VRIJE UNIVERSITEIT BRUSSEL

IMAGE & VIDEO TECHNOLOGY

LAB REPORT

FARHA PARVEEN PADIYATHU PUTHENKATTIL FAZAL
STUDENT ID : 0566305
2021-2022

Introduction

This report summarizes the results of the lab exercises performed for the course on Image & video Technology. As illustrated in the *ivt_exercises_2020.pdf* file, the lab exercises have been divided into the following 6 sessions and their main objectives are summarized below.

- **Session 1** : Basic analysis and operations on images in C++ & ImageJ
- **Session 2** : Noise addition & blur on Images
- **Session 3** : Discrete Cosine Transforms
- **Session 4** : Basic encoding/decoding of images with quantized DCT coefficients.
- **Session 5** : Advanced encoding/decoding of the quantized DCT coefficients with Delta Encoding (for DC coefficients) and Run Length Encoding (for AC Coefficients).
- **Session 6 :** Compression of images by combining binary Golomb codes with the encoding methods in Session 5.

The C++ code for each of these sessions are provided independently in separate folders. Each of these sessions have been divided into different tasks. There are 14 tasks overall, and each task has been divided into sub-tasks. I will describe them sequentially, in the same format provided in the *ivt_exercises_2020.pdf* file.

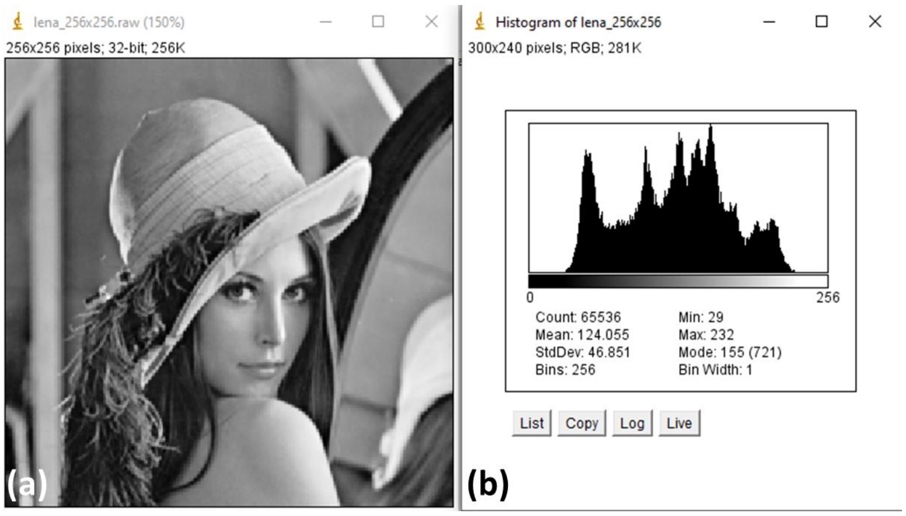
Session 1 : Basic analysis and operations on images in C++ & Image J

Task 1 : Getting Started

1.1 To start, I have been able to compile and run C++ codes (including a code that prints "Hello World" on the output console). For this, I had to install the C++ compilers and Code Runner extensions in *Visual Studio Code*.

1.2 The Lena RAW image was then imported into the ImageJ viewer. The characteristics of the Raw image including the pixel number (256×256), number of bits per pixel (32), have to specified while importing the image. Furthermore, the image type was set as *32 bit-real* and *little-endian byte order* was selected to get the correct image (Fig. 1a).

1.3 The characteristics of the lena image was then analysed in ImageJ based on the histogram (Fig. 1b). The minimum and maximum axis of the histogram was set at 0 and 256 respectively, and 256 bins were considered.


lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
Histogram of lena_256x256
300x240 pixels; RGB; 281K
0
256
List
Copy
Log
Live
(a)
(b)

Count: 65536
Mean: 124.055
StdDev: 46.851
Bins: 256
Min: 29
Max: 232
Mode: 155 (721)
Bin Width: 1

Fig. 1 (a) The Lena image imported in ImageJ (b) Histogram of the Lena image.

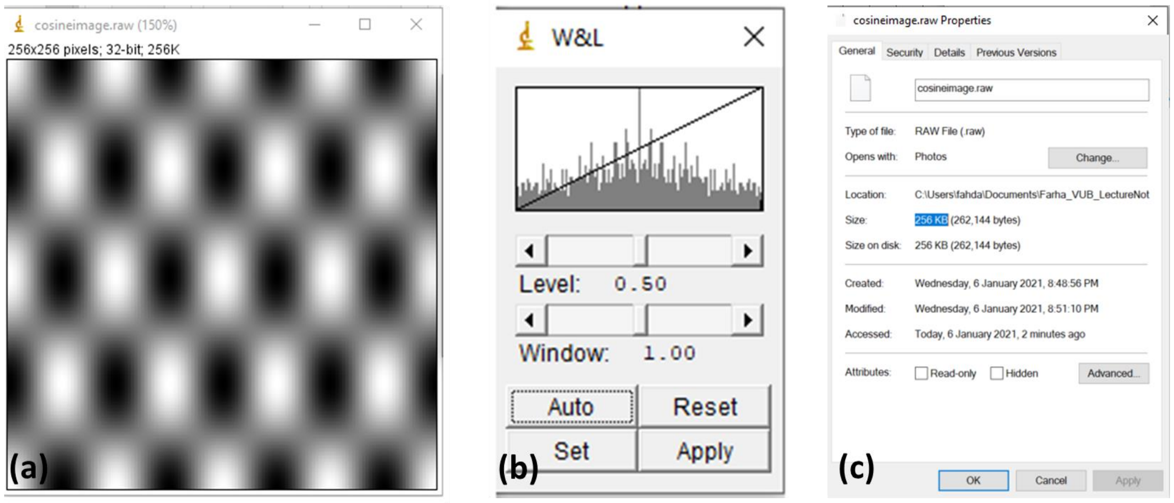
Fig. 1b plots the distribution of intensities of the 65536 (256*256) pixels in the Lena image. The distribution indicates the mean intensity at 124.055 with standard deviation equal to 26.851. The minimum and maximum intensity values in the image were 29 and 232 respectively. The mode of the intensity was 155, and there were 721 pixels in the image that had this intensity value.

