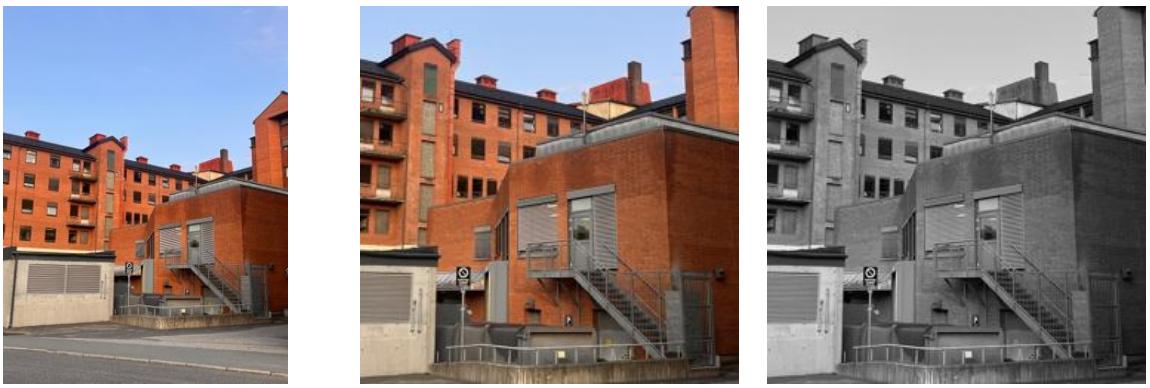
Introduction to Color Image Processing

Author: Seyedeh Kimia Arfaie Oghani

1. Week One

Figure 1-a, shows the original image that was taken, the image was then cropped to an image of size 2230x2230 pixels with the Preview software, which is a photo viewer on MacOS system. Image then was resized to 256x256 with the help of MATLAB function imresize, which is shown in Figure 1-b. Then, with the help of MATLAB function im2gray the 256x256 gray scale image was obtained (Figure 1-c).



a b c

Figure 1 – a. Original Image, b. 256x256 RGB image, c. 256x256 gray scale image.

2. Week Two

Figure 2 shows the histogram of the 256x256 gray scale image.

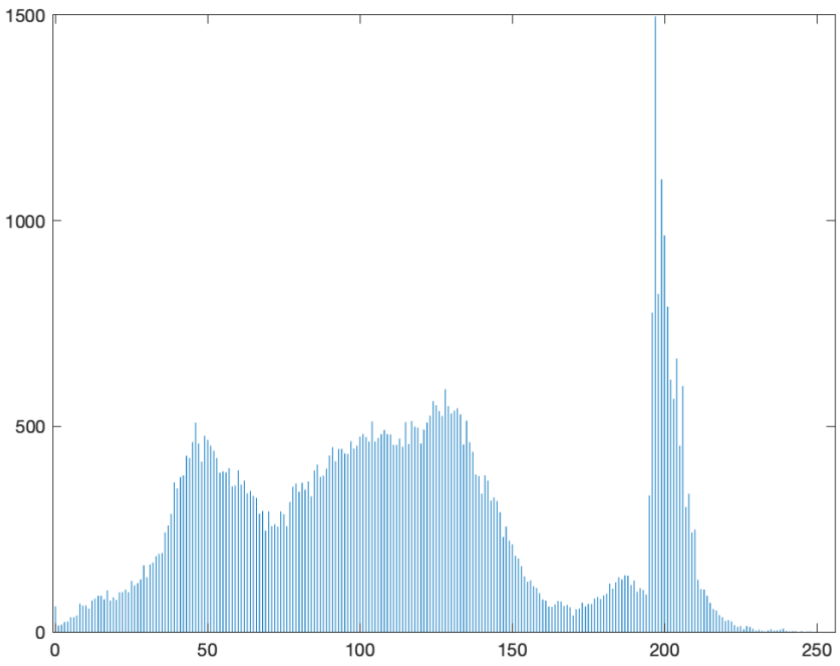
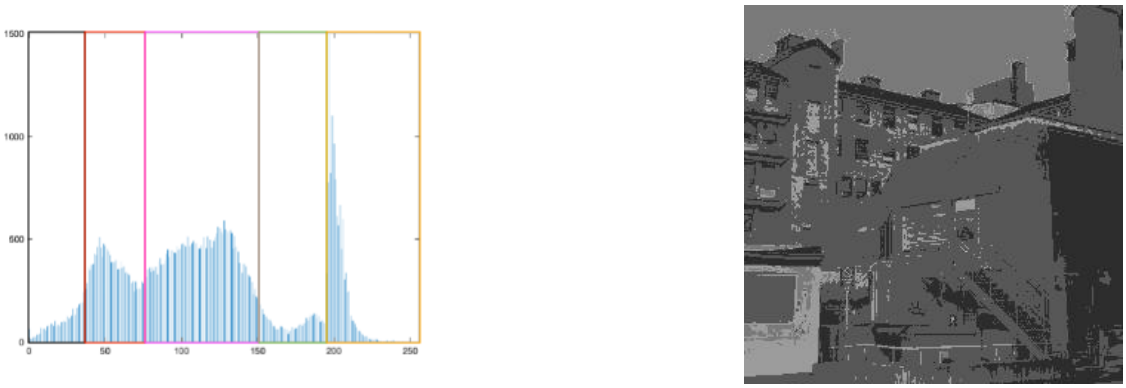


Figure 2 – Histogram of 256x256 gray scale image.

Next, I performed image segmentation, by dividing the histogram into 5 different sections, with corresponding regions of [0-30), [30-70), [70,150), [150-195), [195-255], and replacing the intensities of the regions with their corresponding average of intensities. See Figure 3.



