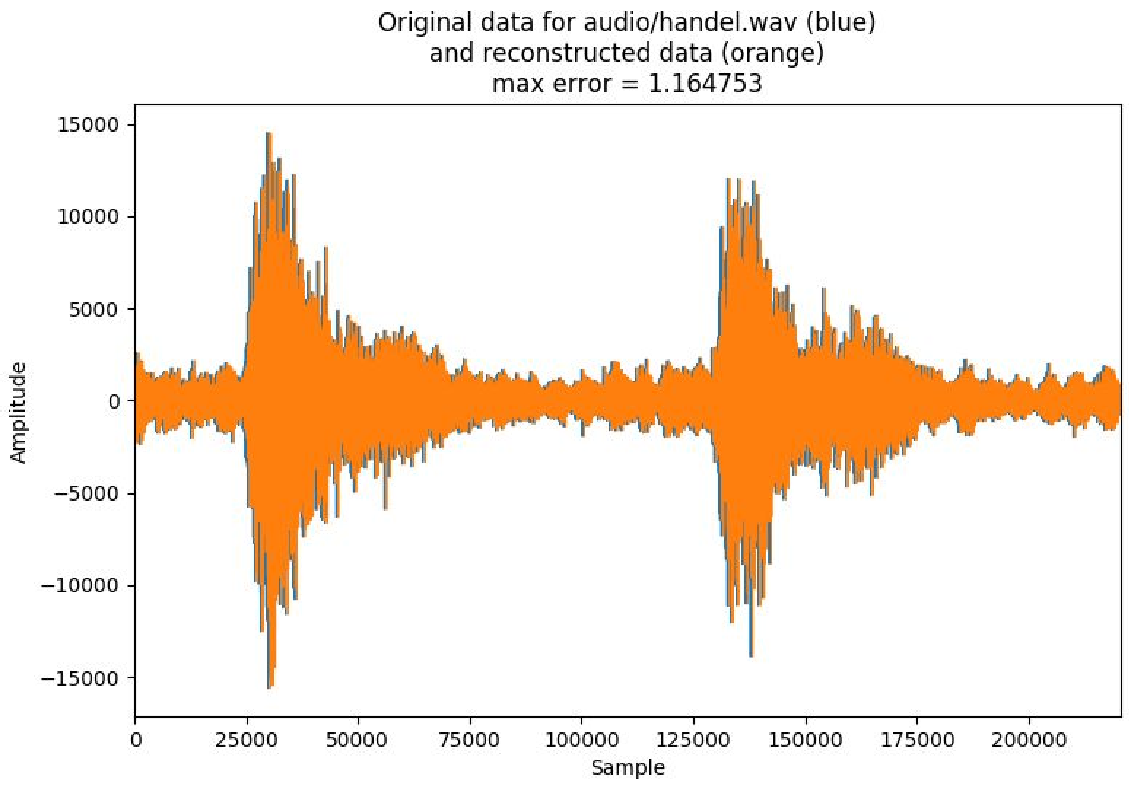
Script test_ipqmf.py, visible in the appendix, was run on several audio samples to compare original and reconstructed audio data with no compression. The following pages show the resulting plots of synthesized and original audio data. We notice that in all the plots, the synthesized signal is shifted to the right. Intuitively, we estimate that this time shift is around 512 samples, because this is the number of samples required to fill the input sample buffer. By experimenting with different shift values, we determine the exact sample shift to be 481. The first four plots show 1024 samples of audio data, and the last three plots show 5 seconds of audio data. Each plot indicates the absolute maximum error between the original and synthesized signals. We find the average max error for these samples to be 1.245, or about 4.5 x 10^{-5} percent.

