Communications Lab 2 Report B05901092 歐瀚墨

1. Preface

My code is uploaded to the following link:

https://github.com/ouhanmo/Comm_Lab

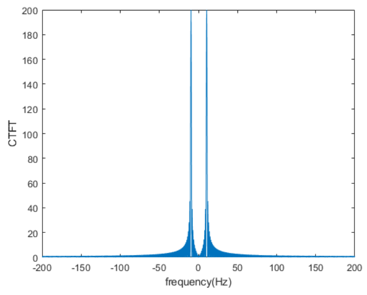
The main program is lab2.m

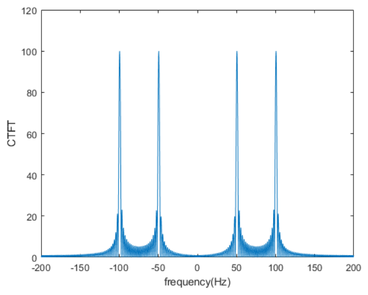
To run the program, please set the exper variable to the experiment number (1-4).

2. Experiment Results

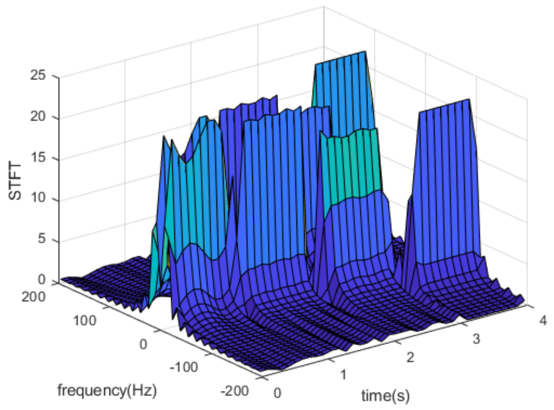
I. Short Time Fourier Transform

ab) Figure as follows:

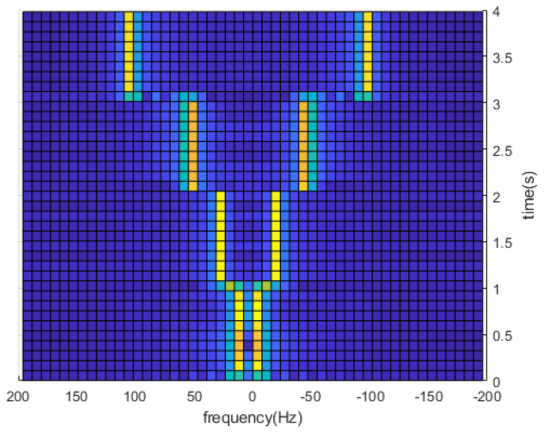




c) STFT of x is plotted below (window size = 50 Noverlap=10):



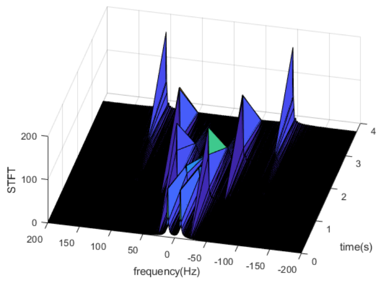
When observing from the top, the figure becomes:

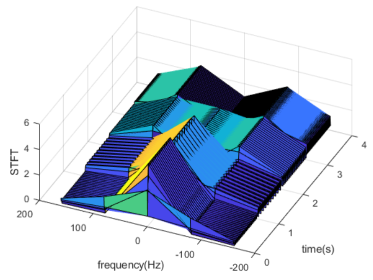


From the figure, we can observe the change of frequencies as t=1,2,3 s.

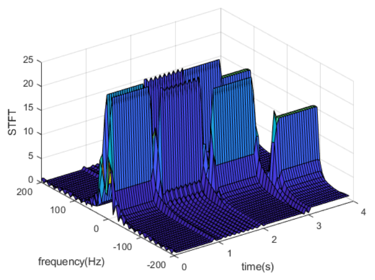
d) Window size

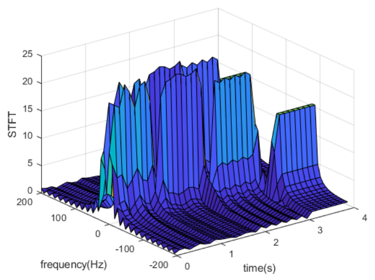
The left figure is window size 500, while the right corresponds to window size 5, it is clear that a larger window size provides better estimation of the frequencies, while small window sizes give us a cleaner cut when the frequency changes.





