DCT-based JPEG compression:

Project Documentation

I. INTRODUCTION:

Image compression is a necessary part of the storage and transfer processes of graphical data. Computers represent images as multidimensional arrays of pixel values. That said, It is not difficult to see how the storage complexity can add up for higher resolution images and larger numbers of them given the standard pixel by pixel representation. Furthermore, transferring graphical data between devices across wired or wireless networks is typically a slow process that consumes a lot of valuable bandwidth. Hence the importance of compression. Compression is the process of reducing a file's size for ease of storage and transfer, without compromising the data integrity and quality. In the context of image data, there are two main types of compression algorithms; ***lossy***, and ***lossless***. In simple terms, **Lossless compression** relies on changing the encoding scheme of the data so as to minimize all forms of redundancy and represent each symbol with the fewest possible number of bits. This way, the original data can be perfectly recovered in the decompression process. **Lossy compression** can achieve much higher compression rates by sacrificing the finer details in the image and exploiting the properties of the human visual system- with the condition that the quality remains acceptable.

II. Preliminary- the concept of 2d image frequency:

Normal 1-dimensional frequency represents the rate of change of the sample magnitudes in a signal overtime, with rapidly changing signals having higher frequency and vice versa. Similarly image frequency is the rate of change of pixel values across space instead of time in the X and Y directions. A high frequency image is one where the pixel values experience significant increases or decreases in magnitude moving from one pixel to the next, while a low frequency image has a smaller change in pixel values across space. In normal images, the low frequency components tend to dominate. High frequency components will typically be in the solid lines and edges between surfaces in the image, and too much of it can appear as noise. The human eye is more sensitive to the low frequency components in the image, and not so much to high frequency components.

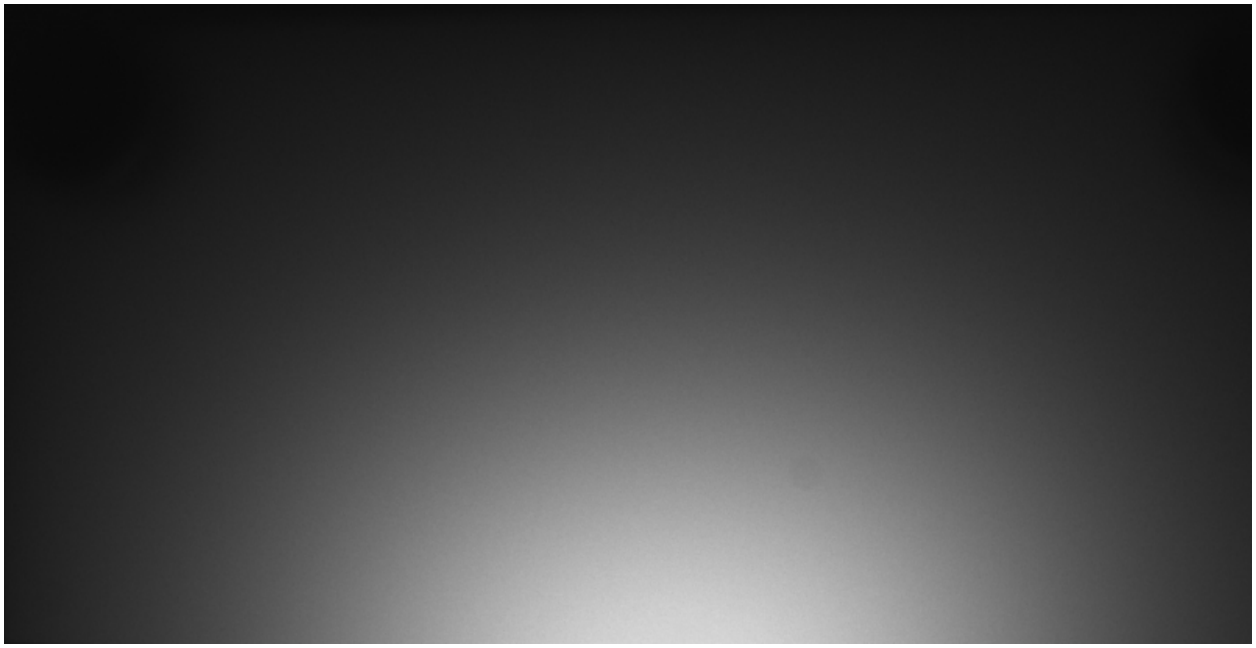

Figure 1: example of a low frequency image.

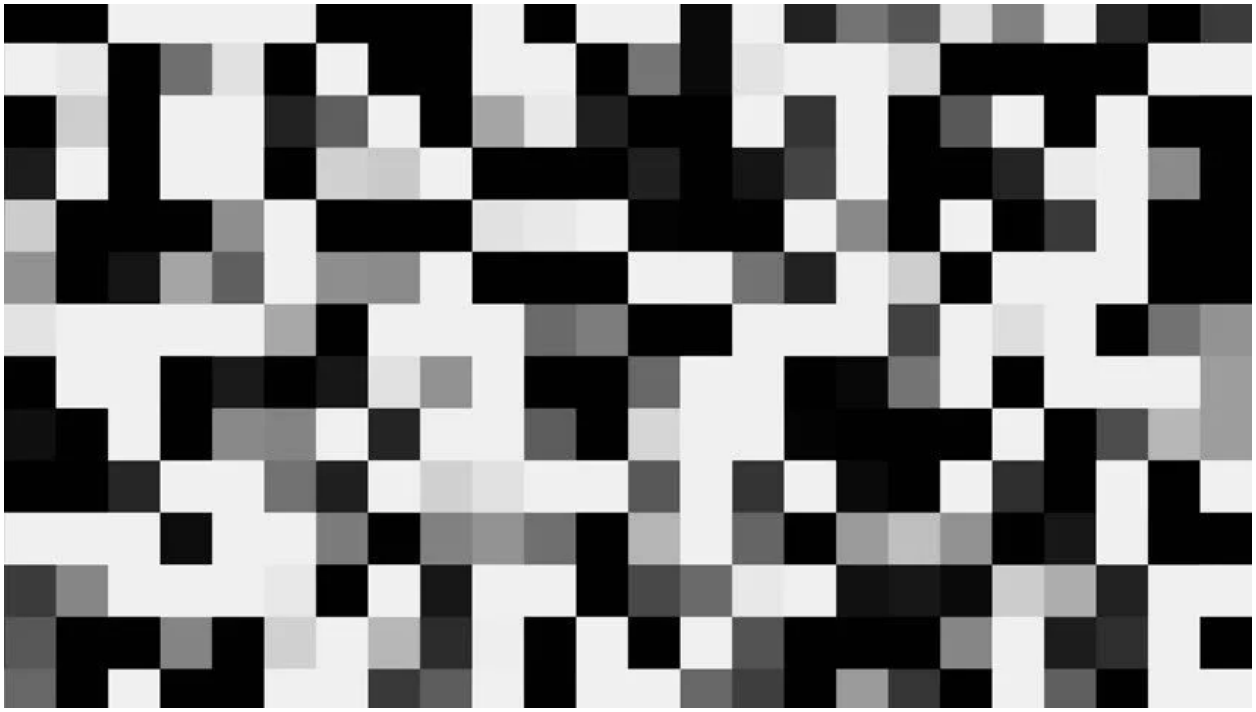

Figure 2: example of a high quality image

III. Preliminary- RGB and YCbCr color spaces:

Colored images can be stored inside computers in many representations. The most popular is the **Red-Green-Blue (RGB)** format. In RGB each pixel's color is represented as a combination of red, green, and blue color values ranging from 0 to 255. Together these 3 values can express 16,777,216 different colors with the combination (0,0,0) expressing black and (255,255,255) expressing white. An RGB image is a 3 dimensional array that has the color channels (R,G,b) in the 3rd dimension and the spatial dimensions of the image (rows,columns) in the first and second dimensions.