Task 2 : Create and Store a RAW 32 bpp grayscale image

2.1 An image (Fig. 2a) having spatially-dependent intensities in the range [0,1] was created using the following formula.

$$I(x,y) = \frac{1}{2} + \frac{1}{2}\cos\left(\frac{\pi x}{32}\right)\cos\left(\frac{\pi y}{64}\right), \qquad \text{Eq. 1}$$

where $I(x,y)$ corresponds to the intensity of pixel at (x,y).



256x256 pixels; 32-bit; 256K

Level: 0.50
Window: 1.00

cosineimage.raw Properties
General | Security | Details | Previous Versions

cosineimage.raw
Type of file: RAW File (.raw)
Opens with: Photos
Location: C:\Users\fahda\Documents\Farha_VUB_LectureNot
Size: 256 KB (262,144 bytes)
Size on disk: 256 KB (262,144 bytes)
Created: Wednesday, 6 January 2021, 8:48:56 PM
Modified: Wednesday, 6 January 2021, 8:51:10 PM
Accessed: Today, 6 January 2021, 2 minutes ago
Attributes: ☐ Read-only ☐ Hidden

Fig. 2 (a) An image with cosine pattern shown in Equation 1. (b) The optimal window and level for capturing the intensities well. (c)Size of the raw image when stored as raw

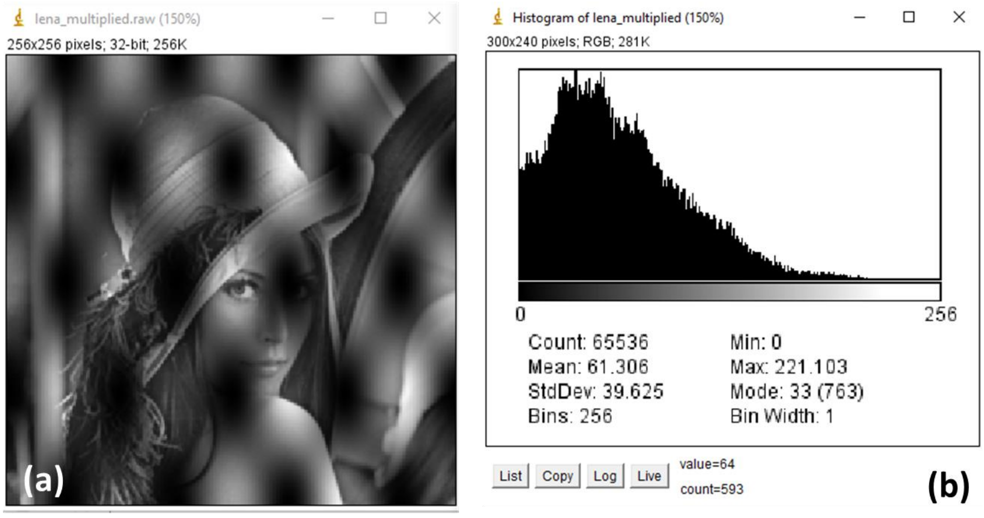
2.2 The above image is stored as a RAW file and visualized (Fig. 2a). Note that there are 256×256 pixel value and each pixel is stored as 4 bytes (for floating point representation). Hence the total size on the RAW file should be 256×256×4 = 262,144 bytes and is the same as the one shown in Fig. 2c.

2.3 I also varied the window and adjust levels to capture all the intensities well in the image. The optimal window and level are 1.0 and 0.5, as shown in Fig. 2b

Task 3 : Load and modify a RAW 32 bpp grayscale image

3.1-3.2 I wrote a function (with name *load_image*) that can take a raw file and load in memory (a 2D array with size 256×256) for further processing.

3.3 The loaded image was multiplied (pixel by pixel) with the cosine pattern in Task 2, stored as a RAW file and plotted in imageJ (Fig. 3a). Note that the histogram (Fig. 3b) skews towards the lower intensity values due to Eq. 1.


lena_multiplied.raw (150%)
(a)
Histogram of lena_multiplied (150%)
0
256
List
Copy
Log
Live
(b)

256x256 pixels; 32-bit; 256K

300x240 pixels; RGB; 281K

```
Count: 65536      Min: 0
Mean: 61.306      Max: 221.103
StdDev: 39.625    Mode: 33 (763)
Bins: 256         Bin Width: 1
```

value=64
count=593

Fig. 3 (a) The Lena image multiplied with the cosine pattern and (b) its histogram.

3.4 The mean square error (MSE) between the multiplied image and the original lena image was calculated using the following formula.

$$\mathrm{MSE} = \frac{\sum_{x=0}^{N1}\sum_{y=0}^{N2}|\mathrm{Image1}(x,y) - \mathrm{Image2}(x,y)|^2}{N}, \qquad \text{Eq. 2}$$

where N_1 = N_2 = 255 and N = 65536 in our case. We obtain an MSE of 5613.47, and is also printed in the code.

3.5 Based on the MSE, we can also calculate the Power Signal Noise Ratio (PSNR) using the following formula

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{\mathrm{MAX}^2}{\mathrm{MSE}}\right), \qquad \text{Eq. 3}$$

where MAX = 255 is the maximum intensity value possible. We obtain a PSNR of 10.64 dB in this scenario.