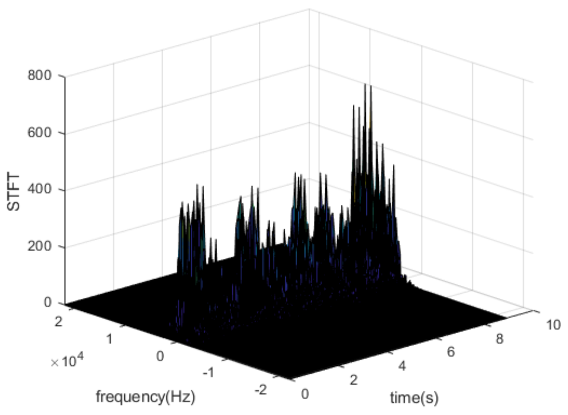
d) Noverlap





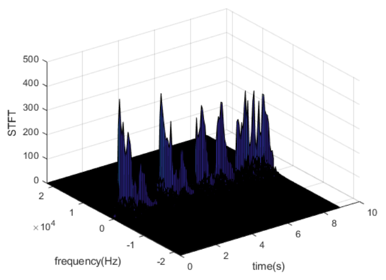
The left figure is when Noverlap = 30, the right one is when Noverlap = 4. It can be observed that larger Noverlap give us more samples and a smoother graph, amid the tradeoff of heavier computation.

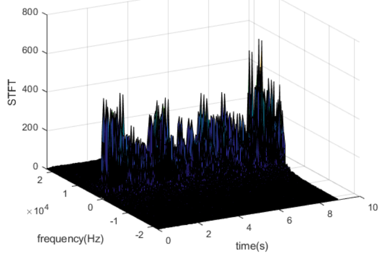
e) STFT of handel.ogg



The magnitude of the spectrogram matches the sounds we hear. However, after many tries, I couldn’t find a way to actually observe the frequency change in relation to the tone I hear.

Then I plotted the spectrograms of the squared handel.ogg and quantized handel.ogg, the resulting figures is as follows, respectively:





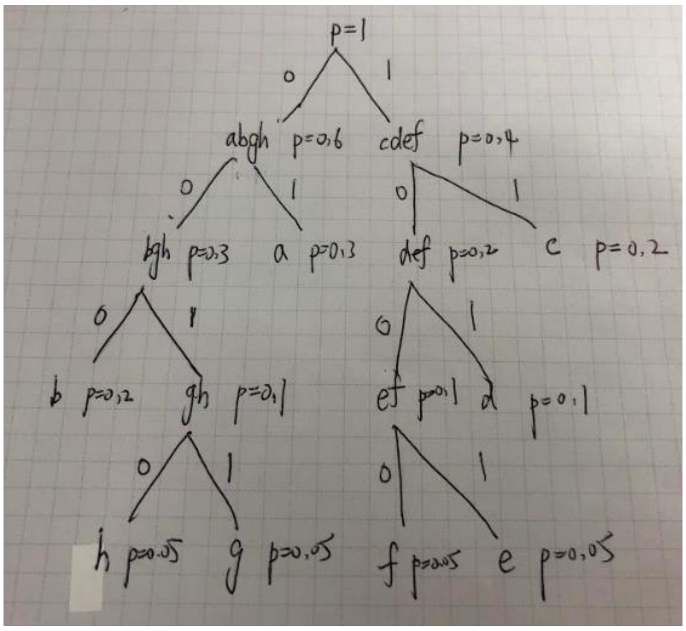
The spectrogram on the left shows that squaring the signal exaggerates the magnitude features, while neglecting the others, the frequency spectrum becomes more condensed towards the DC term.

The right figure is the 2-bit quantized result. The spectrum resembles the original signal, but with some extra noises on the frequency domain. The observation is concurrent with the sound after quantization.