Figure 3: RGB image example

Another color representation scheme is YCbCr. Here, the image is represented as a 3 dimensional array with 3 channels on the last dimension. One channel represents the luminance (Y); which is the variation in light and dark information (black and white image), and the two other channels represent the red and blue chrominance respectively. The green chrominance information is implicitly encoded in these 3 channels and can be interpolated from them when converting to RGB.

Figure 4: YCbCr image example

IV. The JPEG algorithm- principle of operation :

Joint Photographic Experts Group (JPEG) is a **lossy** compression algorithm defined in the standards ISO/IEC 10918, ITU-T T.81, ITU-T T.83, ITU-T T.84, ITU-T T.86 [1]. JPEG performs good quality lossy compression by exploiting the fact that the human eyes don't perceive the high frequency components of an image very well, while being very sensitive to the low frequency components. With this in mind, the algorithm trims off much of the high frequency components while keeping most of the low frequencies intact. This produces a final compressed image with very little decrease in visual quality but with a much smaller size. In colored images, we also exploit the desensitivity of the eye to red and blue chrominance intensities to achieve significant compression by down sampling the chrominance components by a factor of 2 (or more).

V. The JPEG algorithm Pipeline :

1- For compression:

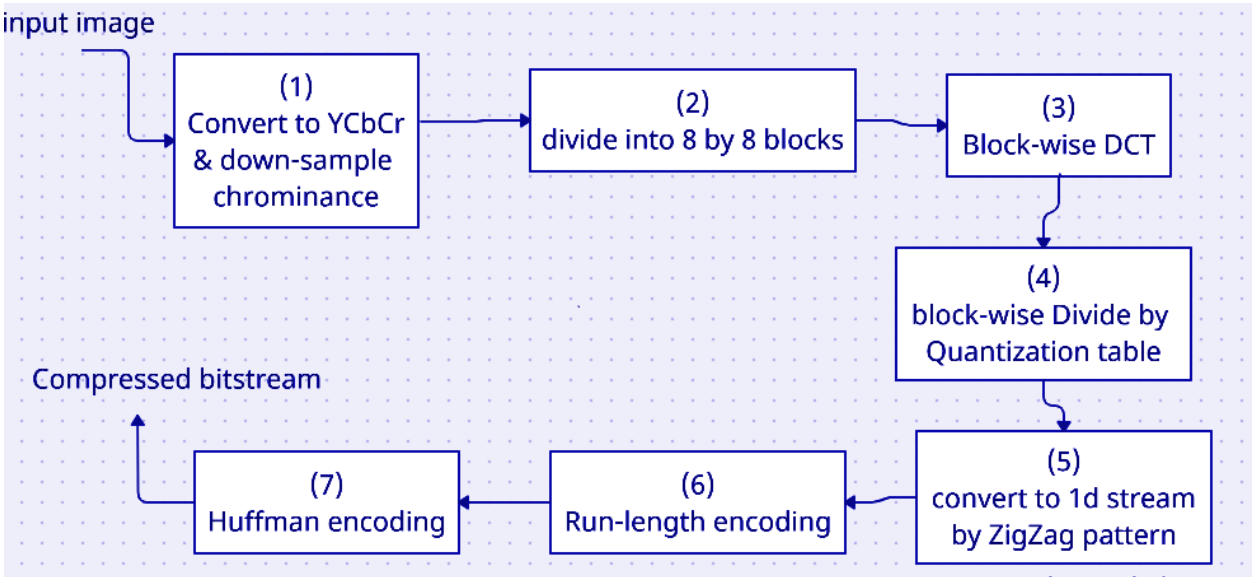

Fig 5: the JPEG compression pipeline

For RGB images, we start by converting them to YCbCr and downsampling the chrominance components by a factor of 2. We then divide up our image into 8 by 8 blocks and take the discrete cosine transform of each block. We then element-wise divide each block of DCT coefficients by an 8 by 8 quantization table and round to the nearest integer values. The quantization step is actually the lossy part of the compression because it results in zeroing out most of the high frequency DCT coefficients at the end of each block. The next step is to convert these blocks into a long 1-dimensional stream of numbers by stacking the numbers in a zig-zag pattern resulting in a long train of zeros at the end. The number stream is then run-length encoded to compress all of the trailing zeros which reduces the size significantly. Afterwards, the number stream is encoded into a bit stream by Huffman encoding to result in a very efficient, prefix codes. The resulting binary file is the compressed image. The huffman dictionary used in the entropy encoding is also saved as a binary file.

2- For decompression:

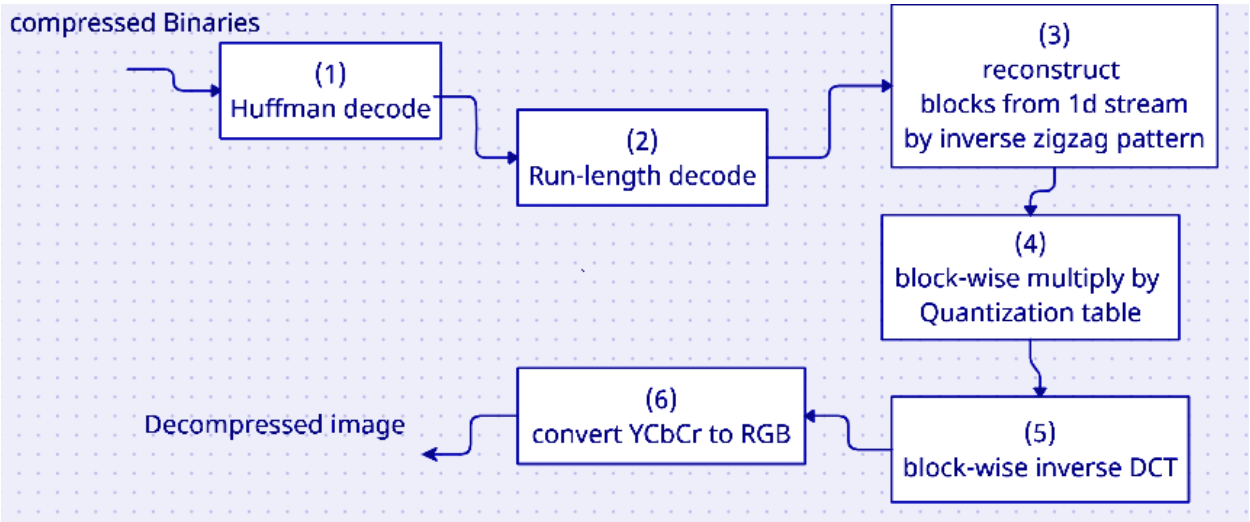

Fig 5: the JPEG decompression pipeline

The decompression pipeline centers around undoing all of the transformations conducted at compression. The input is the compressed bit stream and the huffman dictionary and the output is the reconstructed image. The bitstream is first Huffman decoded, then run length decoded to retrieve the quantized DCT coefficients. The 1 dimensional integer array is then reconstructed into 2-dimensional 8by8 DCT coefficients blocks by inverting the Zig-Zag pattern we used in compression. Each block is then multiplied by the same quantization table used in compression. When compressed the DCT coefficients were divided by the quantization table and rounded to the nearest integer, and in compression these rounded values are re-multiplied by the quantization table which will produce ever so slightly different values from what we started with and this is where the lossy distortion comes. We then take the inverse discrete cosine transform (IDCT) to retrieve the pixel values from DCT coefficients. The original sized image is then re-constructed from the 8by8 blocks and converted back to RGB after upsampling the chroma components by interpolation.

VI. The discrete cosine transform DCT [2]:

Much like the discrete fourier transform, the discrete cosine transform is a frequency domain transformation that uses cosines of different frequencies as its bases functions instead of complex exponentials. For image compression, we are interested in calculating the 2-dimensional DCT which uses 64 2-dimensional (8 by 8) cosine basis functions. Arranged in a grid, The first (upper-leftmost) basis is a zero frequency cosine which is the dc component and the basis x, and y frequencies increase as we move to the right and down.