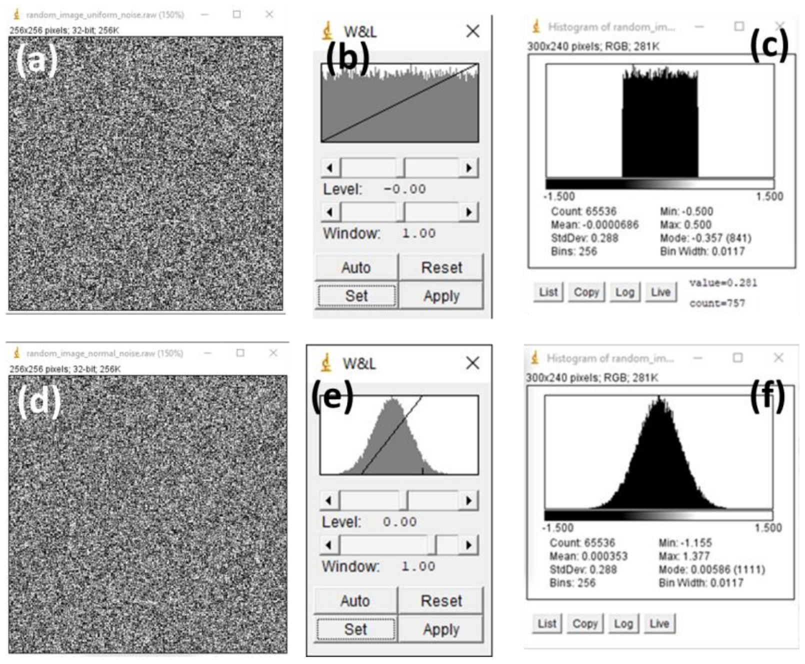
Session 2 : Noise and Blur on Images

Task 4 : Uniform and Gaussian Random White Noise

4.1 A 256×256 image with uniformly distributed random values in the range [-0.5,0.,5] was generated. I obtained the MSE of the random image (with respect to the zero-mean) to be 0.083, and is printed in the code. Note that variance (i.e MSE) of a uniform distribution in the range $[a, b]$ should be $\frac{(b-a)^2}{12}$. Substituting $a = -0.5$ and b = 0.5, the analytical value of MSE should be 0.083, and is the one obtained for the generated image.

4.2 Another 256×256 image with gaussian distributed random numbers with mean 0 was also generated. The distribution should have a variance of 0.083 in order to match the MSE of the random image.

4.3 We now compare the uniform and Gaussian distributed noise pattern in ImageJ (Fig. 4a vs Fig. 4c). For the comparison of the images, we assume a window level of 0 and width of 1, as shown in Fig. 4b and Fig. 4d.


(a)
random_image_uniform_noise.raw (150%)
256x256 pixels; 32-bit; 256K
(b)
W&L
Auto
Reset
Set
Apply
(c)
Histogram of random_im...
300x240 pixels; RGB; 281K
List
Copy
Log
Live
(d)
random_image_normal_noise.raw (150%)
256x256 pixels; 32-bit; 256K
(e)
W&L
Auto
Reset
Set
Apply
(f)
Histogram of random_im...
300x240 pixels; RGB; 281K
List
Copy
Log
Live

(b) Level: -0.00 Window: 1.00

(c)
```
-1.500                              1.500
Count 65536            Min: -0.500
Mean: -0.0000686       Max: 0.500
StdDev: 0.288          Mode: -0.357 (841)
Bins: 256              Bin Width: 0.0117
value=0.281
count=757
```

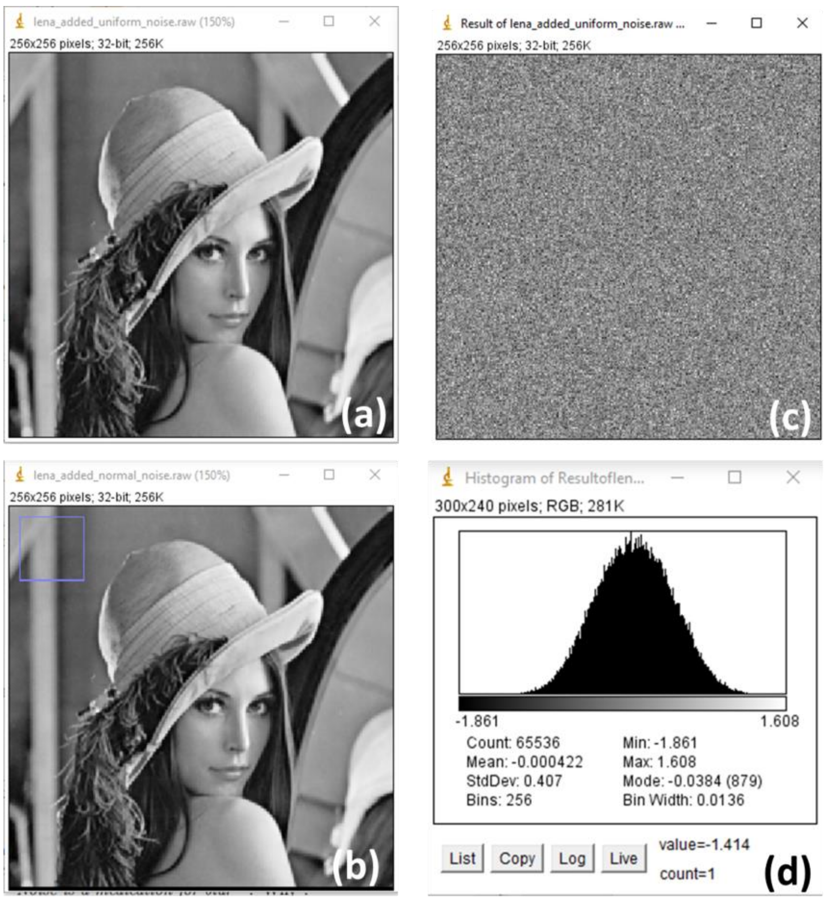
(e) Level: 0.00 Window: 1.00

(f)
```
-1.500                              1.500
Count 65536            Min: -1.155
Mean: 0.000353         Max: 1.377
StdDev: 0.288          Mode: 0.00586 (1111)
Bins: 256              Bin Width: 0.0117
```

Fig. 4 (a) Image with uniformly distributed noise between -0.5 and 0.5. (b) Setting level and window of uniform noise image at 0 and 1 respectively. (c) Histogram of image with uniformly distributed noise. (d) Image with Gaussian distributed noise having variance the same as the MSE of that of uniform noise. . (e) Setting level and window of gaussian noise image at 0 and 1 respectively. (c) Histogram of image with gaussian distributed noise distributed noise.