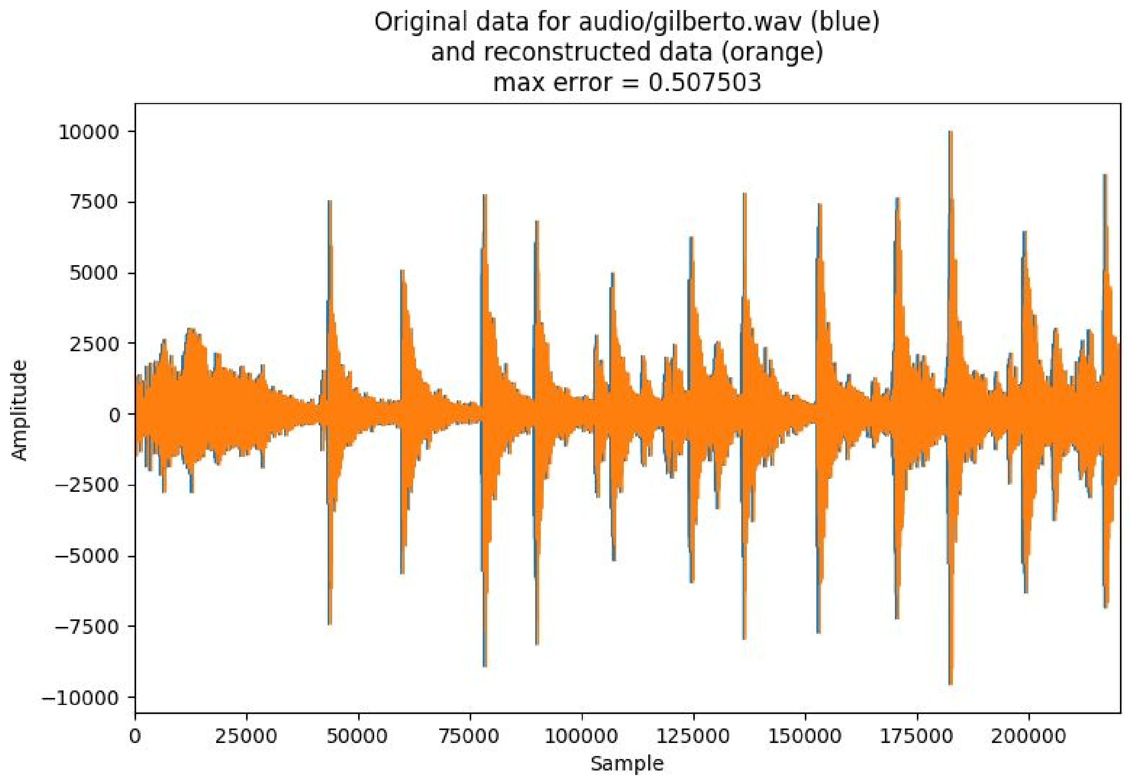

Original data for audio/sample1.wav (blue)
and reconstructed data (orange)
max error = 0.758830
10000
5000
0
−5000
−10000
Amplitude
0
200
400
600
800
1000
Sample