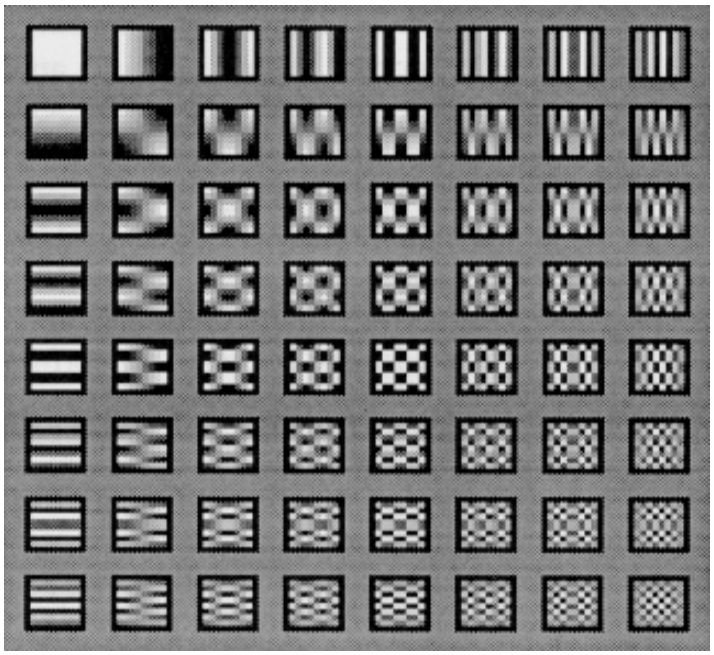

Figure 6: the DCT basis functions

The following formula is used to calculate the DCT of an image:

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

$$\text{for } u = 0,\ldots,N-1 \text{ and } v = 0,\ldots,N-1$$

$$\text{where } N = 8 \text{ and } C(k) = \begin{cases} 1/\sqrt{2} & \text{for } k = 0 \\ 1 & \text{otherwise} \end{cases}$$

Where F(u,v) is the u^{th}, v^{th} DCT coefficient, f(x,y) is the image pixel value at index (x, y). The image size is M by N, or in our case N by N where N=8, to run the DCT on 8by8 image blocks. The output of the DCT is an 8 by 8 grid with the 64 coefficient values. The inverse DCT converts these coefficients back into an image of pixel values following the equation:

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)F(u,v) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

$$\text{for } x = 0,\ldots,N-1 \text{ and } y = 0,\ldots,N-1 \text{ where } N = 8$$

VII. Quantization:

Quantization is simply the process of element-wise dividing the DCT coefficients by a standard quantization table. Depending on the block-size of DCT different standard quantization tables of corresponding sizes exist. All quantization tables are designed to have smaller integer numbers at the upper leftmost corner with increasing numbers in the bottom rightmost direction. What this does is penalize the higher frequency components while leaving the lower frequencies mostly intact. Since the higher frequencies in the DCT at the bottom rightmost corners will be divided by large numbers then rounded to the nearest integer, this leaves us with mostly zeros in place of the high frequencies. The quantization table format is the largest determinant of the compression degree.

If more aggressive compression is desired, a table with very large numbers in most of the high frequency locations will guarantee that most of the high frequency components in the image are cut out, at the expense of causing a blurring effect in the final image. A more conservative quantization table will only cut off the highest few frequencies and mildly penalize the intermediate frequencies while keeping the low frequencies virtually unchanged.

Here is an example of a low compression 8by8 Q.table:

$$Q_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 \\ 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 \\ 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 \\ 1 & 1 & 1 & 1 & 2 & 2 & 4 & 8 \\ 1 & 1 & 2 & 2 & 2 & 2 & 4 & 8 \\ 2 & 2 & 2 & 2 & 2 & 4 & 8 & 16 \\ 2 & 2 & 2 & 4 & 4 & 8 & 8 & 16 \\ 4 & 4 & 4 & 4 & 8 & 8 & 16 & 16 \end{bmatrix}$$

And a high compression 8by8 Q.table:

$$Q_8 = \begin{bmatrix} 1 & 2 & 4 & 4 & 8 & 16 & 32 & 128 \\ 2 & 4 & 4 & 8 & 16 & 32 & 64 & 128 \\ 4 & 4 & 8 & 16 & 32 & 64 & 128 & 128 \\ 8 & 8 & 16 & 32 & 64 & 128 & 128 & 256 \\ 16 & 16 & 32 & 64 & 128 & 128 & 256 & 256 \\ 32 & 32 & 64 & 128 & 128 & 256 & 256 & 256 \\ 64 & 64 & 128 & 128 & 256 & 256 & 256 & 256 \\ 128 & 128 & 128 & 256 & 256 & 256 & 256 & 256 \end{bmatrix}$$

Here's an example of a 16 by 16 low compression table:

$$Q_{16} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 5 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 & 4 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 4 & 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 & 16 \\ 1 & 1 & 2 & 2 & 4 & 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 & 8 & 8 & 16 \\ 1 & 1 & 2 & 2 & 4 & 4 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 16 & 16 & 16 \\ 1 & 2 & 2 & 4 & 4 & 8 & 8 & 8 & 8 & 8 & 8 & 16 & 16 & 16 & 16 & 16 \\ 1 & 2 & 2 & 4 & 4 & 8 & 8 & 8 & 8 & 8 & 16 & 16 & 16 & 16 & 16 & 16 \end{bmatrix}$$

Another example of a high compression 16 by 16 table:

$$Q_{16} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 2 & 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 & 8 & 8 \\ 1 & 1 & 1 & 1 & 2 & 2 & 2 & 4 & 4 & 8 & 8 & 8 & 8 & 16 & 16 & 32 \\ 1 & 1 & 1 & 1 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 16 & 16 & 16 & 16 & 32 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 16 & 16 & 16 & 32 & 32 \\ 1 & 1 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 & 16 & 16 & 32 & 32 & 64 \\ 1 & 1 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 8 & 8 & 16 & 16 & 32 & 32 & 64 \\ 1 & 1 & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 & 8 & 16 & 32 & 32 & 64 & 64 \\ 1 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 & 8 & 16 & 32 & 32 & 256 & 256 \\ 1 & 2 & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 & 32 & 32 & 32 & 64 & 256 & 256 \\ 1 & 16 & 16 & 16 & 32 & 32 & 32 & 32 & 32 & 32 & 32 & 32 & 32 & 256 & 256 & 256 \\ 1 & 16 & 16 & 16 & 4 & 4 & 4 & 16 & 16 & 32 & 32 & 64 & 128 & 256 & 256 & 256 \\ 4 & 16 & 32 & 64 & 64 & 64 & 64 & 128 & 128 & 128 & 128 & 128 & 256 & 256 & 256 & 256 \\ 4 & 16 & 32 & 64 & 64 & 64 & 64 & 64 & 128 & 128 & 128 & 256 & 256 & 256 & 256 & 256 \\ 32 & 64 & 64 & 64 & 128 & 256 & 256 & 256 & 256 & 256 & 256 & 256 & 256 & 256 & 256 & 256 \end{bmatrix}$$

As mentioned above, the element wise division of the DCT block by the quantization table accompanied with rounding operation results in a long trail of zeros in the rightmost bottom corner of the output. As the next step is serializing the output 2d block into a 1d number stream by a zigzag pattern, all of the zeros end up in a consecutive trail at the end of the stream.