We also plot the histogram of the two noise images in Fig. 4c and Fig. 4d. The distributions are as expected for a uniform and gaussian distribution. Note the standard deviation of both the images are the same, as they have been selected to have the same variance. The mean value of both the images are zero.

4.4 We now add the two noise images to the Lena image as shown in Fig. 5a and 5b. Note both these images will be different from the original lena image with an MSE of 0.083. To compare the effect of the two noises, we take the difference of the resulting images with the noise in Fig. 5c. The histogram of the difference is shown in Fig. 5d, and indicates distribution similar to that of the a normal distribution. AS the histogram has positive and negative values with similar amplitudes, we cannot conclude whether one noise is better than the other.


lena_added_uniform_noise.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
Result of lena_added_uniform_noise.raw ...
256x256 pixels; 32-bit; 256K
(c)
lena_added_normal_noise.raw (150%)
256x256 pixels; 32-bit; 256K
(b)
Histogram of Resultoflen...
300x240 pixels; RGB; 281K
-1.861
1.608
List
Copy
Log
Live
(d)

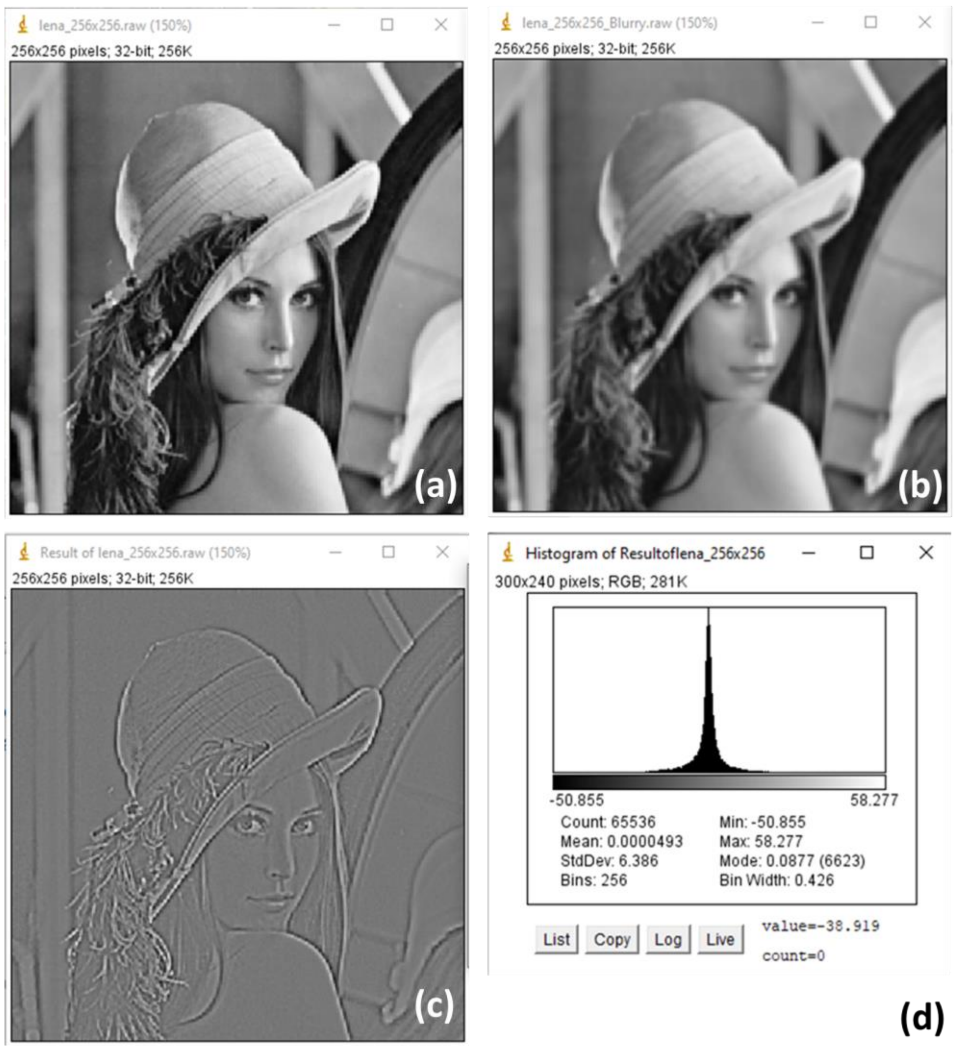
```
Count: 65536        Min: -1.861
Mean: -0.000422     Max: 1.608
StdDev: 0.407       Mode: -0.0384 (879)
Bins: 256           Bin Width: 0.0136
value=-1.414
count=1
```

Fig. 5 (a,b) Lena image superimposed with a (a) uniform distributed noise, and (b) Gaussian distributed noise. (c) The difference in the intensities for the two noisy images. (d) Histogram of the difference between the two noisy images.