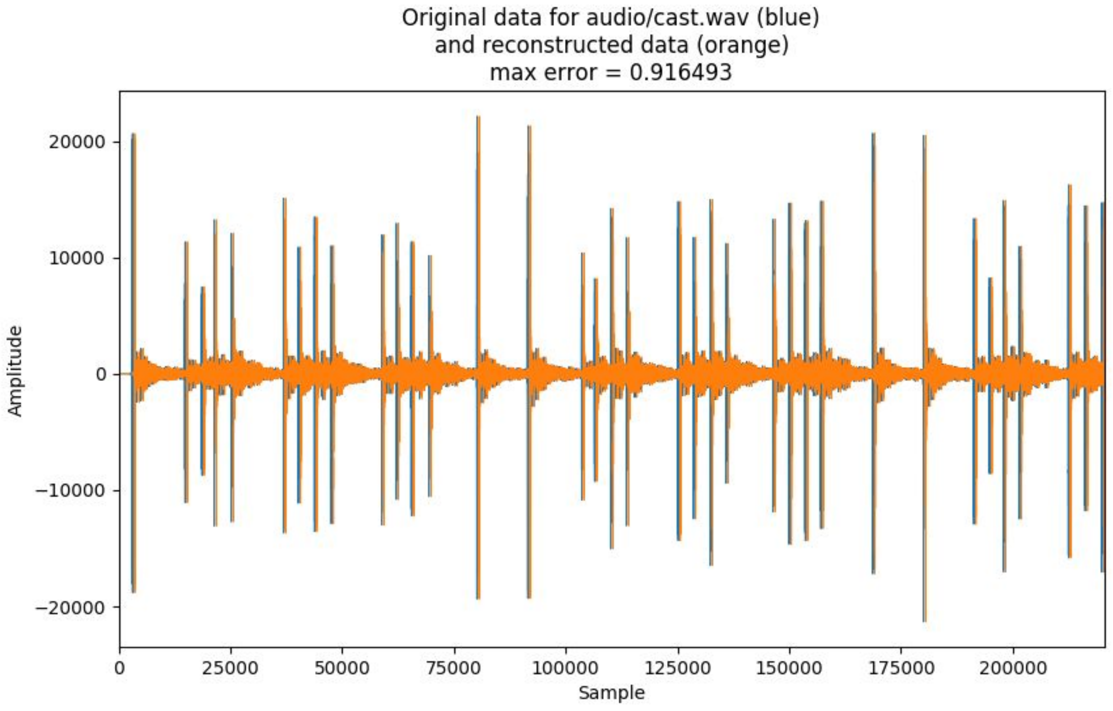

Original data for audio/sine1.wav (blue)
and reconstructed data (orange)
max error = 2.635438
20000
10000
0
−10000
−20000
Amplitude
0
200
400
600
800
1000
Sample


Original data for audio/sine2.wav (blue)
and reconstructed data (orange)
max error = 1.486578
Amplitude
Sample
20000
10000
0
−10000
−20000
0
200
400
600
800
1000


Original data for audio/sample2.wav (blue)
and reconstructed data (orange)
max error = 1.616329
Amplitude
Sample
20000
10000
0
−10000
−20000
0
200
400
600
800
1000


Original data for audio/handel.wav (blue)
and reconstructed data (orange)
max error = 1.164753
15000
10000
5000
0
−5000
−10000
−15000
Amplitude
0
25000
50000
75000
100000
125000
150000
175000
200000
Sample


Original data for audio/gilberto.wav (blue)
and reconstructed data (orange)
max error = 0.507503
10000
7500
5000
2500
0
−2500
−5000
−7500
−10000
Amplitude
0
25000
50000
75000
100000
125000
150000
175000
200000
Sample



c. Compression

We can explore the effects of compression by omitting frequency subbands in the synthesis function. Function `ipqmf_bands` in script `lab04_funcs.py` takes takes a bitmap array which allows us to specify which subbands to omit. Figure 7 shows the absolute maximum error resulting from removing a number of subbands, starting from highest frequency subbands, for audio `sample1.wav`. (i.e. five indicates bands 28-32 have been omitted, six indicates bands 27-32 omitted, etc.)

