ECE F344: Information Theory and Coding
Assignment-1

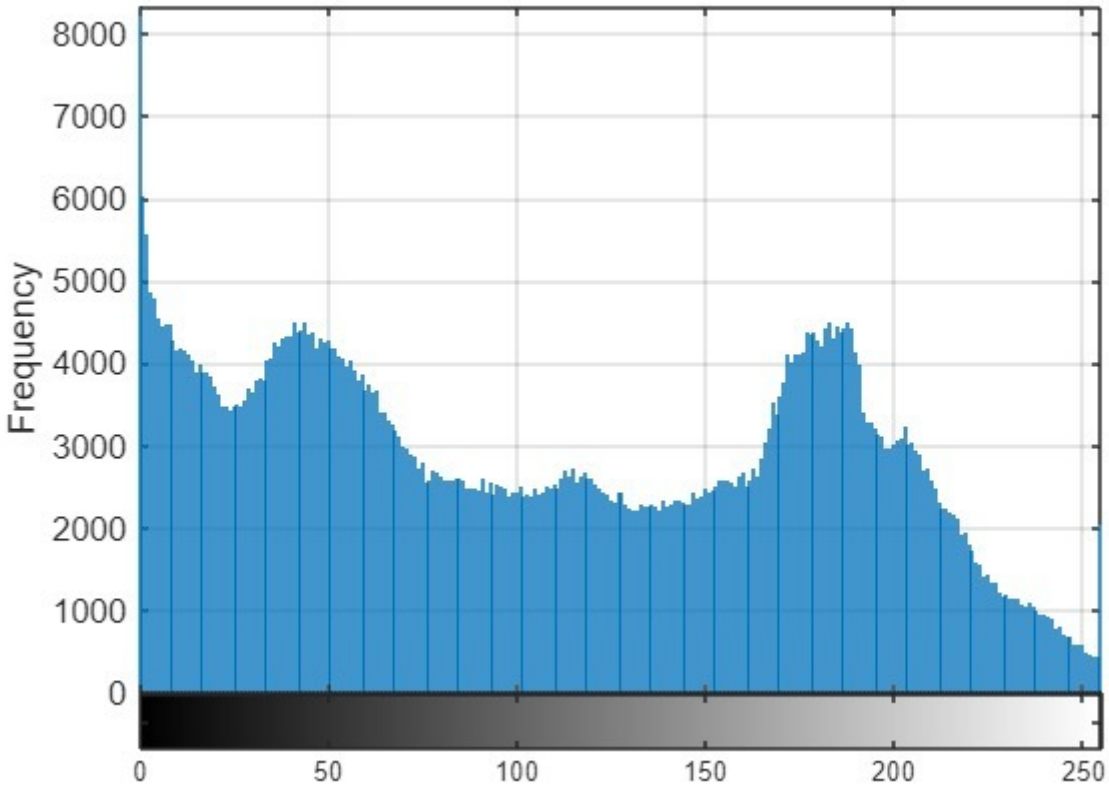
Group Details:

Satvik Sardesai	2020AAPS1417H
Rahul Karna L K	2020AAPS0437H
Thathapudi Sanjeev Paul Joel	2020AAPS0120H
Meghana Bellamkonda	2020AAPS0395H
Harish Yuvaraj G.P	2020AAPS1735H

Test Image:

Tasks and Results:

Histogram of pixel colors



Probability of Occurrence of Colours

This is calculated by first evaluating the frequencies of pixel colors in the image and then dividing this value by the total number of pixels in the image.

Refer to the Excel file named “huffman_code” for the resulting symbol probabilities for the chosen image.

Huffman Coding

To keep track of the Huffman tree's roots as we proceed, i.e. update all the children symbols that belong to a particular parent, cell arrays have been used in this assignment. Basically, two or more symbols are clubbed together as an array and treated as a single symbol.

To calculate the efficiency of the Huffman Code, the following relation was used:

$$Efficiency\ (\eta)\ = \frac{Source\ Entropy}{Average\ Codeword\ Length}$$

The following results were obtained for the above image:
Entropy of source letters: 7.8431
Average codeword length: 7.8823
Efficiency: 99.5026

Refer to the excel file named "huffman_code" for the codebook for individual symbols.

Lempel-Ziv Algorithm

Two approaches have been used for constructing the dictionary.

- A traditional approach in which the phrases are first extracted from the bitstream and then the dictionary is made.
- The dictionary entries are filled and codewords are generated in parallel while extracting the phrases.