Approximation of segments Segmented image

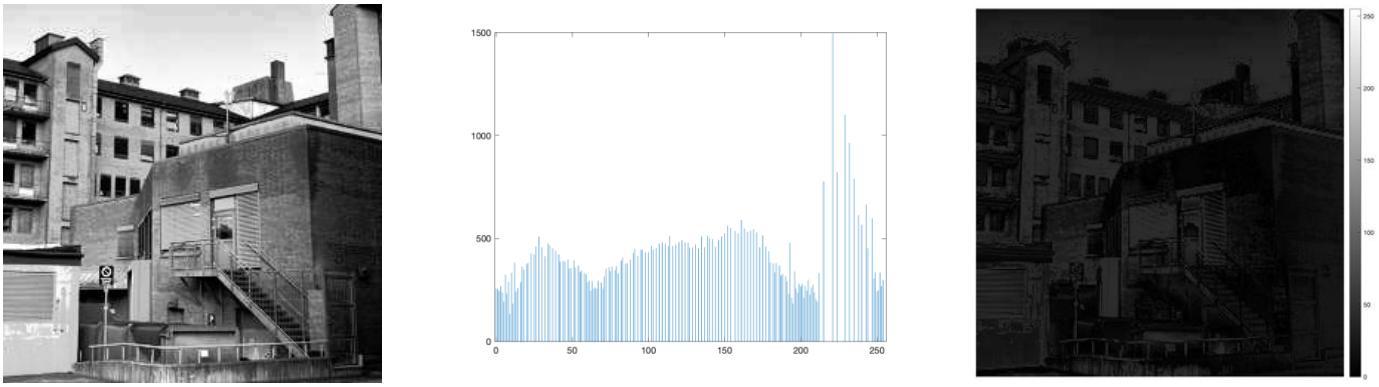
Figure 3 – An approximation of the segmentation and the segmented Image.

Next, image normalization was done. Figure 4. Shows the original image next to normalized image and their difference. Since normalization only changes the range of pixel intensity values, the difference of two images is so small that the difference image is black, since the values of the original image is already between 0-255.

a b c

Figure 4 – a. Original Image, b. Normalized image, c. Difference image.

Figure 5 shows the image obtained after histogram equalization and its histogram, next to the difference image to the original gray scale image.



a b c

Figure 5 – a. Image after histogram equalization, b. Histogram of a, c. Difference image.

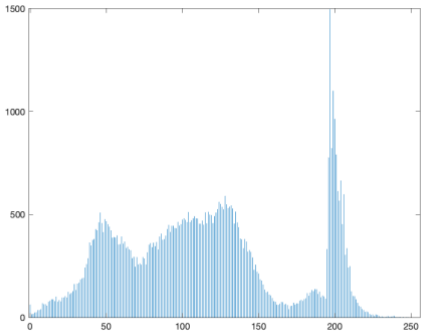
3. Week Three

In this section, I apply histogram matching to the image, and match 256x256 grey scale to the target image. Figure 6 shows the 256x256 grey scale and its histogram, original image and its histogram and the output of histogram matching.

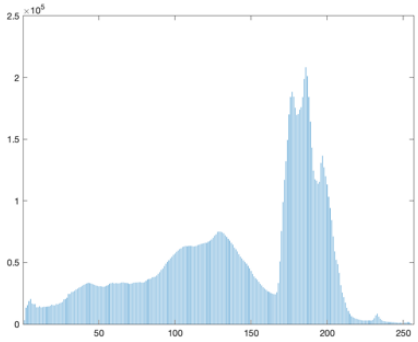
A: 256x256 image

B: Original Gray scale image

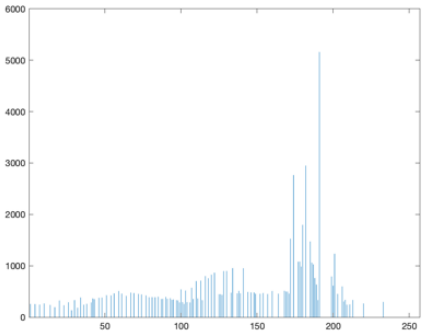
A matched to B



Histogram of A



Histogram of B



Matched Histogram

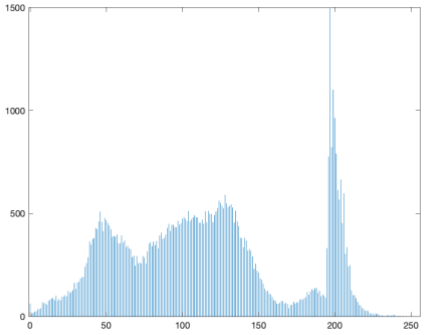
Figure 6 – 256x256 image, original image, and output of histogram matching with their histograms.

Figure 7, shows the same process but this time, A is matched to a random histogram.

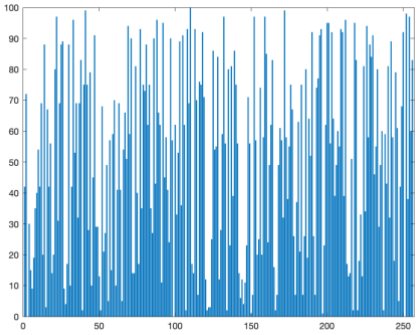
A: 256x256 image

B: Random histogram

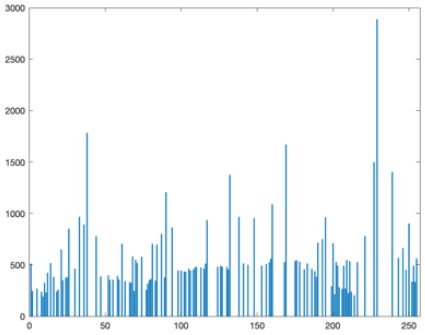
A matched to B



Histogram of A



Random histogram



Matched Histogram

Figure 7 – 256x256 image matched to a random histogram.