Here's an example of an 8 by 8 quantized block (C) retrieved from a DCT coefficients block D and a quantization table Q:

$$C_{i,j} = round\left(\frac{D_{i,j}}{Q_{i,j}}\right)$$

$$C = \begin{bmatrix} 10 & 4 & 2 & 5 & 1 & 0 & 0 & 0 \\ 3 & 9 & 1 & 2 & 1 & 0 & 0 & 0 \\ -7 & -5 & 1 & -2 & -1 & 0 & 0 & 0 \\ -3 & -5 & 0 & -1 & 0 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The ZigZag serialization pattern:

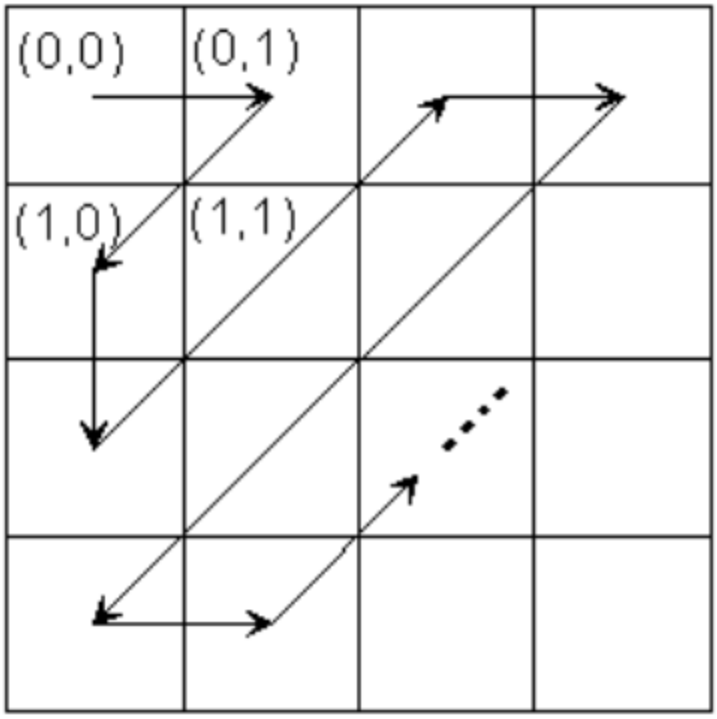

Figure 7: the serializing Zigzag pattern

Which results in an integer stream C_{flat} :

[10,4,3,-7,9,2,5,1,-5,-3,-2,-5,1,2,1,0,1,-2,0,1,0,0,0,0,-1,-1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0]

Notice the trailing zeros at the end; these will be combined in the next step of run-length encoding which will reduce the storage size significantly. At the decompression stage, each reconstructed 8 by 8 block will be multiplied by the same quantization table to retrieve the DCT coefficients, however, the resulting coefficients will not be exactly the same as the original due to the lossy nature of the rounding process.

VIII. Encoding:

1- Run-length encoding:

Before carrying out the entropy encoding we conduct run-length encoding on the resulting symbol stream first. Run-length encoding is a simple form of lossless compression in which sequences of identical symbols are represented by only two values, the symbol value and its repetition count.

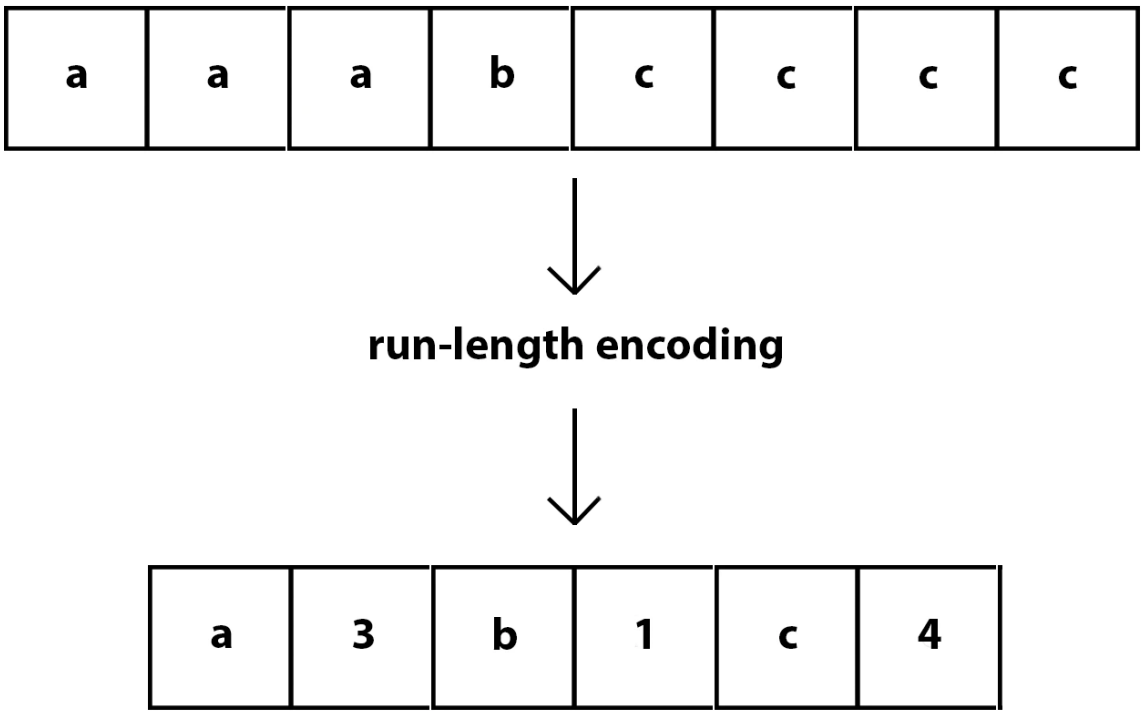

Figure 8: simple demonstration of Run-length encoding

1- Huffman encoding:

Huffman code is an entropy coding scheme used to convert the image symbols into bits (0s and 1s). It can be used to losslessly compress data because it guarantees that each symbol is represented with the bare minimum number of bits while maintaining the prefix property and unique decodability. It uses a greedy algorithm that minimizes the number of bits in the representation of the most frequent symbols. For a more detailed explanation of the algorithm refer to [3].

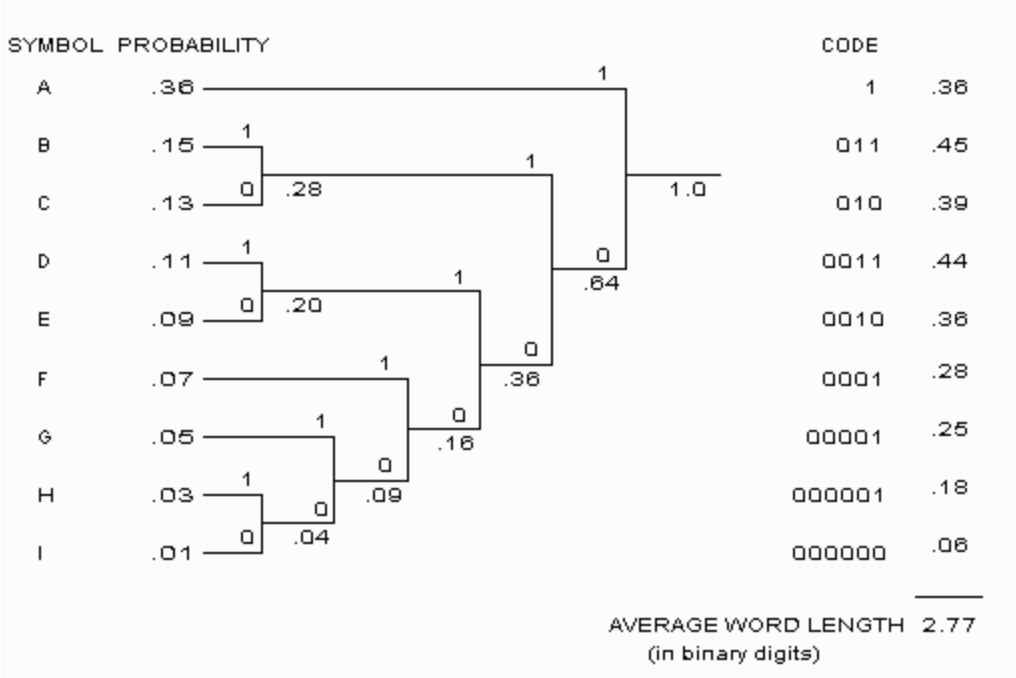

Figure 9: example of Huffman encoding algorithm tree based on symbol frequencies

The huffman encoding process results in the final encoded bitstream as well as a huffman dictionary of each symbol and its binary representation. In the decompression stage, the huffman decoder uses this dictionary to convert each binary code back to the original symbol.