The results from both approaches are documented below.

Note: Since the algorithm is time-consuming and requires a lot of processing, the image has been resized to limit execution time.

Resized image : (60 X 100 pixels)

Traditional approach

bitstream = 1×48000

1 1 0 1 0 1 1 0 1 1 0 1 ...

c = 1×4543 cell

	1	2	3	4	5	6
1	1	[1,0]	[1,0,1]	[1,0,1,1]	0	[

code = 1×4543 cell

	1	2	3	4	5	6
1	"00000000...	"00000000...	"00000000...	"00000000...	"00000000...	"000000

CR = 1.3250

A snippet of the results is shown above. The image is first converted into a bitstream and then fed into the algorithm.

Total number of bits = 60*100*8

= 48000 bits (size of the variable 'bitstream')

'c' is a cell array which is used to store the phrases as and when they are observed in the bitstream

'code' stores the codewords of the respective phrases.

For example, the phrase [1 0 1 1] is associated with the codeword [1 1 1].

CR (Compression Ratio) is the ratio of number of bits after running it through the algorithm, to the number of bits in the original bitstream.

For the traditional approach,

CR = 1.3250

Second approach (Real-time Lempel Ziv)

bitstream = 1×48000

1 1 0 1 0 1 1 0 1 1 0 1 ...

c = 1×4543 cell

	1	2	3	4	5	6
1	1	[1,0]	[1,0,1]	[1,0,1,1]	0	[

code = 1×4543 cell

	1	2	3	4	5	6
1	"01"	"10"	"101"	"111"	"0000"	"0100"

CR = 1.1544

For real-time Lempel Ziv,

CR = 1.1544

Run Length Encoding

We have interpreted this problem in two ways:

- Treating each individual pixel color as a symbol and thus finding out the run lengths based on this assumption.
- Considering the image as a stream of bits and treating 0 and 1 as the only two symbols.

Pixel colors as symbols:

Code flow:-

1. Read an image file into MATLAB and convert it to grayscale using the **rgb2gray** function.
2. Flatten the image into a 1D vector of pixel values using the **(:)** notation
3. Loop through the pixel values, counting consecutive pixels with the same value and storing the corresponding run-length encoded values and counts in arrays.
4. Add the last run of pixels to the arrays.
5. Compute the compression ratio

The resulting compression ratio for the above image was as follows:

Compression Ratio: 12.9159

Refer to the file “RLE_pixels.txt” for the list of run lengths, run values, and corresponding codewords.

Binary symbols:
In this approach as the name suggests, we used runs of symbols 0 and 1 as the basis for coding the image. For the image chosen the compression ratio achieved was greater than 1:

Compression Ratio: 4.9322

Refer to the file “RLE_binary.txt” for the list of run lengths, run values, and corresponding codewords.

This might be because there are only a few runs in the image that are very large in value causing the bits used to represent a run to be unnecessarily very large.

To understand the performance of the codes, we tested out with the following image below which has a large amount of black color in it:

The performance of both the approaches then turned out as follows:

Pixel colors as symbols:

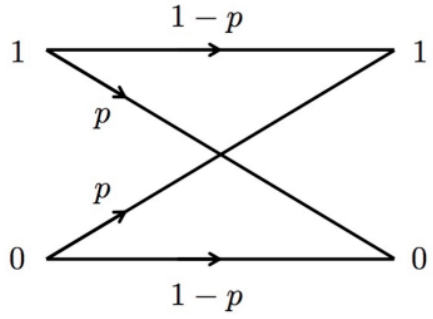
Compression Ratio: 0.77393

Binary symbols:

Compression Ratio: 0.29478

Binary Symmetric Channel (BSC)

In this assignment, we have modeled a binary symmetric channel as a system that generates a random number between 0 and 1, and based on the value we have allocated p (probability of error) parts of this range for flipping the received bit while in the other (1 - p) parts the channel transmits the bit received as it is.