4. Week Four

In each one of the distortions, Average, Weighted Average, Local Median and Laplacian filters were tested to find the best one suitable for removing the distortion. Difference image of each filter was obtained, and through comparing them, the best filter was chosen.
The 3x3 matrices that were tested are as followed:
Average filter: 1/9*[1 1 1; 1 1 1; 1 1 1];
Weighted Average filter = 1/16*[1 2 1; 2 4 2; 1 2 1];
For Local median, in each block the median of the pixels were chosen;
Laplacian kernels of: [0 -1 0; -1 4 -1; 0 -1 0] and [-1 -1 -1; -1 8 -1; -1 -1 -1].

A. Salt and Pepper noise

In salt and pepper noise, local median filter noticeably had the best results. Figure 8 shows the distorted, filtered and difference images.

a b c

Figure 8 – a. Distorted Image with salt & pepper noise, b. Filtered image, c. Difference image.

B. Speckle noise

In speckle noise, none of the filters showed good results, however, weighted mean had better results compared to other filter, but it can be seen that it has high blurring effect on the output image. Figure 9 shows the distorted, filtered and difference images.

a b c

Figure 9 – a. Distorted Image with speckle noise, b. Filtered image, c. Difference image.

C. Blurring

In blurring, overall, Laplacian filter had the best performance, in which among them [0 -1 0; -1 4 -1; 0 -1 0] had the best result.
Figure 10 shows the distorted, filtered and difference images.

a b c

Figure 10 – a. Distorted Image with blurring, b. Unblurred image, c. Difference image.

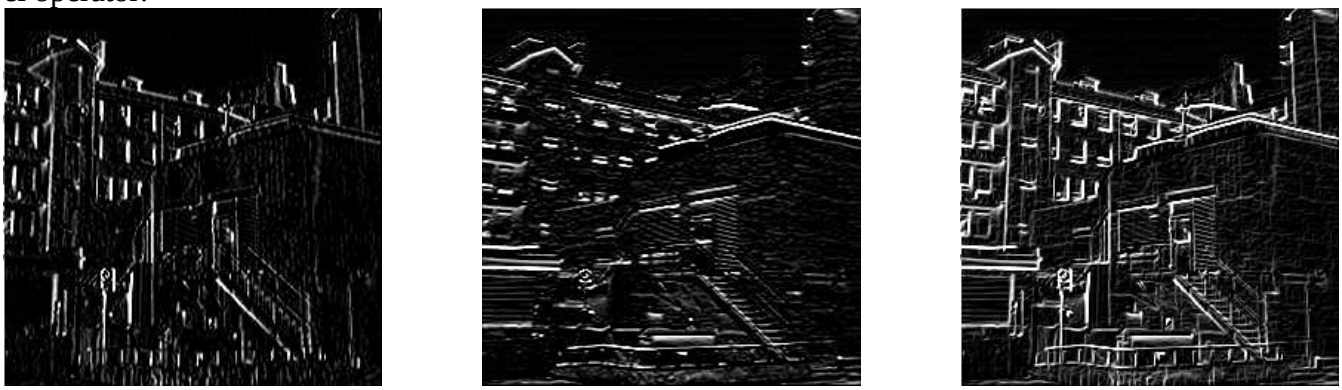
It has to be mentioned that in order to remove the blurring as best as we can with Laplacian filter, applying the filter alone isn't enough. After applying the Laplacian filter to a blurred image, the output was a black and white image that in which the outlines/edges of the image are evident. So, in order to unblur the image, we have to add/subtract the obtained black and white image (Laplacian image) to/by the blurred image, so the background features of the image can be recovered. The formula is as follows (Gonzalez, 2009):

Unblurred Image = Blurred Image + c*Laplacian Image

In which c, is a constant, that is -1 if the Laplacian filter used is [0 +1 0; +1 -4 +1; 0 +1 0], and it is +1 if the filter is [0 +1 0; +1 -4 +1; 0 +1 0]. The same also applies for other Laplacian filters. It has to be mentioned that for other sections, this formula was also used.

D. Image Gradient

In figure 10, image gradient in X and Y direction and its magnitude can be seen which were calculated using Sobel operator.



a b c

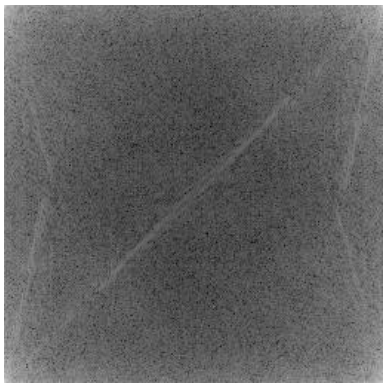
Figure 11 – a. Image gradient in X direction, b Image gradient in Y direction, c. Magnitude.

5. Week Six

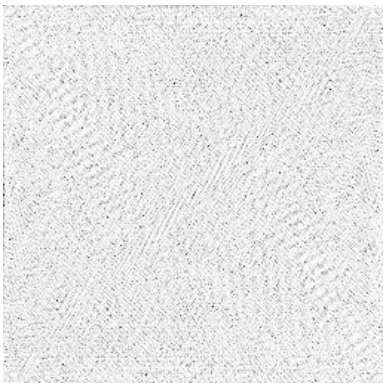
In this section Fourier transform of the image was obtained. In figure 12, you can see the DFT magnitude and phase in logarithmic view, before and after applying zero shift frequency.

