William Liem

CM 3065 – Intelligent Signal Processing Final Project Report

Exercise 2

In Exercise 2, I created an application that does lossless data compression on wav files. It is capable of encoding and decoding these wav files based on the Rice coding algorithm. To start this project, I have been provided with 2 wav files, being 'Sound1.wav' and 'Sound2.wav'.

Rice Coding is a compression method that uses the Rice algorithm to encode and decode data. It is commonly used in lossless data compression schemes. To be used in real life effectively, some preprocessing must be done before it is implemented. An example of Rice Coding usage in real life is using it on the residuals of FLAC compression.

The 1st part of this exercise is to load all the wav files and get their byte array values, sample rates, channels, and sample width. Then, I implemented the rice encoding and rice decoding algorithms into 2 functions based on these instructions:

Rice's algorithm

Encoding

1. Fix an integer value K.
2. Compute the modulus, M by using the equation $M = 2^K$
3. For S, the number to be encoded, find
   A. quotient = $q = int(S/M)$
   B. remainder = $r = S modulo M$
4. Generate Codeword
   A. The Quotient_Code is q in unary format.
   B. The Remainder_Code is r in binary using K bits.
   C. The Codeword will have the format <Quotient_Code> <Remainder_Code>

Decoding

1. Determine q by counting the number of 1s before the first 0.
2. Determine r reading the next K bits as a binary value.
3. Write out S, the encoded number, as q × M + r.

Figure 1 Rice Coding algorithm from Coursera

```python
def rice_encoder(S, K):
    # Compute Modulus
    M = 2**K

    # quotient
    q = S // M

    # generate the codeword
    quotient_code = ""
    for i in range(q):
        # Add 1 q number of times.
        quotient_code += "1"
    quotient_code += "0"

    # remainder
    remainder = S % M
    # generate remainder code
    remainder_code = bin(remainder)[2:].zfill(K)

    # put together the encoded string
    code_word = quotient_code + remainder_code

    # print(code_word)
    return code_word
```

Figure 2 Implementing Rice Encoding in Python

After coding the Rice algorithms, they have to be implemented and called correctly in order to actually be applied on the wav files. It is in this wav encoding and decoding process that the wav files' byte array values, sample rates, channels, and sample widths are used to implement the Rice algorithms properly. For the encoding, the byte array values are used. For decoding, the rest of the data is used. The wav encoding function will output an ex2 file, while the decoding function will output wav files.

The table below shows the results of the effectiveness of the Rice coding algorithm based on different variables. (KB = Kilobytes)

	Original Size	K=2	K=4	% Compress K=2	% Compress K=4
Sound1.wav	978 KB	33.131 KB	12.824 KB	3385 %	1310 %
Sound2.wav	984 KB	34.957 KB	13.291 KB	3551 %	1350 %

Based on the table above, it can be concluded that a longer bit length is more effective. It can be seen that the smallest ex2 files is when K = 4, and the largest K = 2. It also shows that the compression percentages are very high. The ex2 files are much larger than that of the originals. The reason for this is the wav files were inserted into the algorithms without preprocessing.

The additional implementation I did was to improve the previous Rice Coding algorithms. The arrays from the audio bytes contain values from 0 - 255. But a lot of them are 3 digits. So, I split the digits up if the integer contains 2 or more digits. The table below shows the results of the further implementation. (KB = kilobytes)

	Original Size	K=2	K=4	% Compress K=2	% Compress K=4
Sound1.wav	978 KB	8.377 KB	11.508 KB	856 %	1176 %

Sound2.wav	984 KB	9.435 KB	12.935 KB	959 %	1314 %

Although not much improvement was achieved for the compression percentage of K = 4, great improvement was achieved for K = 2, reaching compression rates of under 1000 %. This happens because K=2 on the previous algorithm was very inefficient. It reached bit lengths of around 60 digits when encoding 3-digit integers. The new algorithm manages to encode it into only around 10-20 bits in length.

Coursera Lab Link:
https://hub.labs.coursera.org/connect/sharedzmyxsiob?forceRefresh=false&isLabVersioning=true