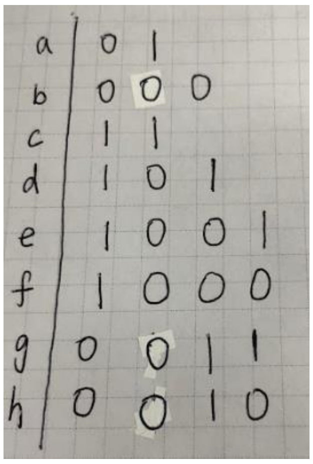
II. Huffman Coding

a) Construct Huffman Tree

The tree is constructed based on my algorithm in the matlab code. I take the smallest branch first, and merge them into the next smallest one. If there are two nodes with same probabilities, the first one is considered the smallest. Although my tree my seem unorthodox and counterintuitive, it is based on a strict algorithm.



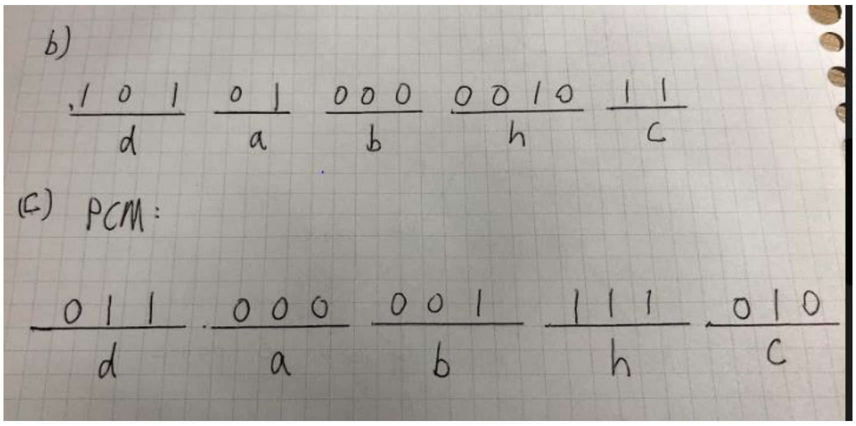
The resulting code:



a	0 1
b	0 0 0
c	1 1
d	1 0 1
e	1 0 0 1
f	1 0 0 0
g	0 0 1 1
h	0 0 1 0

b) Huffman Encoding by hand

c) PCM Encoding by hand



b)

1 0 1	0 1	0 0 0	0 0 1 0	1 1
d	a	b	h	c

(c) PCM:

0 1 1	0 0 0	0 0 1	1 1 1	0 1 0
d	a	b	h	c

d) Data Size

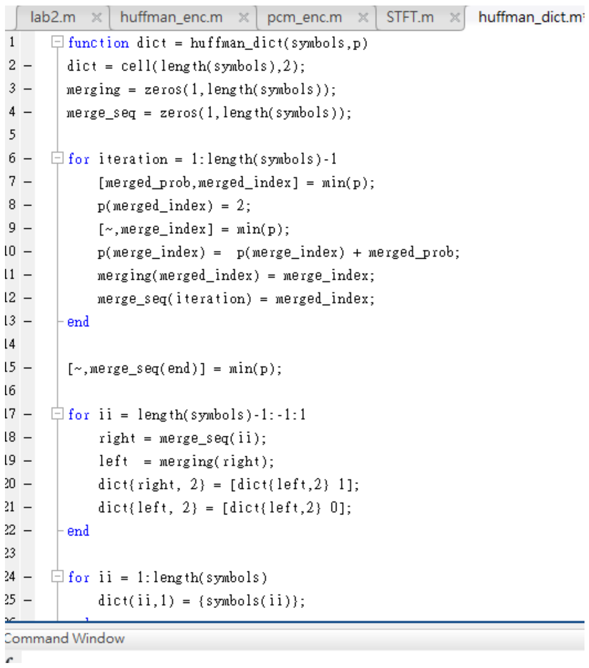
Huffman: 14 bits, PCM: 15 bits

Huffman is more efficient than PCM in this case.

f) Huffman Dict

I take the smallest branch first, and merge them into the next smallest one. If there are two nodes with same probabilities, the first one is considered the smallest. Then the algorithm merges the second smallest node into the smallest node. The merging node would be the 0 node, and the merged node would connect to the 1 path.