Original Image

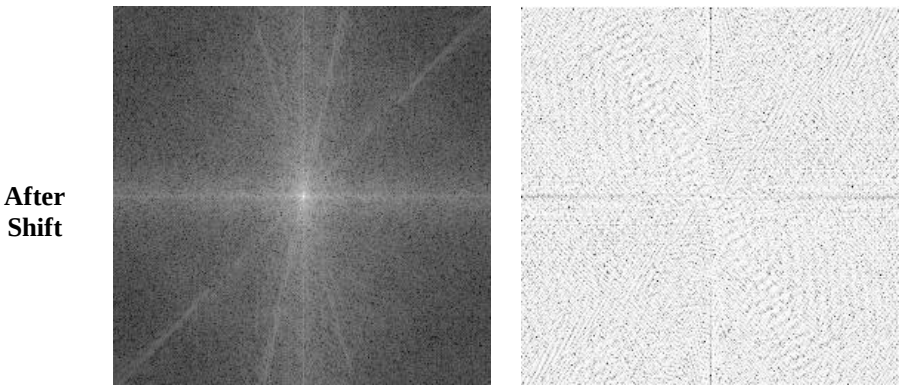
Before shift



DFT magnitude in log10



DFT phase in log10



DFT magnitude in log10 DFT phase in log10

Figure 12 – DFT magnitude and phase in logarithmic view, before and after applying zero shift frequency.

Next, inverse Fourier transform was gradually applied by using sections (from central 2x2 pixels to all the pixels of Fourier transform) of Fourier transform to reconstruct the image. Figure 13 shows an example of using 40x40 central pixels of Fourier transform to reconstruct the image.

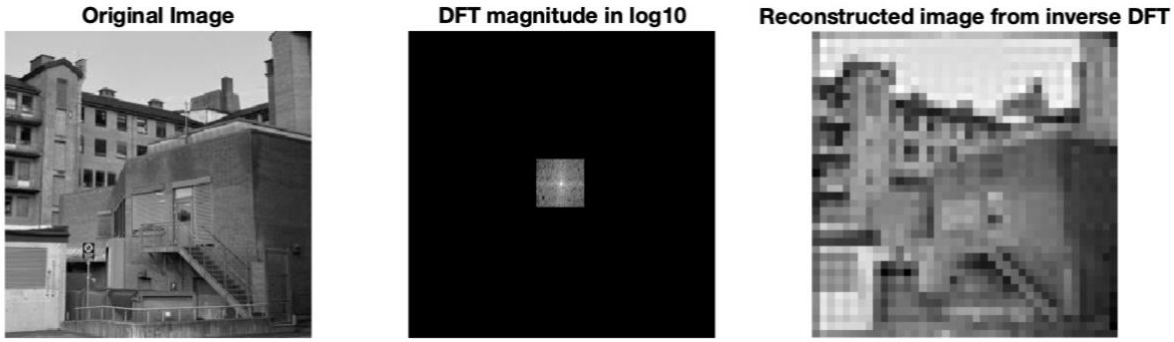
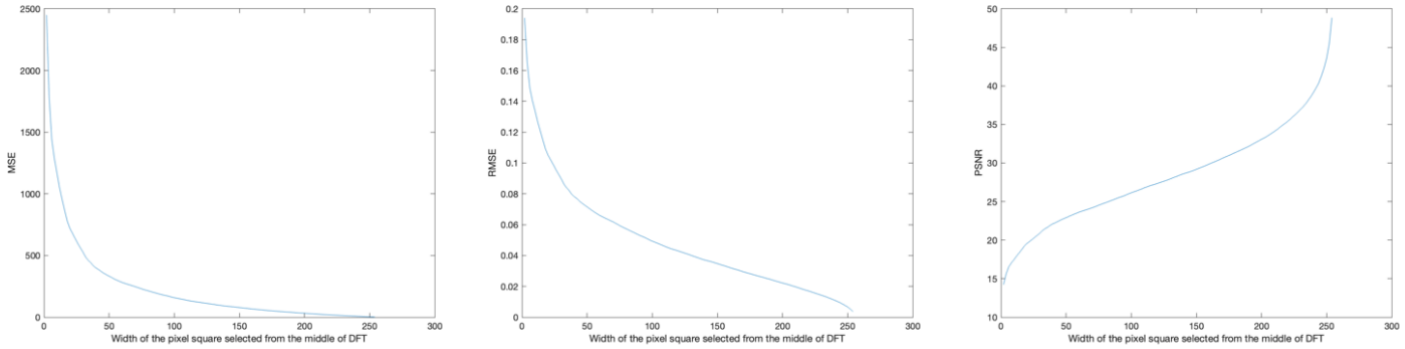


Figure 13 – Reconstructing the image, using 40x40 central pixels of Fourier Transform.

Figure 14 shows the MSE, RMSE and PSNR values for reconstructing the image against the width of the square selected from the center of the Fourier Transform.



a b c

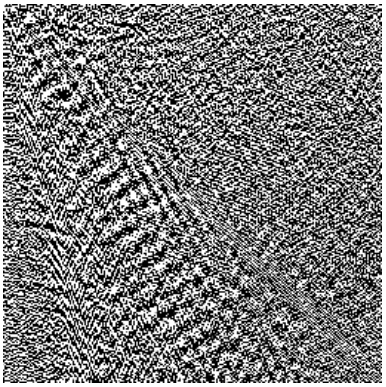
Figure 14 – a. MSE, b. RMSE, c. PSNR plots against the width of the square selected from the center of the Fourier Transform.

6. Week Seven

A. Discrete Cosine Transform

Figure 15, shows the original image and DCT of the image.

Original Image



DCT

Figure 15 – Original image and its DCT.

Next, inverse Cosine transform was gradually applied by using sections (by selecting specific pixels of rows and columns of CDT) of DCT to reconstruct the image. Figure 16 shows an example of using 25 rows and columns of pixels of DCT to reconstruct the image.