Task 5: Blur and Additive White Noise using ImageJ

5.1 The Lena image is Blurred in Image J with a Gaussian Blur with standard deviation of 1. The resulting image is shown in Fig. 6b, and we observe that the image has been smoothened out (i.e. blurred) with a filter. The difference of the blurred image and the original lena image (Fig. 6a) is shown in Fig. 6c. The histogram of the difference is shown in Fig. 6d. In the histogram, we note that the standard deviation of the difference is 6.386. This would correspond to a <u>mean square error (MSE) of 6.386^2= 40.78.</u>

5.2 We now add a gaussian distributed noise to the blurred image. The resulting images for different variances (σ^2 between 1 and 225) of the gaussian noise are plotted in Fig. 7. Visually, I note that none of the images with gaussian noise makes the image better, when compared to the blurred image. This is expected as a blur is a filter of the image, and noise just gets added on top of that filtered image. Hence, we can conclude by highlighting that <u>adding noise is not a medication of blur.</u>


lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_Blurry.raw (150%)
256x256 pixels; 32-bit; 256K
(b)
Result of lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflena_256x256
300x240 pixels; RGB; 281K
-50.855
58.277
List
Copy
Log
Live
(d)

```
Count: 65536        Min: -50.855
Mean: 0.0000493     Max: 58.277
StdDev: 6.386       Mode: 0.0877 (6623)
Bins: 256           Bin Width: 0.426
```

```
value=-38.919
count=0
```

Fig. 6 (a) Original Lena image. (b) Image after applying a gaussian blur with standard deviation unity to the lena image. (c) The difference between the original lena image and blurred image. (d) Histogram of the differences indicating a standard deviation of 6.386.

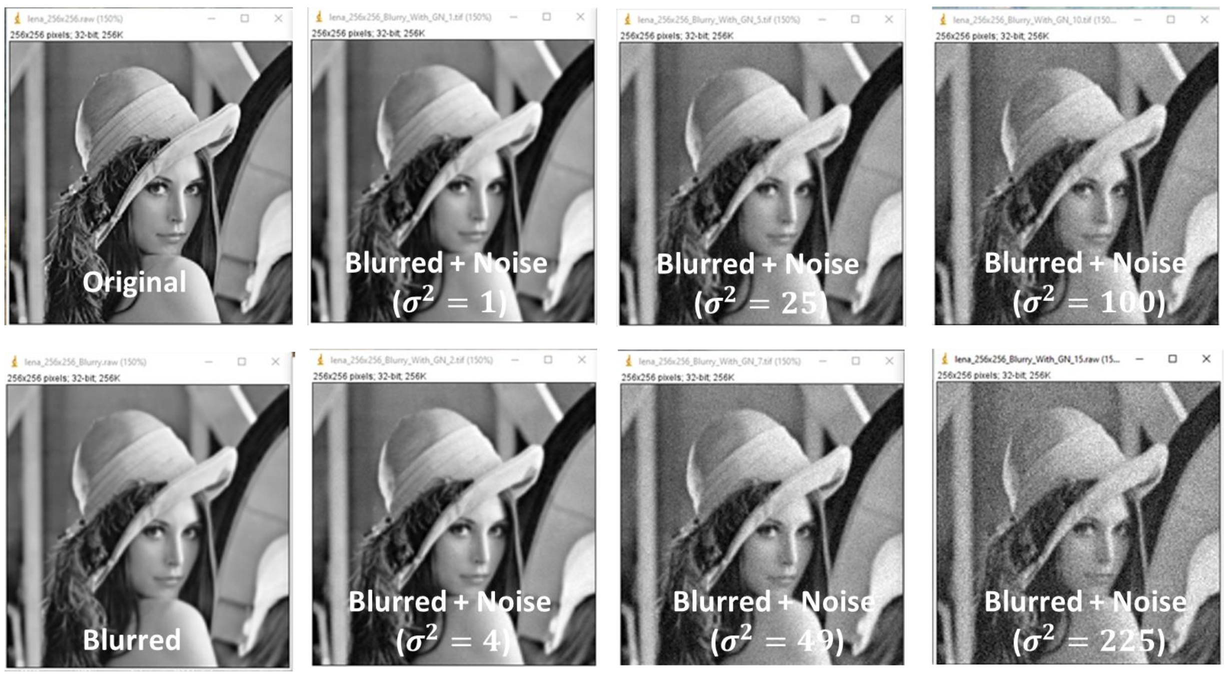
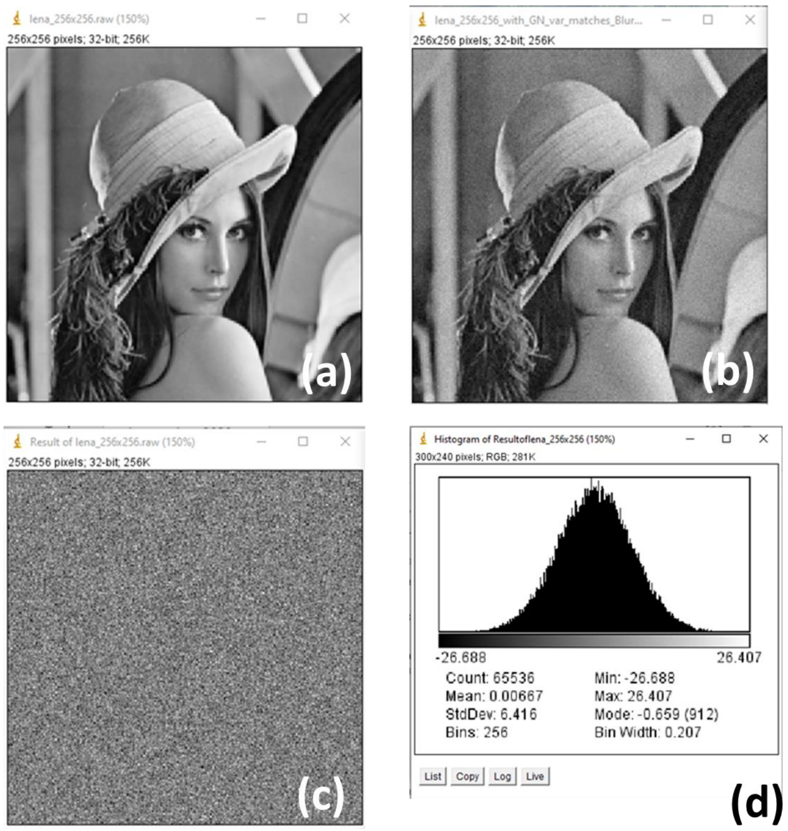

Original
Blurred + Noise ($\sigma^2 = 1$)
Blurred + Noise ($\sigma^2 = 25$)
Blurred + Noise ($\sigma^2 = 100$)
Blurred
Blurred + Noise ($\sigma^2 = 4$)
Blurred + Noise ($\sigma^2 = 49$)
Blurred + Noise ($\sigma^2 = 225$)

Fig. 7. Original Lena image, the image after applying a gaussian blur, and images after adding gaussian distributed noise to the blurred image. Different variances σ^2 of the noise are considered.