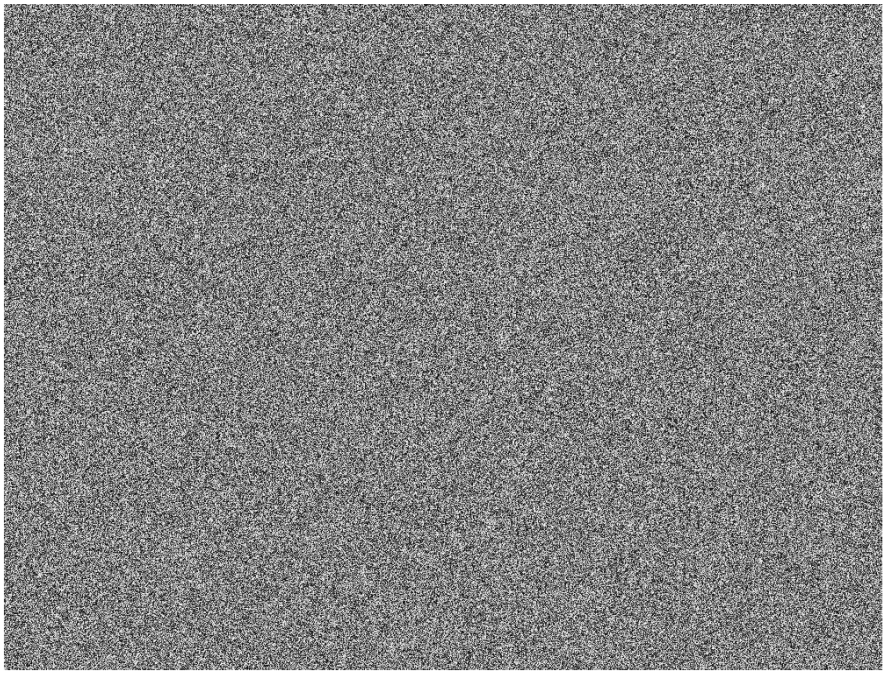
What our code does:-

1. Firstly we create a lookup table of strings. We save all bytes from 0 to 255 as strings. This is handy because directly accessing the image bytes and modifying them is tough.
2. Now we implement "*2 for loops*" going through all the image bytes one by one. (Image is a 2D Matrix)
3. Inside the *for loops* we first get the corresponding string of the image byte using the lookup table.
4. Next we use a bsc function created and in the same folder of our assignment (instead of using the inbuilt bsc function). This takes in the string and the probability of error (p) as inputs and gives the output byte by randomly flipping some bits based on given p.
5. We thus keep updating each element in the received image matrix following the same process in each iteration.
6. Finally we display this received image matrix.

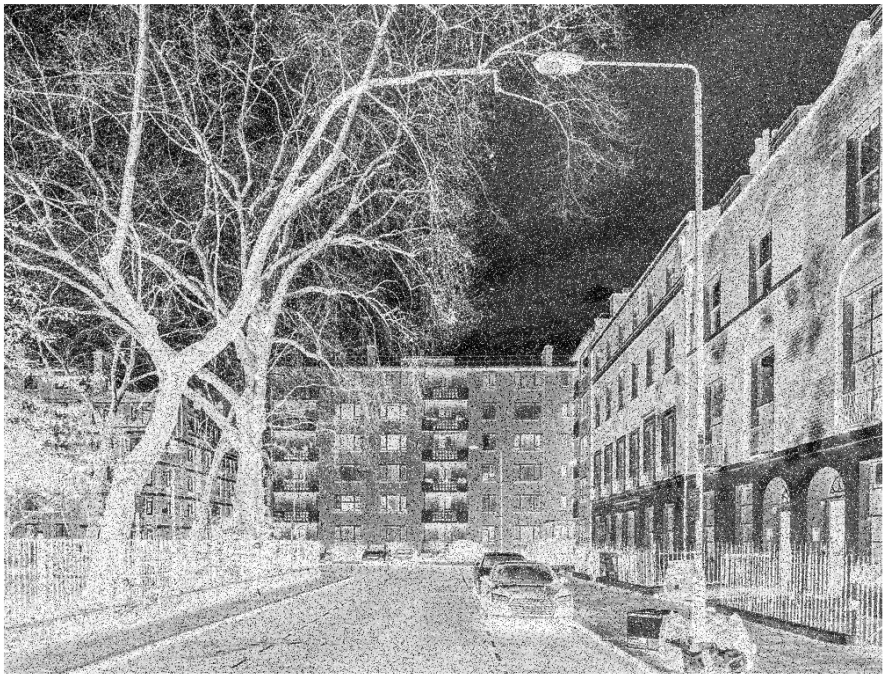
The images obtained after transmitting the above test image through a bsc for different error probabilities are:-

Error Probability = 0.1:

Error Probability = 0.5:



Error Probability = 0.9:



Parity Encoding and Decoding

We have used two approaches to implement the parity encoder and decoder, with one approach(Refer to Question7_1.mlx) using the available inbuilt functions that are highly optimized(Memory, Time) and the other approach(Refer to Question7_2.mlx) without inbuilt function.