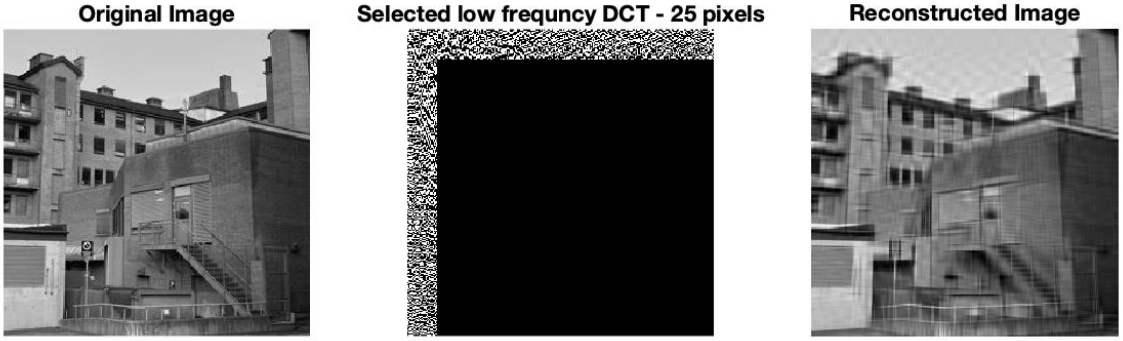
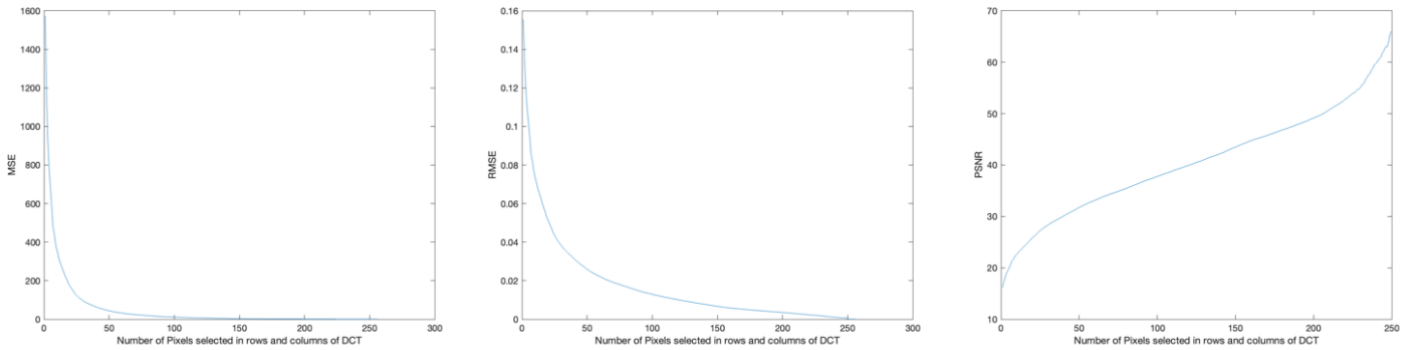


Figure 16 – Reconstructing the image, using 25 row and column pixels of DCT.

Figure 17 shows the MSE, RMSE and PSNR values for reconstructing the image against the number of pixels selected in rows and columns of the DCT.



a b c

Figure 17 – a. MSE, b. RMSE, c. PSNR plots against the number of pixels selected in rows and columns of the DCT.

B. Walsh – Hadamard Transform

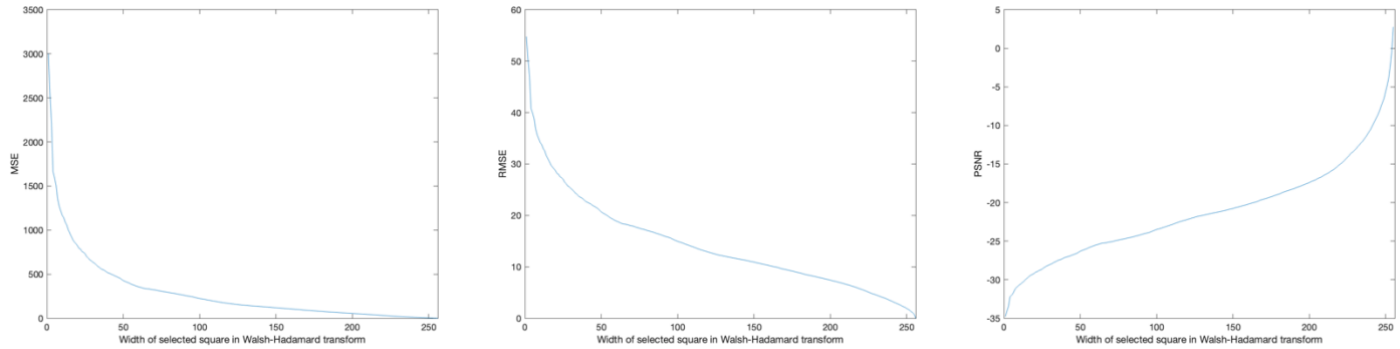
Figure 18, shows the original image and Walsh-Hadamard Transform. In order to obtain the Walsh-Hadamard transform the inbuilt function of MATLAB was used which is a column wise Walsh-Hadamard Transform, so with using the inverse of column wise Walsh-Hadamard, the Walsh-Hadamard transform can be calculated.

Original Image

Walsh-Hadamard Transform in log10

Figure 18 – Original image and its WHT in log10.

Next, inverse Walsh-Hadamard transform was gradually applied by using sections (by selecting thee left upper corner square from WHT, from one pixel until all of the transform is applied) of WHT to reconstruct the image. Figure 19 shows the MSE, RMSE and PSNR values for reconstructing the image against the width of selected square of pixels from the left upper corner of WHT.