Below is the screenshot of my code.



```
function dict = huffman_dict(symbols,p)
dict = cell(length(symbols),2);
merging = zeros(1,length(symbols));
merge_seq = zeros(1,length(symbols));

for iteration = 1:length(symbols)-1
    [merged_prob,merged_index] = min(p);
    p(merged_index) = 2;
    [~,merge_index] = min(p);
    p(merge_index) =  p(merge_index) + merged_prob;
    merging(merged_index) = merge_index;
    merge_seq(iteration) = merged_index;
end

[~,merge_seq(end)] = min(p);

for ii = length(symbols)-1:-1:1
    right = merge_seq(ii);
    left  = merging(right);
    dict{right, 2} = [dict{left,2} 1];
    dict{left, 2} = [dict{left,2} 0];
end

for ii = 1:length(symbols)
    dict(ii,1) = {symbols(ii)};
```

First, I keep the information of the node each symbol is merged to, along with the order of symbols when they are merged. After merging, I start from the latest merged symbol and assign each symbol with its respective codeword along the way backwards.

The resulting dictionary of symbols a-h is as follows:

'a'	[0,1]
'b'	[0,0,0]
'c'	[1,1]
'd'	[1,0,1]
'e'	[1,0,0,1]
'f'	[1,0,0,0]
'g'	[0,0,1,1]
'h'	[0,0,1,0]

Which is identical to the answer in (a) but different from the built-in function.

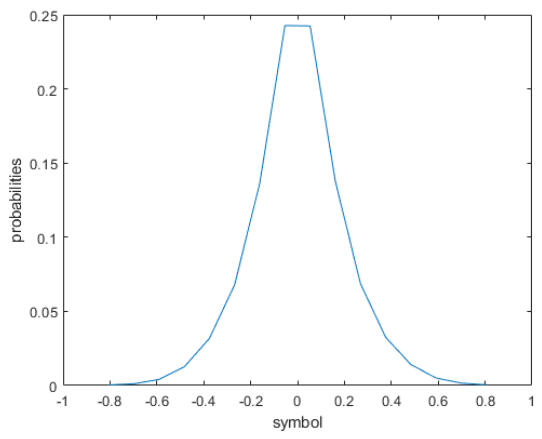
h) Decoding

My decoding algorithm relies on two “pointers”, “start” and “finish”. Starting from the beginning, if the bits between start and finish (ends inclusive) is a codeword, then I add the corresponding symbol to x_dec and then both pointers skip to the next bit after finish. If not, then finish would increase by 1.

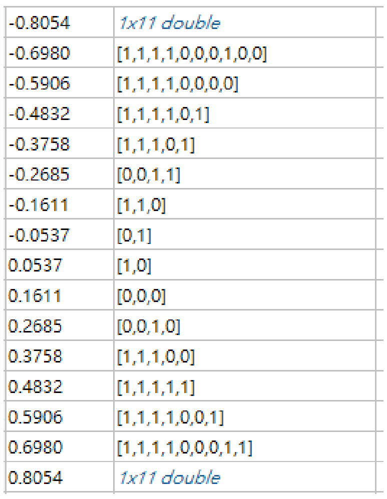
III. Converting Signals into Bits

In the following experiment, 4-bit(L=16) quantization is used.

a) Probabilities:



b) Map samples to bits



-0.8054	*1x11 double*
-0.6980	[1,1,1,1,0,0,0,1,0,0]
-0.5906	[1,1,1,1,0,0,0,0]
-0.4832	[1,1,1,1,0,1]
-0.3758	[1,1,1,0,1]
-0.2685	[0,0,1,1]
-0.1611	[1,1,0]
-0.0537	[0,1]
0.0537	[1,0]
0.1611	[0,0,0]
0.2685	[0,0,1,0]
0.3758	[1,1,1,0,0]
0.4832	[1,1,1,1,1]
0.5906	[1,1,1,1,0,0,1]
0.6980	[1,1,1,1,0,0,0,1,1]
0.8054	*1x11 double*

-0.8054 is mapped to [1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1]

0.8054 is mapped to [1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0]

Since Huffman encoding is straightforward, the code is rather short. The resulting y is as follows:

```
y                    1x1146074 double
```

c) PCM encoding

```
y_pcm                1x1574344 double
```

d) Data Size

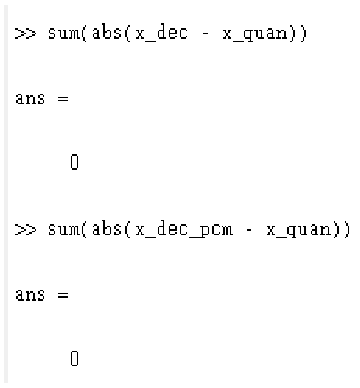
Huffman: 1146074 bits

PCM: 1574344 bits

Huffman encoding only used around 72.8% of bits compared to the simple PCM, thus the importance of such an encoding method.