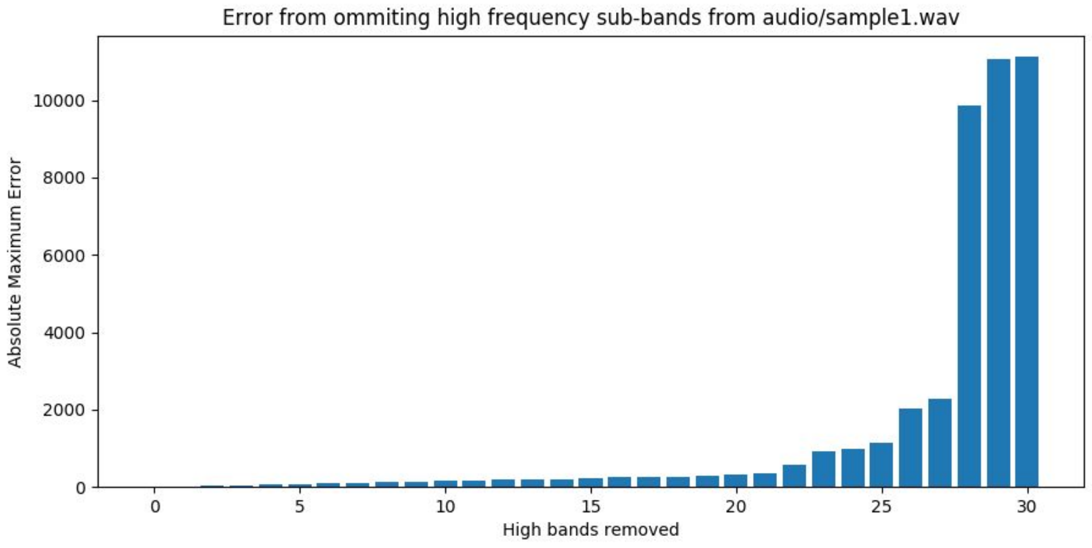
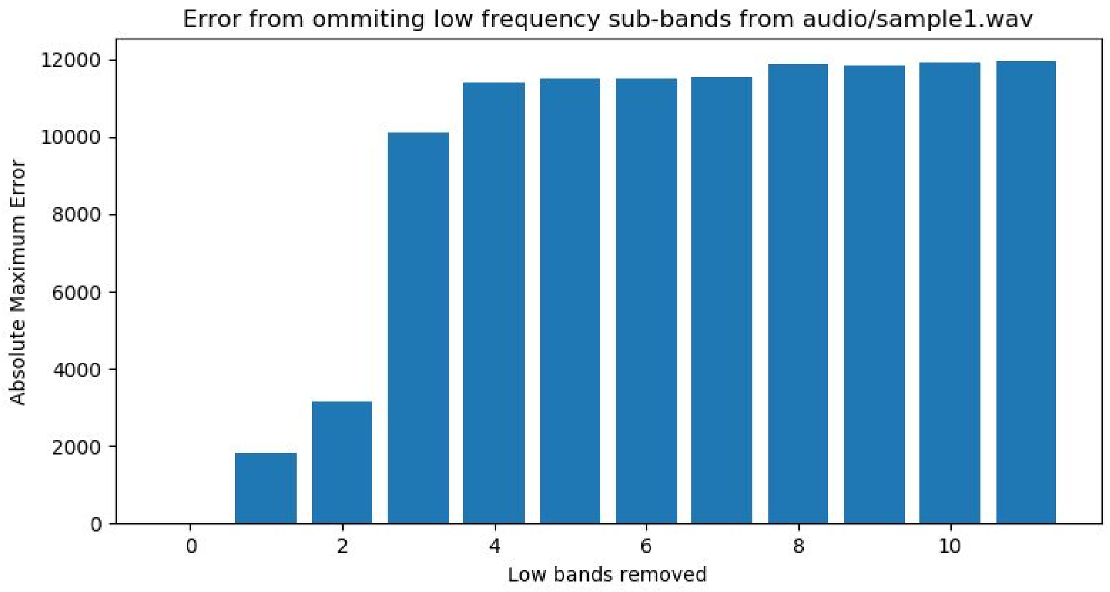


Figure 7 - Absolute max error in synthesizing when removing numbers of high frequency subbands.

This plot indicates that the majority of information for this audio signal is contained in the lowest subbands. Figure 8 shows the error resulting from the omission of the lowest subbands in the synthesis of `sample1.wav`. Again we see that the removal of the first three subbands results in approximately 80% error from the original signal. These plots were generated using script `test_compression.py`, visible in the appendix.



From these results, we conclude that the balance of quality and file size can be optimized by removing a number of the highest frequency subbands. For `sample1.wav`, the removal of the 21 highest subbands (a compression ratio of 34%) results in a synthesis with only 3% error from the original signal. The optimal compression ratio depends on the signal being compressed, however for most music this choice of subband omissions and compression ratio would be satisfactory, given that the compressed file size would be 34% of the original.