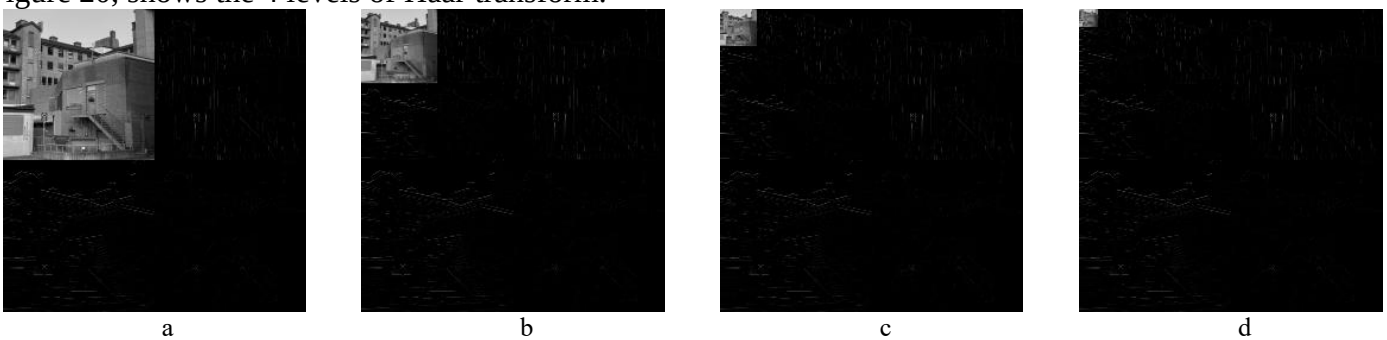


a b c

Figure 19 – a. MSE, b. RMSE, c. PSNR plots against the width of selected square of pixels from the left upper corner of WHT.

C. Haar transform

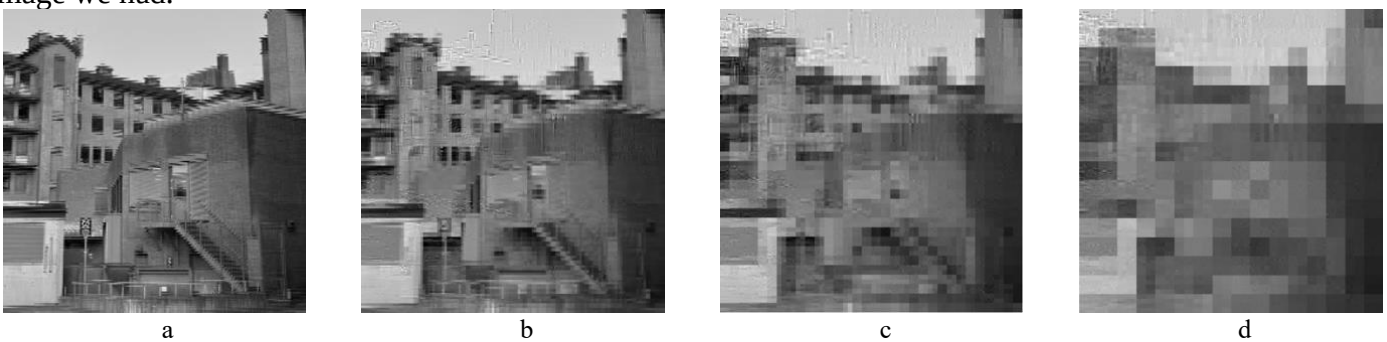
Figure 20, shows the 4 levels of Haar transform.



a b c d

Figure 20 - The Four levels of Haar transform from a being the first level to d being the fourth level.

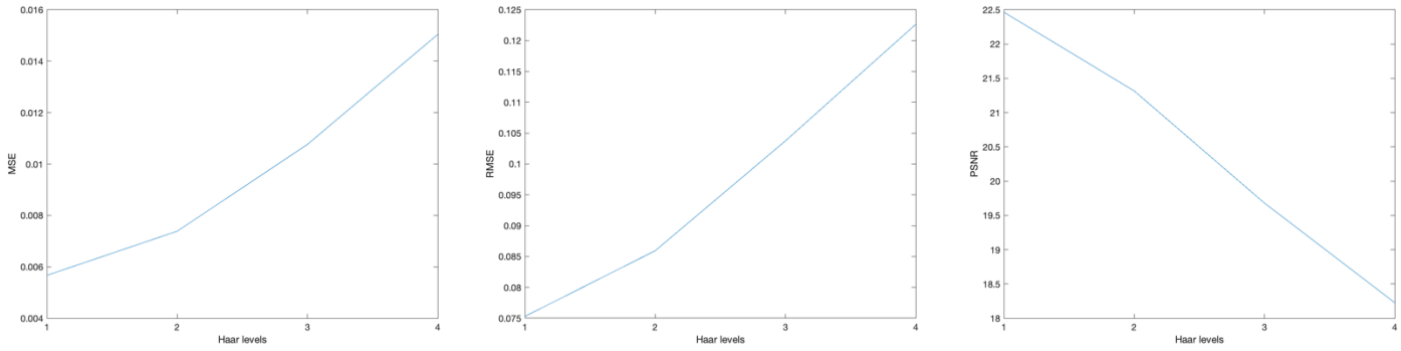
Figure 21, shows the reconstructed images using only the low frequencies, from the four levels of Haar transforms, therefore it is a lossy compression and as it can be seen even in level one, we don't get the exact image we had.



a b c d

Figure 21 – Reconstructed Images from inverse Haar transform for a being the level 1 to d being the level 4.

Figure 22 shows the MSE, RMSE and PSNR values for reconstructing the image against the levels of Haar transform.



a b c

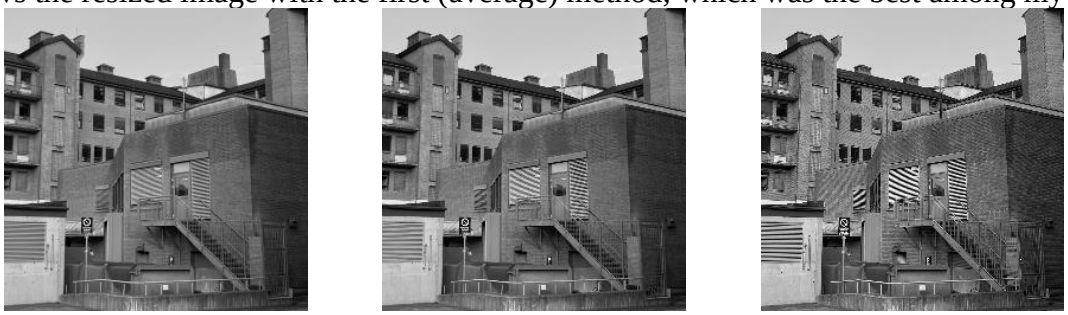
Figure 22 – a. MSE, b. RMSE, c. PSNR plots against the levels of Haar transform.