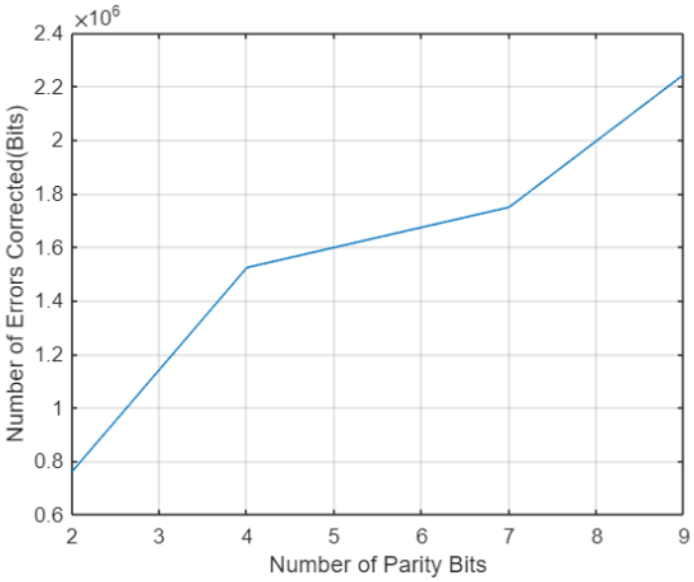
Algorithm:

- The message words are obtained from the image with 8bits(1 pixel) taken as the message word in Approach 1, with message word length varying with the generator matrix in Approach 2.
- Create the generator polynomial, which will be used for later calculations.
- Use this to generate the parity matrix
- Use the parity matrix to generate the generator matrix
- Encode the data using the generator matrix
- Pass it through BSC channels as required.
- On the transmitter end, decode the data using the generator matrix.
- Results obtained in intermediate steps are used to obtain the number of error corrected and error detected bits.

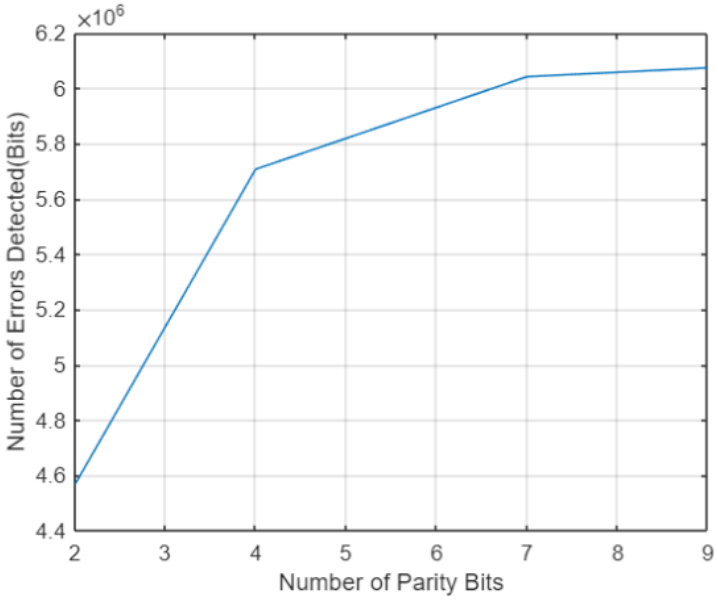
Functions Used:

- **encode(out,n,k,'linear/binary',genmat):** This function is used to encode the data using linear block code (n,k). One of the input arguments also has the generator matrix.
- **decode(ndata,n,k,'linear/binary',genmat):** This is used to decode the data after transmission through the Binary symmetric channel. The input arguments resemble those of the encode function.
- **bsc:** Used to get output of the data after transmission through a binary symmetric channel.
- **cyclpoly(n,k):** Returns the generator polynomial for a cyclic block code(n,k).
- **cyclgen(n,pol):** Returns a parity check matrix for the given code.
- **gen2par(parmat):** This function is used to find the generator matrix of the code by using the parity check matrix which will be given as an input argument to the function.

Results:



Errors Corrected vs Parity Bits



Errors Detected vs Parity Bits