Conclusion

By implementing a simple MP3 codec, several insights into digital signal processing and compression performance have been gained. Analysis of a digital audio signal into frequency subbands allows for the efficient compression of audio data. Synthesis of these subbands results in a restored signal with minimal error depending on the subbands chosen for omission and compression ratio.

Appendix

lab04_funcs.py

```
#!/usr/local/bin/python3
import numpy as np
import math

C_TAP_FILENAME = "the_c_taps.txt"
D_TAP_FILENAME = "the_d_taps.txt"

def load_c_taps( filename_path ):
 fd = open(filename_path, 'r')
 c = np.zeros(512, np.float)
 for ctr, value in enumerate(fd):
   c[ctr] = np.float( value )
 fd.close()
 return c

def load_d_taps( filename_path ):
 fd = open(filename_path, 'r')
 d = np.zeros(512, np.float)
 for ctr, value in enumerate(fd):
   d[ctr] = np.float( value )
 fd.close()
 return d

def pqmf(input):
 """
 Lab assignment 1.
 Input is a buffer of audio data with integer multiple of 32 (trimmed if not).
 Output coefficients has the same size as input buffer and contains the subband
coefficents.
 """
 X = np.zeros(512)
 C = load_c_taps(C_TAP_FILENAME)
 M = np.zeros((64,32))
 for k in range(32):
   for r in range(64):
     M[r,k] = np.cos((2*k+1)*(r-16)*np.pi/64)

 # sort input into 32 columns
 rows = int(np.floor(len(input)/32))
 input = input[:rows*32] # trim extra samples
 input = np.reshape(input, (rows,32))

 # output same size as input
 output = np.zeros(input.shape)
  modulate = np.resize([1,-1], 32)

 for row in range(rows):
   X = np.roll(X, 32)
   X[:32] = np.flip(input[row])
   Z = C * X

   # reshape Z to make the partial calculation easier
   Z = np.reshape(Z, (8, 64))
   v = np.ones(8, np.float)
   Y = v.dot(Z)

   # TODO: matrix multiply to get S = M * Y
   S = np.zeros(32)
   for i in range(32):
     S[i] = np.dot(Y, M[:,i])

   # invert odd coefficients of odd subbands
   if row % 2:
     output[row] = S * modulate
   else:
     output[row] = S

 return output

def ipqmf(coefficents):
 """
 Lab assignment 3.
 input is a buffer of coefficients computed by pqmf.
 Output array has same size as coefficients, and contains the reconstructed audio
data.
 """
 V = np.zeros(1024)
 D = load_d_taps(D_TAP_FILENAME)
 output = np.zeros(coefficents.shape)
 rows = coefficents.shape[0]
 N = np.zeros((64,32))
 for i in range(64):
   for k in range(32):
     N[i,k] = np.cos((2*k+1)*(16+i)*np.pi/64)
 modulate = np.resize([1,-1], 32)

 for row in range(rows):
   S = coefficents[row] # input 32 new subband samples
   if row % 2:          # re-invert odd coefficients of odd subbands
     S = S * modulate
   V = np.roll(V, 64)   # shifting
   for i in range(64):  # matrixing
     V[i] = N[i,:].dot(S)
   U = np.zeros(512)    # build a 512 value vector U
   for i in range(8):
     for j in range(32):
       U[i*64+j] = V[i*128+j]
       U[i*64+32+j] = V[i*128+96+j]
   W = U * D            # window by 512 coefficients
   W = np.reshape(W, (16, 32))
   v = np.ones(16, np.float)
   output[row] = v.dot(W)
 return output

def ipqmf_bands(coefficents, bands):
 """
 Lab assignment 6.
 input:
   coefficients - a buffer of coefficients computed by pqmf.
   bands - bitmap array where...
     bands[i] = 1 if band i is used in reconstruction.
     bands[i] = 0 if band i is not used in reconstruction.
 output: array of reconstructed audio data
 """
 V = np.zeros(1024)
 D = load_d_taps(D_TAP_FILENAME)
 output = np.zeros(coefficents.shape)
 rows = coefficents.shape[0]
 N = np.zeros((64,32))
 for i in range(64):
   for k in range(32):
     N[i,k] = np.cos((2*k+1)*(16+i)*np.pi/64)
 modulate = np.resize([1,-1], 32)

 for row in range(rows):
   # input 32 new subband samples as specified
   S = coefficents[row] * bands
   if row % 2:          # re-invert odd coefficients of odd subbands
     S = S * modulate
   V = np.roll(V, 64)   # shifting
   for i in range(64):  # matrixing
     V[i] = N[i,:].dot(S)
   U = np.zeros(512)    # build a 512 value vector U
   for i in range(8):
     for j in range(32):
       U[i*64+j] = V[i*128+j]
       U[i*64+32+j] = V[i*128+96+j]
   W = U * D            # window by 512 coefficients
   W = np.reshape(W, (16, 32))
   v = np.ones(16, np.float)
   output[row] = v.dot(W)
 return output
```