7. Week Eight

In this section I tried to resize the image from 1024x1024 to 256x256, with three different approaches.

1. By looping through the pixels in the 1024x1024 image, calculating the average of each 4x4 block of the image, and putting the output into the new image which will be 256x256. This method had the SSIM of 0.90, which was the best one among the others.
2. The same method as one, but this time by getting the median of each 4x4 block. This method had the SSIM of 0.86.
3. Going through the matrix of image 1024x0124 and only choosing the pixels with odd indexes, and putting them in a new image, which will give us a 512x512 image. If we do this process again, we will have a 256x256 image. This method had the SSIM of 0.59.

Figure 23 shows the resized image with the first (average) method, which was the best among my methods.



Resized with average method SSIM = 0.90

Resized with median method SSIM = 0.86

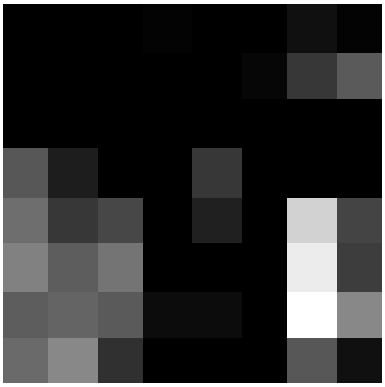
Resized with selecting odd index pixels SSIM = 0.59

Figure 23 – Resized image using different methods and the SSIM comparing these images with the gray scale 256x256 image I used.

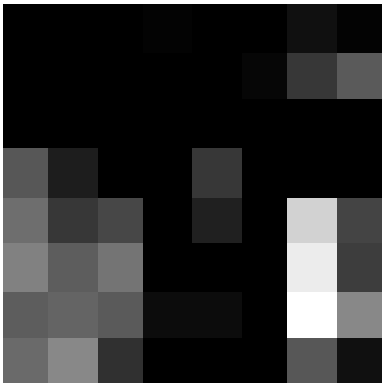
8. Week Nine

A. JPEG Compression

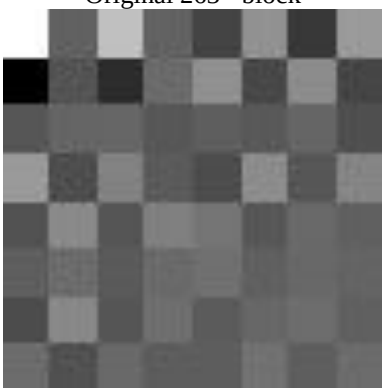
In this part, the image is first divided into blocks of 8x8 size, then I computed DCT for each block, performed quantization, scanned each block and scored the differential coding for each block. Image 24 shows the 263rd original block, before and after subtracting 128, after DCT calculation, and after applying quantization.



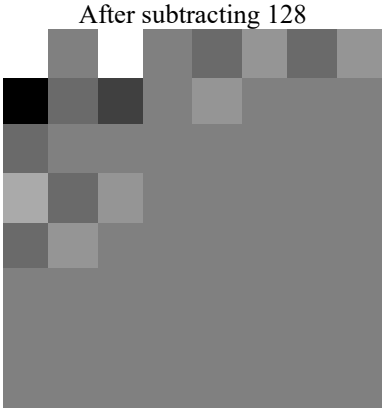
Original 263rd block



After subtracting 128



After DCT calculation

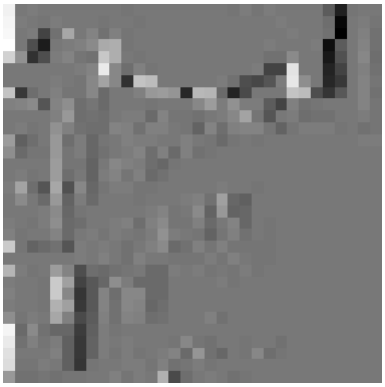


After applying quantization

Figure 24 – 263rd block, before and after subtracting 128, after DCT calculation, and after applying quantization

The coding for this particular block is as follows:
[(0, 3), 6], [(1, 3), -6], [(0, 1), -1], [(0, 1), -1], [(0, 3), 6], [(1, 2), -3], [(1, 2), 2], [(0, 1), -1], [(0, 1), -1], [(2, 1), -1], [(0, 1), 1], [(0, 1), 1], [(1, 1), 1], [(0, 1), 1], [(7, 1), -1], [(0, 1), 1], [(0,0)].
In order to calculate the DPCM, DC coefficients, which are the first element of quantized values were obtained, then DCPM was calculated by subtracting the neighboring DC coefficients except the first one. Image 25 is a visualization of DC components and DCPM that were calculated which were a 32x32 matrix. The DCPM value for the 263rd block was calculated as -3.

DC components



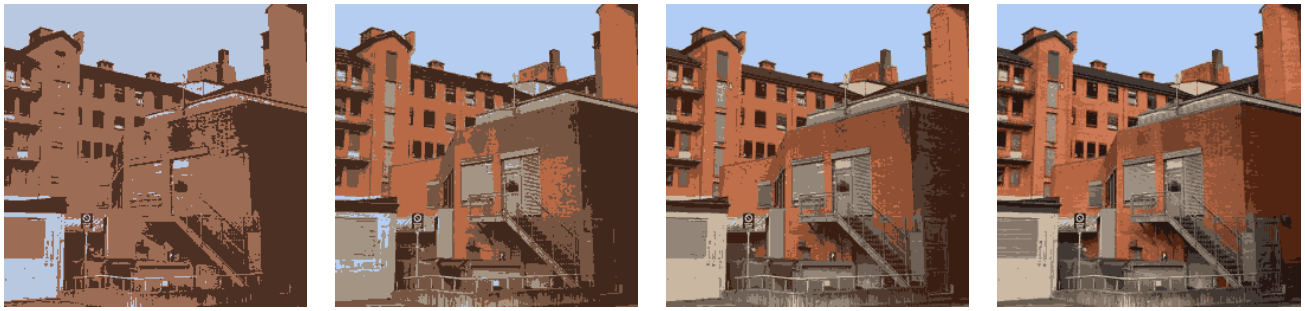
DCPM

Figure 25 – DC components and DCPM of the image.

