Audiofile Coder

@NMme

1 Introduction

In this assignment a lossy coder for stereo music had to be constructed. The given music files which had to be coded were two songs sampled at $44.1kHz$ and with a raw data rate of $1411.2kbits/s$. The coded files should have a data rate of $192kbits/s$ and a signal-to-noise-ratio of at least $30dB$. Python was used to implement these concepts.

2 Overview of the Coder

A transform coding approach was used to code the music files.

2.1 Preprocessing the source-signal

After reading the data from the musicfile an array of 2d-vectors containing the samples for the left and right channels is received. In order to code both channels at the same time interleaving was chosen and implemented by flattening the array. To apply any sort of transform coding the data has to be split into equally sized blocks. As Interleaving was chosen and one constraints was to use a maximum blocks size of 4096 per channel this approach allows a maximum block size of 8192.

```
# read wavefile
fs, data = wavfile.read(path)
data = np.ravel(data)

# divide data in blocks
l_over = len(data)%bl_size
rest = data[len(data)-l_over :]
data = data[: len(data)-l_over]
blocks = np.array([data[i:i+bl_size] for i in range(0, len(data), bl_size)], dtype=np.int32)
```

2.2 Transformation

The resulting blocks are transformed by using the *Discrete-Cosine-Transform* (DCT). This transformation is very similar to the Fourier-Transform and can be implemented using the FFT algorithm which results in a good performance. In Python the scipy module offers a easy to use implementation of the (DCT). After the transform is applied to the blocks, a block of the same size is returned containing the coefficients describing the data sequence in a new basis. As interleaving was used it can often be observed that the coefficients are large for the low and very high frequencies.
In the next step theses coefficients have to be quantized.

```
# transform blocks with DTC and Quantize them
for i in range(0,len(blocks)):
        blocks[i] = fftpack.dct(blocks[i], norm='ortho')
```

2.3 Quantization

Through quantization information is lost but there can be a huge gain in compression. To find the right quantization parameters the statistics of the transform coefficients are utilized. First the variance of each coefficient during the whole song was determined. After that the following equation was used to determine how many bits should be used to quantize each coefficient:

$$R_k = R + 0.5 \cdot \log_2 \frac{\sigma_k^2}{(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}} \tag{1}$$

This equation is only dependent on the parameter R which describes the average number of bits per coefficient. With these parameters a quantization table for uniform quantization can be found. This table needs to be transmitted only once and can be used to reconstruct every block. The quantization is simply done by performing an integer-division on each coefficient with the according value from the table.

```
# group all coeffiecents
coeff = np.zeros((bl_size, len(blocks)))
for i in range(0,len(blocks)):
        for j in range(0, bl_size):
                coeff[j][i] = blocks[i][j]

# calculate variance for each coefficient
var_coeff = [np.var(c) for c in coeff]

# calculate quantization levels per coefficient
nenner = np.prod( np.power(var_coeff, 1.0/bl_size))
r_coeff = [round(avg_q + 0.5*math.log(v/nenner ,2)) for v in var_coeff]

# find uniform quantization intervals
quan_table = [round( (np.amax(coeff[i])- np.amin(coeff[i]))/(2.0**r_coeff[i]) ) for i in
    range(0,bl_size)]

# quantize according to the table
for i in range(0,len(blocks)):
        for j in range(0, bl_size):
                blocks[i][j] = blocks[i][j] / quan_table[j]
        blocks[i] = blocks[i].astype(int)
```