5.3 A gaussian distributed noise is added to the original lena image. As the MSE of the blurred image is 40.78, a gaussian noise with the same variance 40.78 has to be added to the lena image for matching the MSE. For this value of MSE, the resulting noisy image, the difference of the noisy image with respect to the lena image and the histogram of the difference is plotted in Fig. 8b, 8c and 8d respectively. We further verify that the MSE (41.16) of the noisy image is close to the targeted value of 40.78.



lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_with_GN_var_matches_Blur...
256x256 pixels; 32-bit; 256K
(b)
Result of lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflena_256x256 (150%)
300x240 pixels; RGB; 281K
-26.688
26.407
List
Copy
Log
Live
(d)

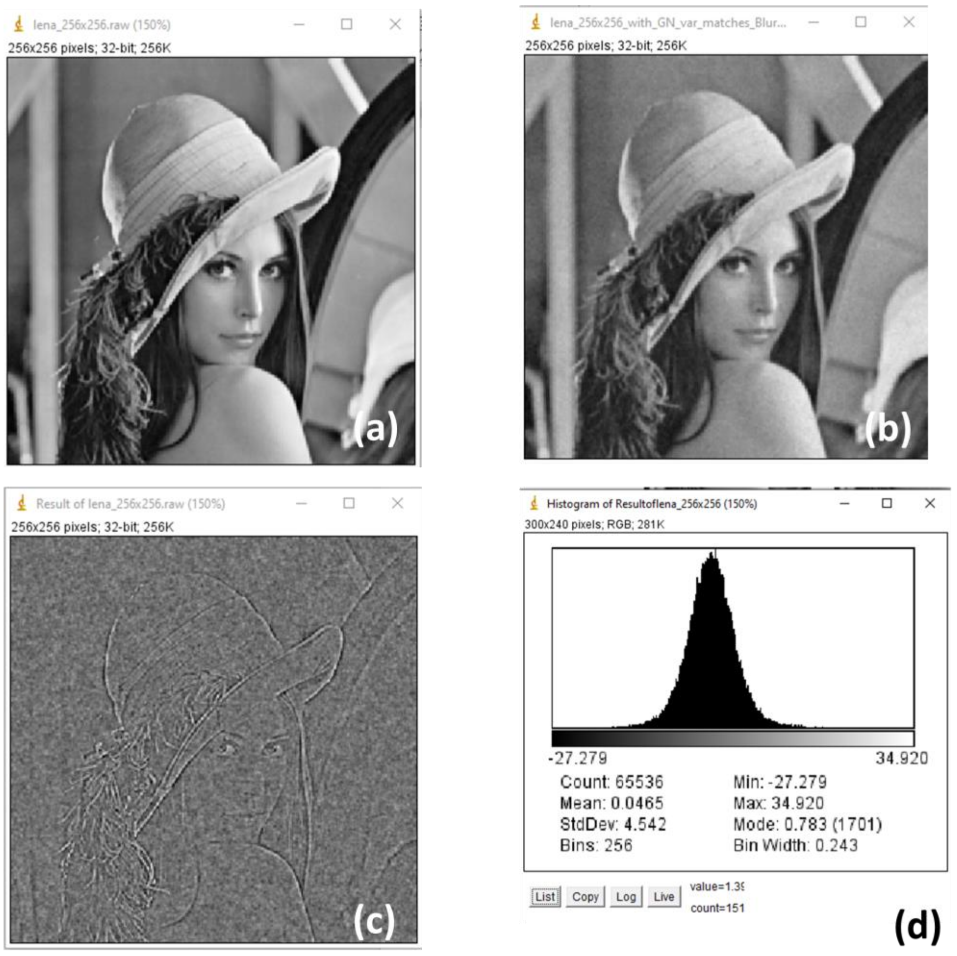
```
Count: 65536      Min: -26.688
Mean: 0.00667     Max: 26.407
StdDev: 6.416     Mode: -0.659 (912)
Bins: 256         Bin Width: 0.207
```

Fig. 8 (a) Original Lena image. (b) Image after applying gaussian noise with variance 40.78. (c) The difference between the original lena image and noisy image. (d) Histogram of the differences indicating an MSE of $6.416^2 = 41.16 \sim 40.78$ The slight discrepancy is due to the fact that noise is random.

5.4 We now apply a blur filter to the noisy image, and repeat for different standard deviations (σ_{BLUR}) of the Blur. The table in Fig. 9 indicates the MSE associated for different σ_{BLUR}.

Extremely small values of σ_{BLUR} just provides similar MSE. Increasing σ_{BLUR} initially lowers the value of MSE, indicating that noise is being filtered by the blur filter. As gaussian noise has all frequency components, the low-pass blur filter out many of the components of noise. Hence, we highlight that <u>Blur (i.e. a filter) is a medication of noise.</u>

As we increase σ_{BLUR}, major components of the image gets filtered out, and hence the MSE increases further again. From the table in Fig. 9, we infer that the optimal value of σ_{BLUR} is 0.6, resulting in an MSE of 20.63.


lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_with_GN_var_matches_Blur...
256x256 pixels; 32-bit; 256K
(b)
Result of lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflena_256x256 (150%)
300x240 pixels; RGB; 281K
-27.279
34.920
List
Copy
Log
Live
value=1.39
count=151
(d)

```
Count: 65536      Min: -27.279
Mean: 0.0465      Max: 34.920
StdDev: 4.542     Mode: 0.783 (1701)
Bins: 256         Bin Width: 0.243
```

σ_{BLUR}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.5	2.0	2.5	3.0
MSE	41.16	41.16	41.16	39.93	30.05	21.34	20.63	24.23	29.79	36.55	54.26	91.76	135.35	208.60	226.53

Fig. 9 (a) Original Lena image. (b) Image after applying gaussian noise and blurring with an optimal value of σ_{BLUR} = 0.5. (c)The difference between the blurred image and lena image. (d) Histogram of the difference indicating an MSE = 4.542^2 = 20.63. (Table) The MSE for different values of σ_{BLUR}.