B. Color Quantization

1. K-means Clustering

This method clusters different colors into k different clusters, which here, k would be the number of colors we want our output image to have. In this part, the built-in function of MATLAB was used, in which it starts by selecting centers of k clusters, calculates the Euclidean distance of colors to the center of the cluster and assigns colors to different clusters based on that, the process repeats and the clusters keep updating, until the clusters won't change significantly. Figure 26 shows different quantized images with different numbers of quantization.

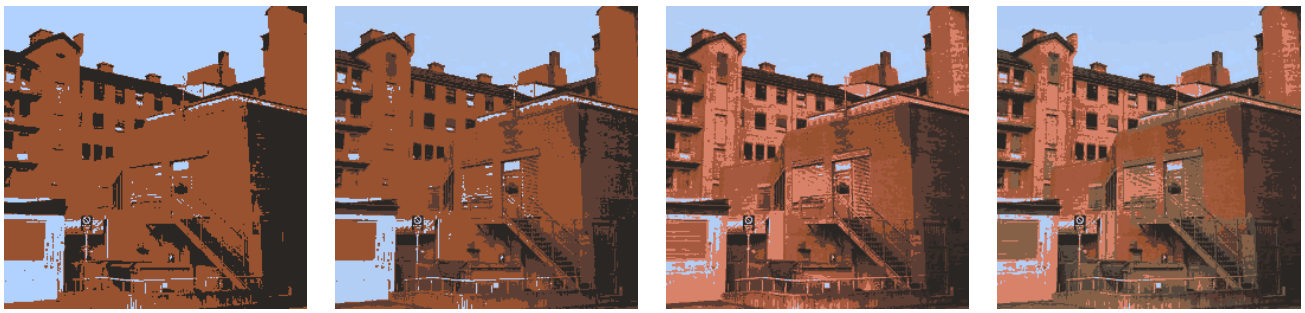


k = 3 k = 5 K = 8 K = 12

Figure 26 – Color quantized images with K-means method, at different levels of quantization.

2. K-nearest neighbor

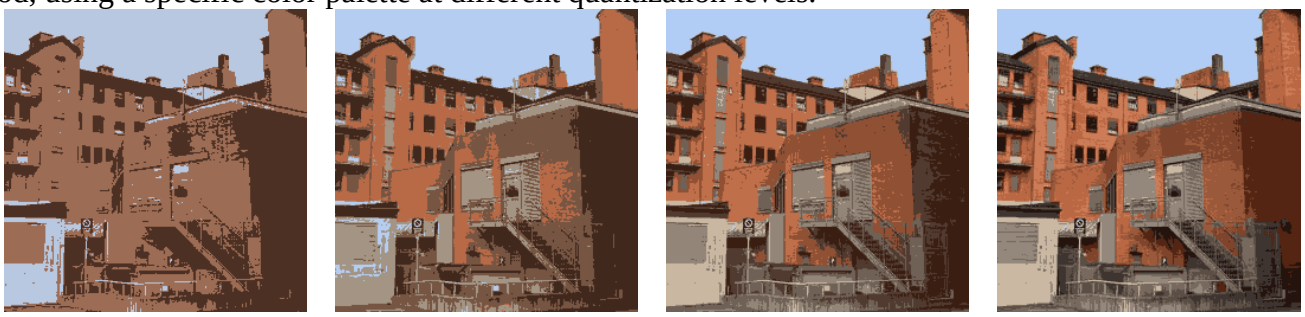
K-nearest neighbor is primarily a classification algorithm, that can be used to treat each unique color in the image as a "class", and then reduce the number of them. At first, k random colors were chosen from the image as the palette for quantization, and then the built-in function in MATLAB was used to find the nearest neighbors of the colors that were randomly chosen. Using random colors didn't have satisfactory results, since they are an important part of the color quantization, as it can be seen in Figure 27.



k = 3 k = 5 K = 8 K = 12

Figure 27 – Color quantized images with K-nearest neighbors method, at different levels of quantization, with a random color palette as the reference of quantization.

So, in order to have a more suitable palette, k-means method was used to extract the palette from the original image and then it was used for the k-nearest neighbor. Figure 28 shows the result of color quantization with this method, using a specific color palette at different quantization levels.

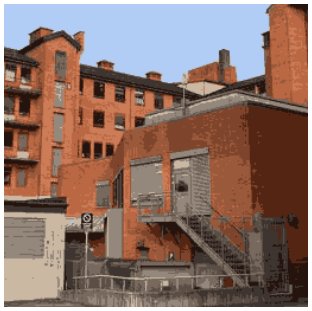


k = 3 k = 5 k = 8 k = 12

Figure 28 – Color quantized images with K-nearest neighbors method, at different levels of quantization, with a color palette obtained from k-means method, as the reference of quantization.

As it can be seen, the color quantization is much better, and now the results look similar to k-means method (see Figure26). But some slight differences can be seen in higher values for k such as 12 and 16; for instance, Figure 29, shows the color quantized image from k-mean and k-nearest neighbor method, and their difference for k=12.

K-mean method
k = 12



K-nearest neighbor method (palette from k-mean method)
k = 12

Difference Image

Figure 29 – Color quantized images from k-mean and k-nearest neighbor method, and their difference for k=12.