test_pqmf.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
import sys
import lab04_funcs as lab4

# generate plots of coefficients for 5 seconds of all audio data

fs, data = wavfile.read(sys.argv[1])
data = data[:fs*5] # trim to 5 seconds
coeffs = lab4.pqmf(data)

plt.plot(coeffs.T.flatten(), linewidth=0.5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient value")
plt.title("Psuedo-QMF Subband Coefficients for %s" % sys.argv[1])
plt.show()
```

test_ipqmf.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
import sys
import lab04_funcs as lab4

fs, data = wavfile.read(sys.argv[1])
data = data[:fs*5] # trim to 5 seconds

coeffs = lab4.pqmf(data)

# plt.plot(coeffs.T.flatten(), linewidth=0.5)
# plt.show()

synth = lab4.ipqmf(coeffs).flatten()

plt.plot(data, linewidth=0.7)
plt.plot(synth, linewidth=0.7)

data = data[:synth.shape[0]] # trim data to match synth
diff = abs(data[:-481] - synth[481:]) # delay found experimentally (expected 512)
err = np.max(diff)

plt.title("Original data for %s (blue)\nand reconstructed data (orange)\nmax
error = %f" % (sys.argv[1], err))
plt.xlim(0, fs*5)
plt.xlabel("Sample")
plt.ylabel("Amplitude")
plt.show()
```

test_compression.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
import sys
import lab04_funcs as lab4

TEST_LOW = True
TEST_HIGH = True

# returns max error between original data and synthesized data
def synth_error(data, synth):
   data = data[:synth.shape[0]]          # trim data to match synth
   diff = abs(data[:-481] - synth[481:]) # delay found experimentally (expected
512)
   return np.max(diff)


fs, data = wavfile.read(sys.argv[1])
data = data[:fs*5] # trim to 5 seconds
coeffs = lab4.pqmf(data)

# removed low bands:
if (TEST_LOW):
   err = np.zeros(12)
   for i in range(12):
       thebands = np.ones(32)
       thebands[:i] = 0
       synth = lab4.ipqmf_bands(coeffs, thebands).flatten()
       err[i] = synth_error(data, synth)

   plt.bar(range(12),err)
   plt.title("Error from ommiting low frequency sub-bands from %s" % sys.argv[1])
   plt.ylabel("Absolute Maximum Error")
   plt.xlabel("Low bands removed")
   plt.show()

# removed high bands:
if (TEST_HIGH):
   num_bands = 31 # number of bands to test removal of
   err = np.zeros(num_bands)
   for i in range(1, num_bands):
       thebands = np.zeros(32)
       thebands[:-i] = 1
```

```
        synth = lab4.ipqmf_bands(coeffs, thebands).flatten()
        err[i] = synth_error(data, synth)

   plt.bar(range(num_bands),err)
   plt.title("Error from ommiting high frequency sub-bands from %s" %
sys.argv[1])
   plt.ylabel("Absolute Maximum Error")
   plt.xlabel("High bands removed")
   plt.show()
```