Session 3 : Discrete Cosine Transforms

Task 6: Matrix of Orthogonal Basis Functions

6.1 The elements $d(i,j)$ of a matrix *D* containing all DCT basis vectors (in each row i) for 1D signal is given by :

$$d(i,j) = \lambda \cos\left(\frac{\pi j}{2N}\,(2i+1)\right), \qquad \text{Eq. 4}$$

, where N = 256 is the number of pixels in one dimension. λ is the normalization factor to ensure that all the basis vectors are normalized (i.e. sum of the squared elements in each row is unity). λ takes a value of $\frac{1}{\sqrt{N}}$ when $i = 0$, and $\sqrt{2/N}$ for all other values of i .
I have written a function called *create_DCT_Matrix* to determine this matrix *D*.

6.2 The matrix *D* called the DCT dictionary is stored and plotted in Image J (Fig. 10a). The DC coefficients of all the basis vector (indicated by rows) are also similar (Fig. 10b). Note that the DC coefficients are $1/\sqrt{256} = 0.0625$.

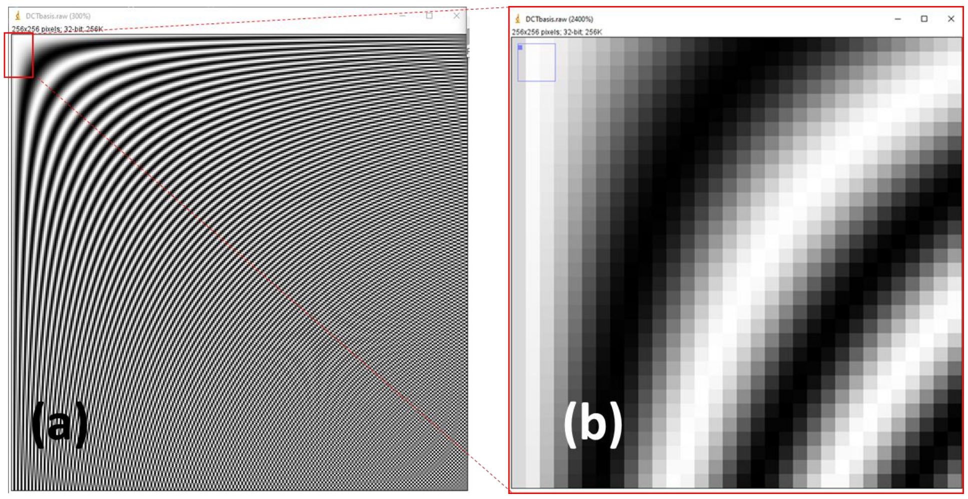


Fig.10 (a) A matrix containing all the DCT basis vectors (in rows) of a 1D signal with length 256. (b) Zoomed in version of DC coefficients of the basis vectors indicating that they are identical.

6.3 The transpose of the matrix *D* is also estimated to get the inverse synthesis IDCT dictionary. The DCT dictionary (*D*) and the inverse synthesis IDCT dictionary (D^T) is plotted in Fig. 11

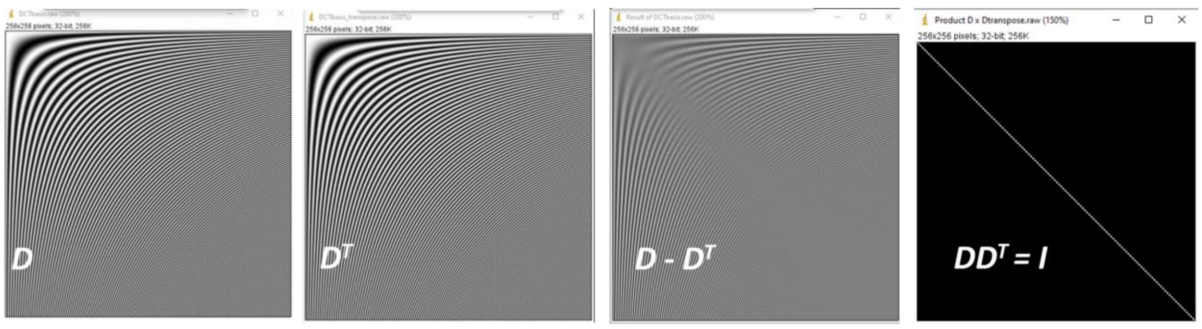


Fig.11 The DCT Dictionary (*D*), the inverse synthesis IDCT dictionary (D^T), their difference and product.

D and D^T look quite identical though they are not the same, as indicted in Fig. 11. I have also verified that DD^T= the identity matrix, and is also plotted in Fig. 11. Hence, we conclude that D^T is the inverse of *D*.

Task 7 : Discrete Cosine Transform

7.1 The Discrete Cosine Transform (DCT) of a matrix (or image) X is given by

$$\mathrm{DCT}(X) = DXD^T, \qquad \text{Eq. 5}$$

, where *D* is the DCT dictionary. I have written a function (called *dct_transform*) that calculates the DCT of an input image. The DCT of the lena image after adjusting the levels/window for clarity is is shown in Fig. 12a. I note that most of the DCT coefficients are low frequency.