2.4 Source Coding

Source coding is very useful to lower the data rate further while no information is lost (lossless coding). Here an approach was used that is inspired by the jpg-format. First the difference of the DC-Component (the first coefficient in each block) to its predecessor is coded by size and amplitude. Then the AC-Components are grouped into a new alphabet containing the number of zeros before the symbol (runlength coding) and the size of the symbol itself. Also the two symbols EOB (End-of-block) and ZRL (Zero-run-length) are added to the alphabet. EOB is used to show that there are only zeros until the end of the block and ZRL is used to encode the maximum runlength of zeros in a row. The resulting symbols are then huffmancoded and to each symbol the amplitude is added.

```
def jpgrate(B):
        bpr = 15
        bits = 0

        # DC components
        dc_comp = [b[0] for b in B]
        dc_diff = np.ediff1d(dc_comp)
        dc_size = []
        for c in dc_diff:
                if c == 0: dc_size.append(0)
                elif abs(c) == 1: dc_size.append(1)
                elif abs(c) <= 3: dc_size.append(2)
                elif abs(c) <= 7: dc_size.append(3)
                elif abs(c) <= 15: dc_size.append(4)
                elif abs(c) <= 31: dc_size.append(5)
                elif abs(c) <= 63: dc_size.append(6)
                elif abs(c) <= 127: dc_size.append(7)
                elif abs(c) <= 255: dc_size.append(8)
                elif abs(c) <= 511: dc_size.append(9)
                elif abs(c) <= 1023: dc_size.append(10)

        # estimate rate for huffmancoding the size
        unique, counts = np.unique(dc_size, return_counts=True)
        dc_p= counts.astype(float)/len(dc_size)
        bits += huffmanrate(dc_p)*len(dc_size)
        bits += np.sum(dc_size)         # num of bits for amplitude

        # AC component
        comp = []
        for b in B:
                indx = np.flatnonzero(b[1:]) # find nonzero components
                indx_diff = np.ediff1d(indx) # find run lengths
                # create new alphabet
                for i in range(0,len(indx_diff)):
                        if indx_diff[i] == 1: comp.append([0, (int)(math.log(abs(b[indx[i
                            ]+1]), 2) +1)])
                        elif indx_diff[i] <= bpr+1: comp.append([indx_diff[i]-1, (int)(math.
                            log(abs(b[indx[i]+1]), 2)+1)])
                        elif indx_diff[i] > bpr+1:
                                for k in range(1, (int)(indx_diff[i]-1)/bpr):
                                        comp.append([bpr, 0]) # ZRL
                                comp.append([(indx_diff[i]-1)%bpr, (int)(math.log(abs(b[indx
                                    [i]+1]), 2)+1)])
                if len(indx) == 0:
                        comp.append([0,0]) # EOB
                elif indx[-1] != len(b)-2:
                        comp.append([0, 0]) # EOB

        # huffman coderate for pairs
        comp_num = [c[0]*(bpr+1)+c[1] for c in comp]
        unique, counts = np.unique(comp_num, return_counts=True)
        comp_p = counts.astype(float)/len(comp_num)
        bits += huffmanrate(comp_p)*len(comp_num)
        bits += np.sum([c[1] for c in comp]) # size=number of bits needed to code amplitude

        # calculate final code rate (without side information)
        rate = bits.astype(float)/np.size(B)

        return rate


# simple Huffman coding
def huffmanrate(p):
        ml=0
        nnz = np.count_nonzero(p)

        if nnz > 0:
                for k in range(0,nnz-1):
                        p = p[np.nonzero(p)]
                        idx = np.argsort(p)
                        b0 = idx[0]
                        b1 = idx[1]
                        p[b1] = p[b1] + p[b0]
                        p[b0] = 0
                        ml += p[b1]
                ml += 1
                print np.sum(p)

        return ml
```

2.5 Decoding

To decode the signal all the steps are reversed. First the quantized signal is reconstructed with the quantization table. Then the inverse DCT is applied on each block in order to get the reconstructed values as a sequence in the time domain. Finally the interleaving is reversed and the signal is split up into the two channels and written into a 16-bit wav file.
The SNR can be determined by calculating the mean-square-error (msqe) between the original and decoded signal and the variance of the original signal.

$$SNR = 10 \cdot \log_{10} \frac{\sigma_o^2}{msqe} \tag{2}$$

```
# Decoding
for i in range(0,len(blocks)):
        for j in range(0, bl_size):
                blocks[i][j] = blocks[i][j] * quan_table[j]                          #
                    reconstruction
        blocks[i] = fftpack.idct(blocks[i], norm='ortho')               # inverse dct
        blocks[i] = blocks[i].astype(int)
                                        # integer casting

blocks = np.ravel(blocks)

out = np.array([ np.array([blocks[i], blocks[i+1]], dtype=np.int16) for i in range(0,len(
    blocks),2)], dtype=np.int16 )
wavfile.write("out.wav", fs, out)
```

2.6 Sideinformation

The sideinformation required to decode the coded signal is quite small. Only the quantization table, the huffman codes for the DC-component and the AC-components. As all three tables are only needed once for all blocks, for a large number of blocks they can easily be neglected for the data rate.

3 Results

For the file *heyhey.wav* the best results that could be found were 32.88dB at a datarate of 192 kbps for a blocksize of 2046. For the file *nuit.wav* the best results were 40.84dB at a datarate of 192 kbps.

4 Quality of the Compression

The quality of the sound after the compression can be described as quite ok. One can definetly hear that the music lost its depth and sounds more harsh and metallic, but otherwise its close to the original file.
The lost data rate that is possible without hearing a difference to the original file (*heyhey.wav*) is 376 kbps. At this datarate the SNR is at 43dB.
On the other hand the lost data rate that is possible while one can still hear the song in a decent quality is 139 kbps with a SNR of 28.8dB.