e) Decoding

I wrote another function pcm_dec(y, symbols, numBits) to decode PCM. Both recovered handel.ogg are identical to the quantized version of handel.ogg. Which shows that PCM and Huffman codes can encode the signal without any loss. The proof of recoverable is below.



```
>> sum(abs(x_dec - x_quan))

ans =

     0

>> sum(abs(x_dec_pcm - x_quan))

ans =

     0
```

IV. Average Codeword Length of Huffman Codes

a) Entropy

$$\mathrm{H}(\{\mathrm{a,b,c,d,e,f,g,h}\}) = -\sum plog(p) = 2.6464$$

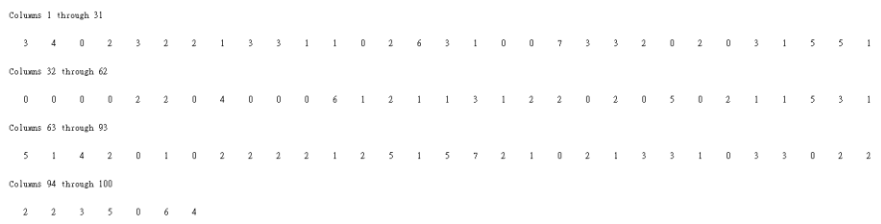
b) The Huffman Code Length

The Huffman code is shown in II(f). The average length is calculated:

$$\bar{\mathrm{L}} = 0.3 \times 2 + 0.2 \times 3 + 0.2 \times 2 + 0.1 \times 3 + 0.05 \times 4 + 0.05 \times 4 + 0.05 \times 4 + 0.05 \times 4 = 2.7$$

c) Length of encoded 100 symbols

I used 0-7 instead of a-h in order to call the randsrc function to generate sequences. A sample is as follows:



```
Columns 1 through 31

  3   4   0   2   3   2   2   1   3   3   1   1   0   2   6   3   1   0   0   7   3   3   2   0   2   0   3   1   5   5   1

Columns 32 through 62

  0   0   0   0   2   2   0   4   0   0   0   6   1   2   1   1   3   1   2   2   0   2   0   5   0   2   1   1   5   3   1

Columns 63 through 93

  5   1   4   2   0   1   0   2   2   2   2   1   2   5   1   5   7   2   1   0   2   1   3   3   1   0   3   3   0   2   2

Columns 94 through 100

  2   2   3   5   0   6   4
```

The length of encoded binary data for the above sequence is 269 bits. Since the average codeword length is 2.7 and sequence length 100. The expected total length is 270. This sample is rather close.

d) Repeat 1000 Times

$$\overline{\mathrm{L}_{100}} = \frac{1}{1000}\sum_{i=1}^{1000} L_{100}^{(i)} = 270.095$$

e) Compare Observed Length to Expected Length and Entropy

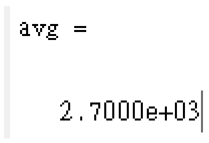
Entropy: 2.6464

Average Length: 2.7

$$\frac{\overline{\mathrm{L}_{100}}}{100} = 2.70095$$

The observed length per symbol is very close to the average length, a very trivial property. If PCM is used to encode the sequences, each sequence would need 300 bits to encode. However, after designing the Huffman Code. We can compress the sequence to 2.7 bits per symbol, a figure reasonable close to its entropy, the lower bound by theory.

f) n=10000 and R=10000



```
avg =


   2.7000e+03
```

$$\frac{\overline{\mathrm{L}_{1000}}}{1000} = 2.7000$$

3. Remarks:

The STFT part of this lab is quite interesting, as from the spectrogram I was able to explain why the audio sound like what we hear. Plotting 3D figures is also fun especially when colorful patterns appear on the figures. The Huffman Codes part of the lab includes encoding decoding and code design. I really enjoyed the Huffman decoding part since it has a very neat algorithm, and the easiness of programming such ideas in Matlab makes it quite a comfortable experience.