IX. Performance Criteria in Image Compression [4]:

We evaluate the effectiveness of the compression algorithm by estimating the **compression ratio** as follows:

CR = n1/n2

Where the compression ratio (CR) is the ratio of the compressed image size n1 and the original image size n2.

We evaluate the visual quality of the compressed image by defining **a lossy distortion factor** that is calculated as a simple **mean absolute error (MAE)** between the compressed image and the original image pixel values.

$$MAE = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} |X(i,j) - Y(i,j)|$$

To evaluate the time complexity of our codes and how it relates to the input size and block size; we counted the number of Floating point operations (FLOPS) in both the compression and decompression main functions.

X. Experiments and results:

To test our code against the aforementioned performance measures; we ran a total of 8 test cases with different input image types: (grayscale vs RGB), different compression degrees: (high compression, vs low compression), and different options for the image blocks (8 by 8 blocks as per the JPEG standard, and 16 by 16 to investigate the effect of changing the block size). We used the quantization tables listed as examples above for the 8 by 8, and 16 by 16 high and low compression cases.

Here are the reference images used:

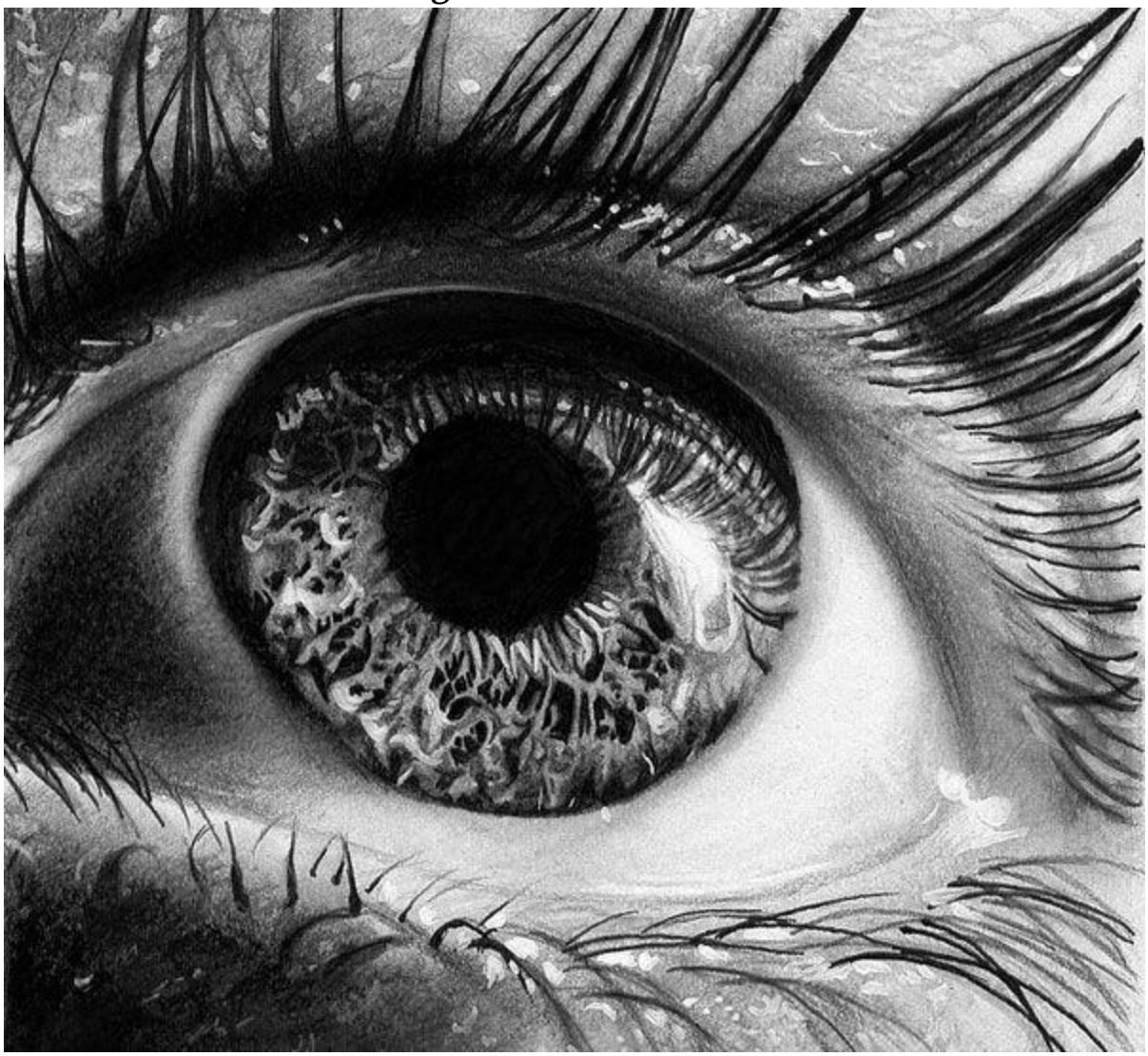

Figure 10: reference grayscale image

Figure 11: Reference RGB image

Experiment 1: Grayscale_image, blocksize = 8, Low-compression:

Figure 12: experiment 1, compressed image

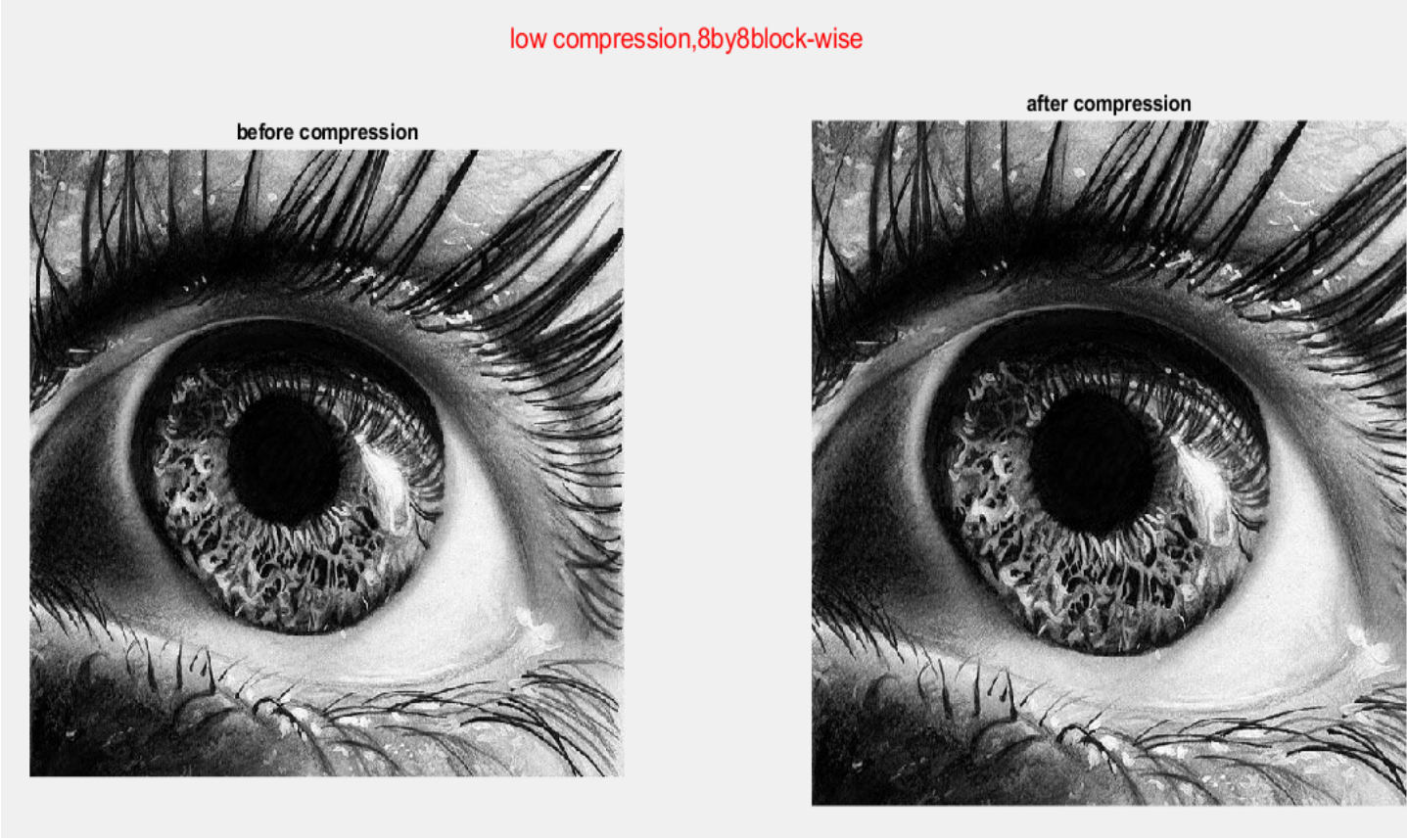

Figure 13: side by side comparison of the original and reconstructed image

Experiment 2: Grayscale_image, blocksize = 8, HIGH-compression:

Figure 14: experiment 2, compressed image

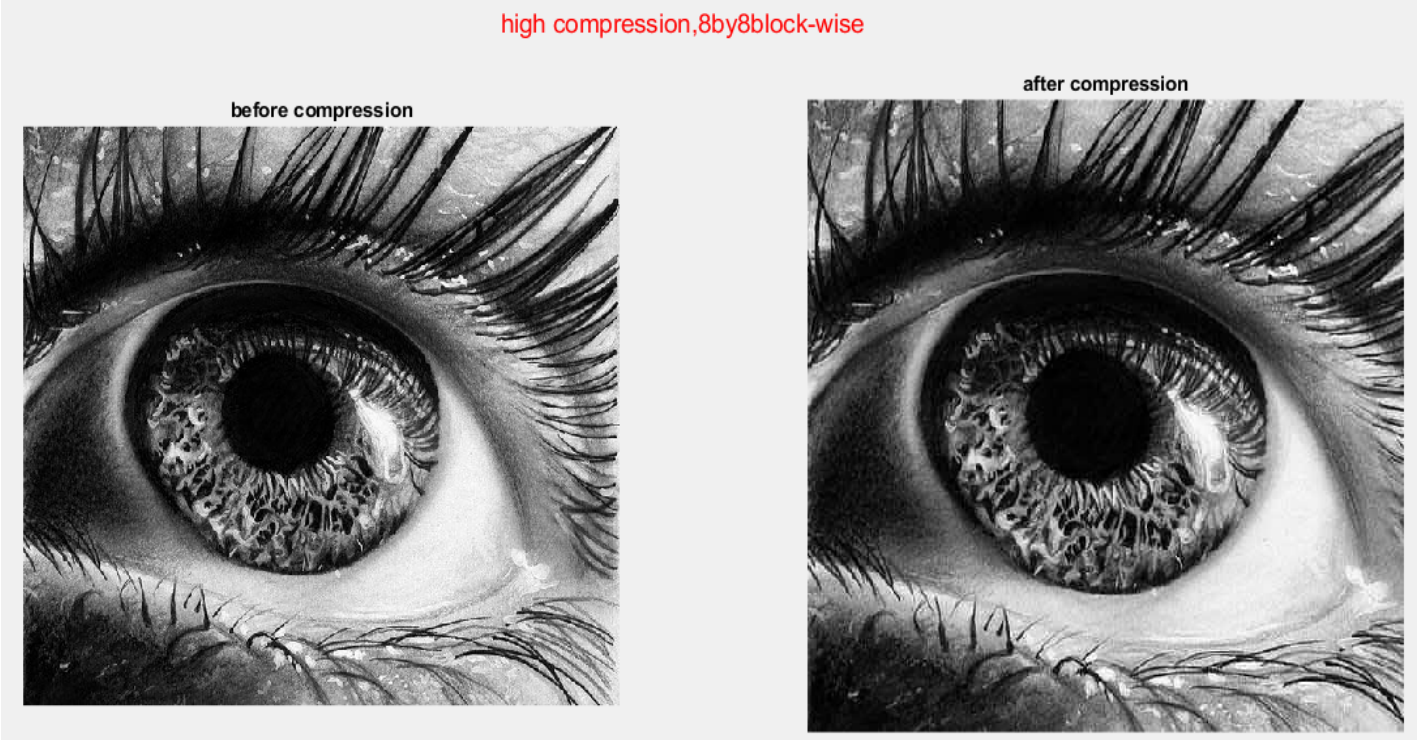

Figure 15: side by side comparison of the original and reconstructed image

Experiment 3: Grayscale_image, blocksize = 16, LOW-compression:

Figure 16: experiment 3, compressed image

Figure 17: side by side comparison of the original and reconstructed image

Experiment 4: Grayscale_image, blocksize = 16, HIGH-compression:

Figure 18: experiment 4, compressed image

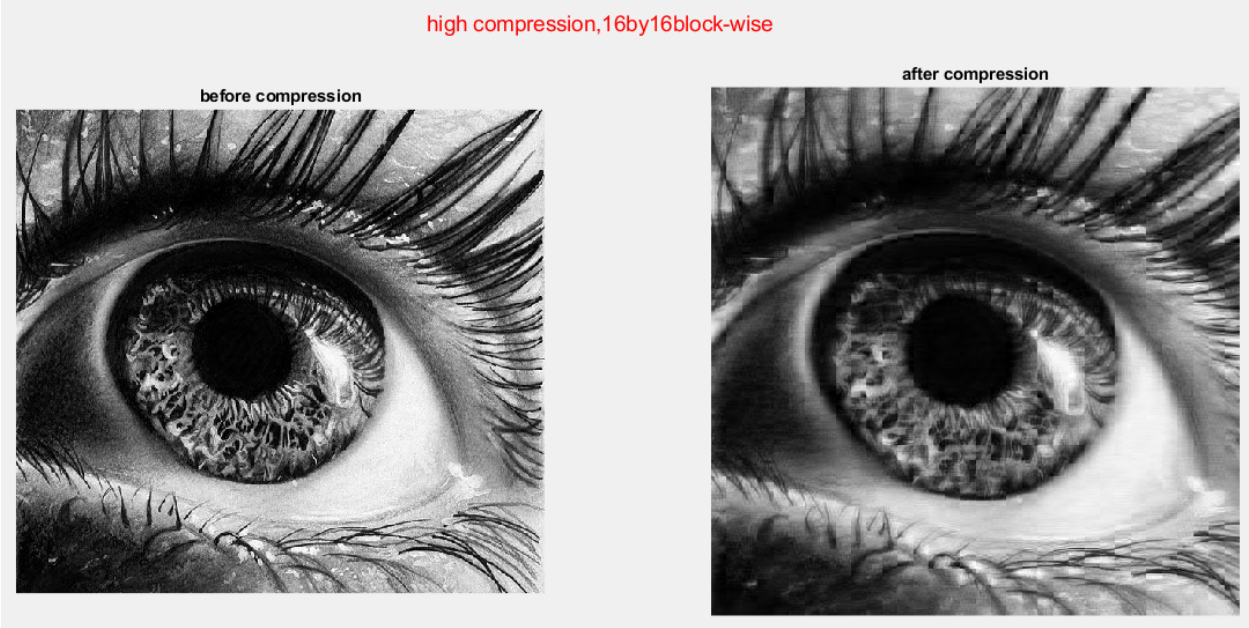

Figure 19: side by side comparison of the original and reconstructed image

Experiment 5: RGB_image, blocksize = 8, Low-compression:

Figure 20: experiment 5, compressed image

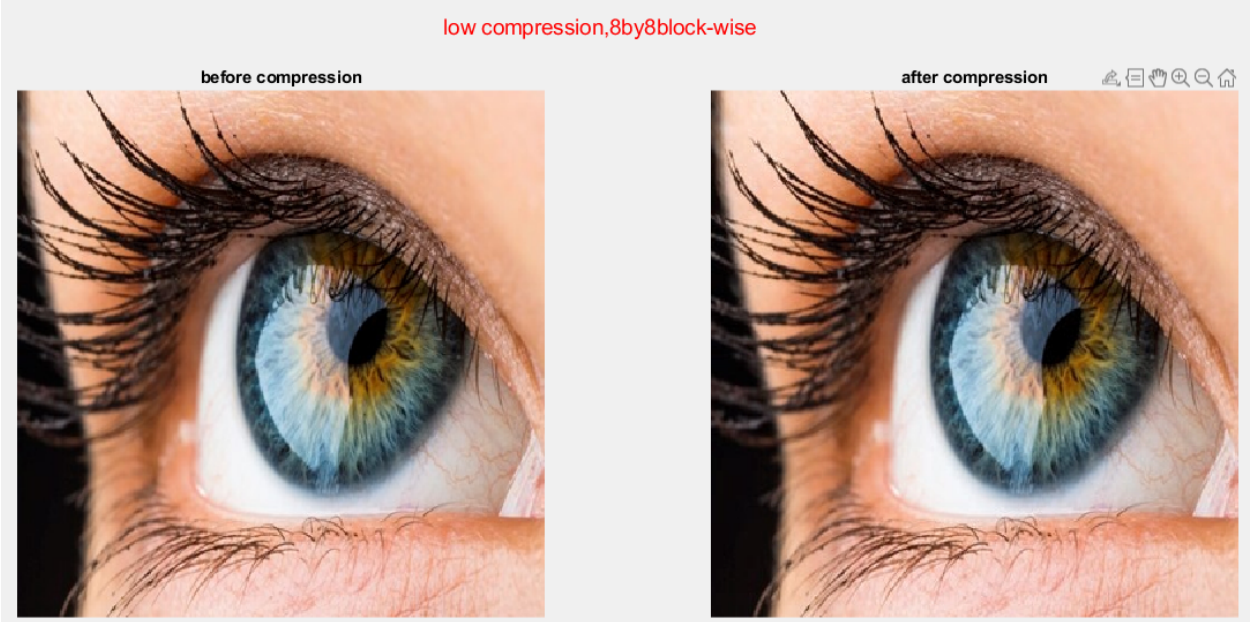

Figure 21: side by side comparison of the original and reconstructed image

Experiment 6: RGB_image, blocksize = 8, HIGH-compression:

Figure 22: experiment 6, compressed image

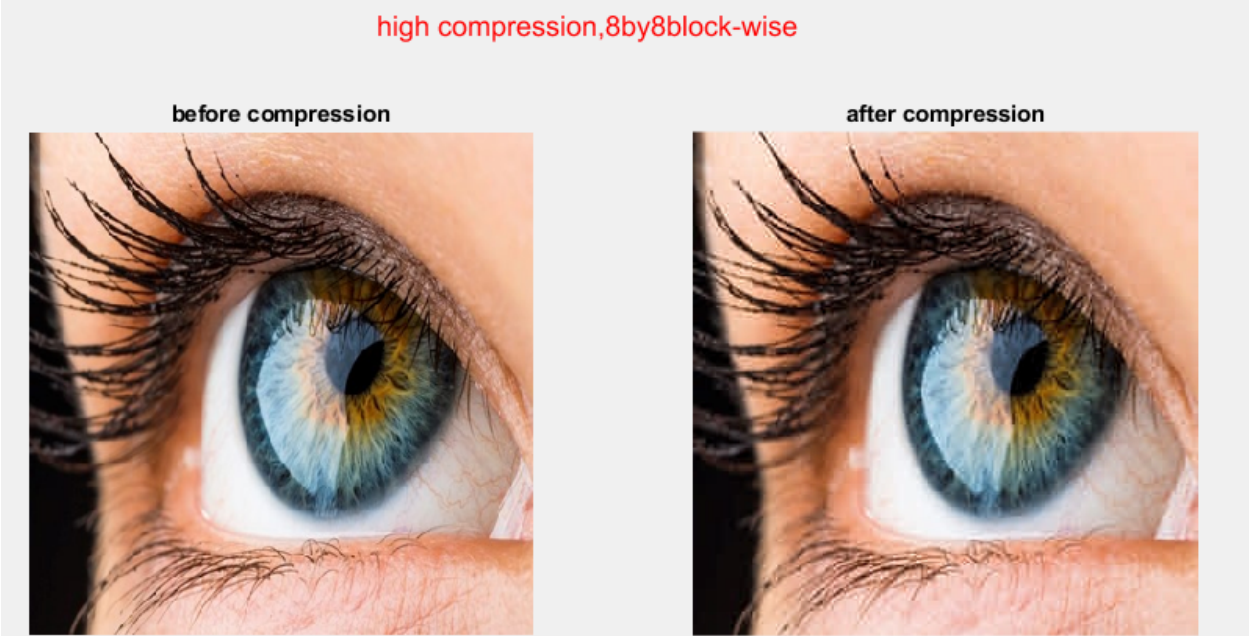

Figure 23: side by side comparison of the original and reconstructed image

Experiment 7: RGB_image, blocksize = 16, LOW-compression:

Figure 24: experiment 7, compressed image

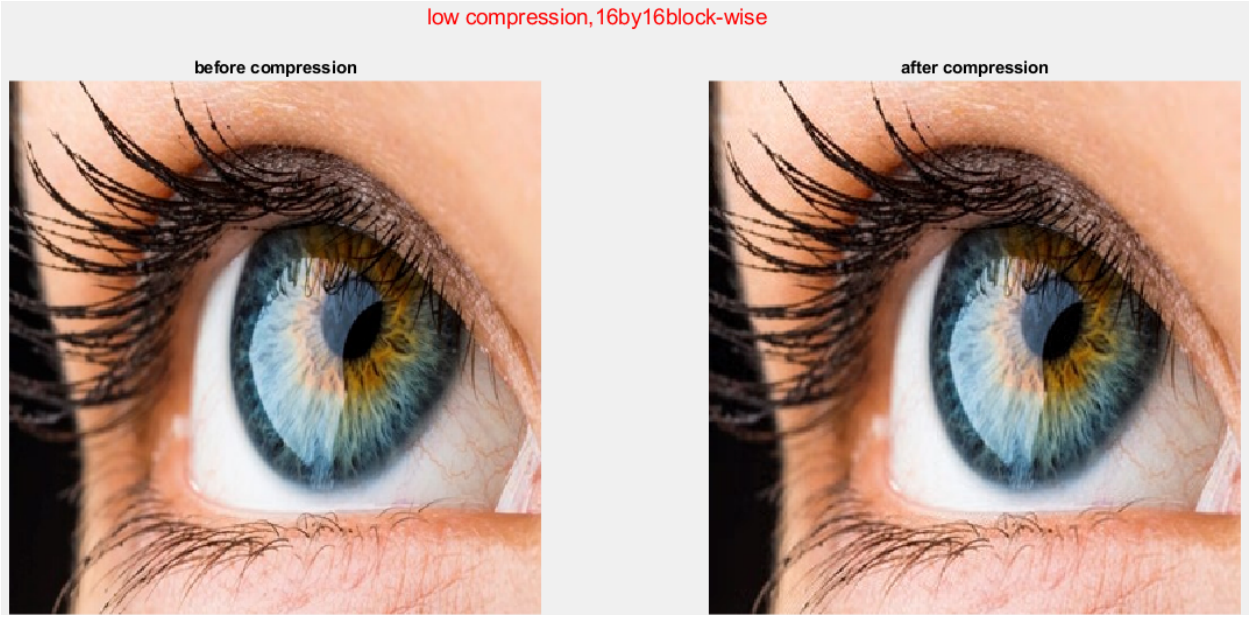

Figure 25: side by side comparison of the original and reconstructed image

Experiment 8: RGB_image, blocksize = 16, HIGH-compression:

Figure 26: experiment 8, compressed image

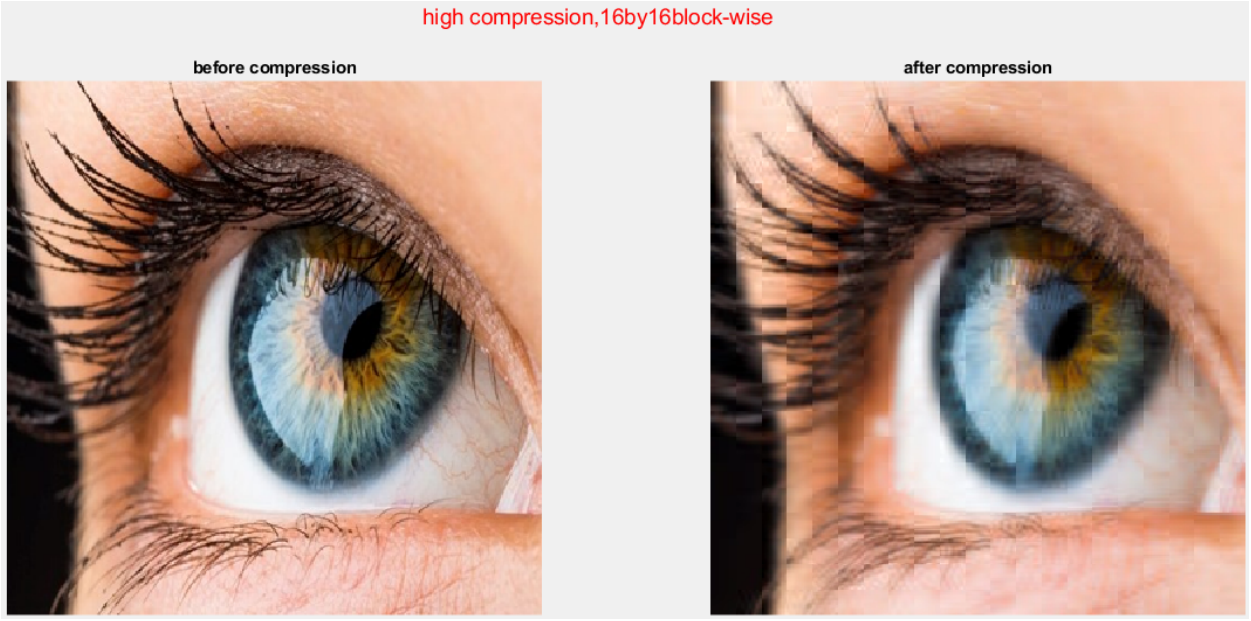

Figure 27: side by side comparison of the original and reconstructed image

From the previous experiments we can see by qualitatively assessing the compressed images that the low compression quantization tables provide the least image distortion. Using the high compression tables reduces the image quality by adding a blurring effect; which is expected given that this translates to elimination of the high frequencies in the image and thus eliminating the fine details. Increasing the block size from 8 to 16 results in a further decrease in quality by causing the image to look patchy. This is because larger blocks of the image are being deprived of their high frequency components at once resulting in a lower perceived quality. We note then that the best quality image, as expected, is the 8 by 8 low compression results, and the worst quality is the 16 by 16 high compression.

Summarized performance statistics:

For the detailed statistics of all experiments, refer to the text files in the '***performance_statistics***' folder in the project files.

1- Reconstruction Quality (MAE):

8 by 8 blocksize, with the low compression quantization matrix:
MAE= 0.7164

8 by 8 block size with the high compression quantization matrix:
MAE=5.208

16 by 16 blocksize with the low compression quantization matrix:
MAE= 2.273

16 by 16 blocksize with the high compression quantization matrix:
MAE= 7.3

As expected, we find that the high compression tables always produce a higher error than their low compression counterparts. Similarly, the 16 by 16 blocksize yields higher errors than the corresponding 8 by 8 case.

2- Compression ratios:

We used two interpretations of the original and output image sizes to measure the compression ratios. First of all, we compared the lengths of the arrays before and after entropy encoding, and measured the length-wise compression ratio as:

$$\mathbf{CR_{LEN}} = \frac{(length\ uncompressed - length\ compressed) \times 100}{length\_compressed}$$

Second of all, we compared the disk sizes of the pre- and post-compression image files.

The Size-wise c. ratio:

$$\mathbf{CR_{SZ}} = \frac{(size\ uncompressed - size\ compressed) \times 100}{size\_compressed}$$

As per these compression metrics we got the following results:

<u>8 by 8 block-wise with the low compression quantization matrix:</u>

- Array length C. ratio: 34.5%
- Disk size C.ratio: (from 96.1 kb to 86 kb) 10.5%

<u>8 by 8 block-wise with the high compression quantization matrix:</u>

- Array length C. ratio: 76.23%
- Disk size C.ratio: (from 96.1 kb to 49 kb) 49.95 %

<u>16 by 16 block-wise with the low compression quantization matrix</u>

- Array length C. ratio: 36.57%
- Disk size C.ratio: (from 96.1 kb to 83 kb) 13.63%

<u>16 by 16 block-wise with the high compression quantization matrix</u>

- Array length C. ratio: 77.97%
- Disk size C.ratio: (from 96.1 kb to 43.5 kb) 54.7%

We generally note a significant increase in compression ratio using the high compression Q-tables as compared to the low compression ones; this is attributed to the increase in the number of tail zeros that result after quantization which are efficiently compressed by RL encoding.

3- Time Complexity (FLOPS):

8 by 8 block-wise with the low compression quantization matrix:
FLOPS_compressJPEG = 285565974
FLOPS_decompressJPEG = 284899410

8 by 8 block-wise with the high compression quantization matrix:
FLOPS_compressJPEG = 285413692
FLOPS_decompressJPEG = 284609440

16 by 16 block-wise with the low compression quantization matrix:
FLOPS_compressJPEG = 1137833451
FLOPS_decompressJPEG = 1137161765

16 by 16 block-wise with the high compression quantization matrix:
FLOPS_compressJPEG = 1137756018
FLOPS_decompressJPEG = 1137013508

As can be noted from the recorded number of flops, using a larger block-size dramatically increases the number of FLOPS. We believe this can be attributed to our DCT and IDCT iterative algorithms, because each of them uses 4 nested for loops, which makes their complexity approach $\mathbf{O(n^4)}$.

XI. References:

[1]Jpeg.org. n.d. JPEG - JPEG 1. [online] Available at: <https://jpeg.org/jpeg/> [Accessed 5 January 2022].

[2]Math.cuhk.edu.hk, 2022. [Online]. Available: https://www.math.cuhk.edu.hk/~lmlui/dct.pdf. [Accessed: 06- Jan- 2022].

[3]"Huffman Code | Brilliant Math & Science Wiki", Brilliant.org, 2022. [Online]. Available: https://brilliant.org/wiki/huffman-encoding/. [Accessed: 06- Jan- 2022].

[4]R. A.M, K. W.M, E. M. A and W. Ahmed, "Jpeg Image Compression Using Discrete Cosine Transform - A Survey", International Journal of Computer Science & Engineering Survey, vol. 5, no. 2, pp. 39-47, 2014. Available: 10.5121/ijcses.2014.5204.