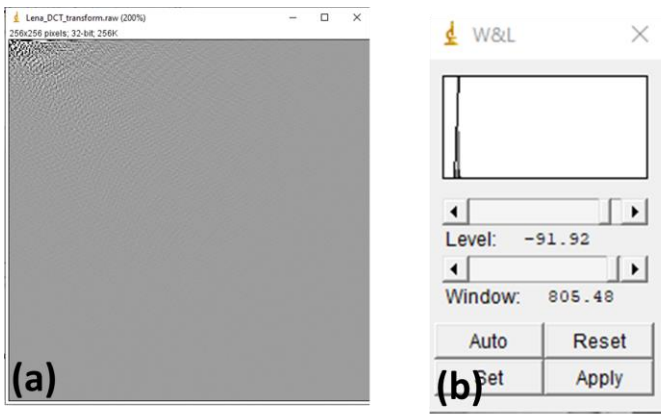


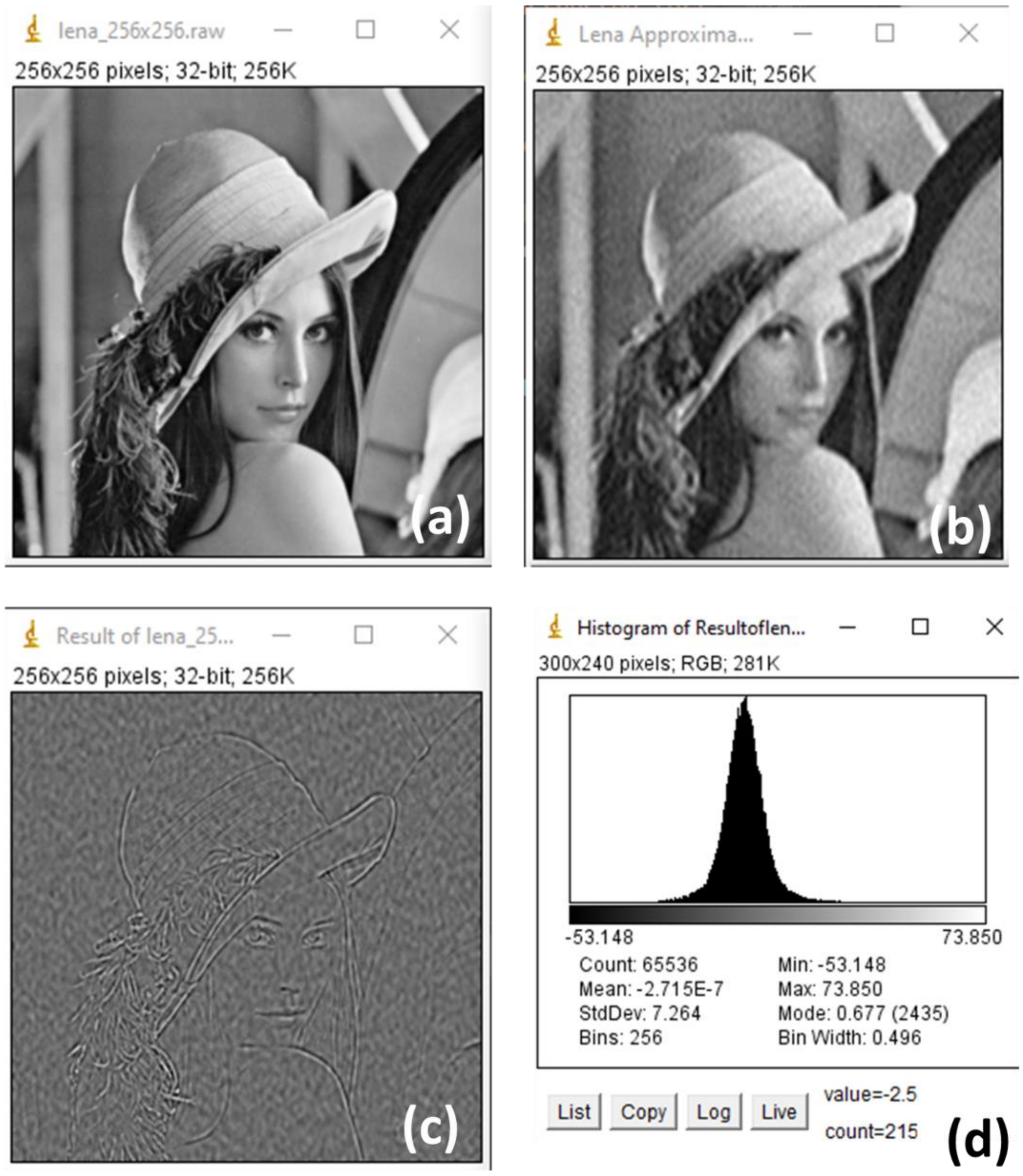
Fig.12 (a) DCT transform of Lena image with (b) adjusted window/levels for visual clarity.

7.2 I have written a function (called *cut_threshold_matrix*) which approximates the DCT value by truncating values whose absolute values are less than a threshold.

7.3 Following the threshold, the inverse DCT of a matrix X is calculated to reconstruct the original image. The inverse DCT is calculated using the following relation "

$$I\mathrm{DCT}(X) = D^T X D, \qquad \text{Eq. 6}$$

In Fig. 13b, we plot the reconstructed image, assuming a threshold value of 30. Note that the quality of the reconstructed image reduces, as especially some of the high frequency components are set to zero. The difference of the reconstructed image with the original image, and histogram of the difference is shown in Fig. 13c and Fig. 13d. We further calculate the PSNR for different values of threshold and the results are tabulated in Fig. 13.


lena_256x256.raw
256x256 pixels; 32-bit; 256K
(a)
Lena Approxima...
256x256 pixels; 32-bit; 256K
(b)
Result of lena_25...
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflen...
300x240 pixels; RGB; 281K
-53.148
73.850
List
Copy
Log
Live
(d)

```
Count: 65536        Min: -53.148
Mean: -2.715E-7     Max: 73.850
StdDev: 7.264       Mode: 0.677 (2435)
Bins: 256           Bin Width: 0.496
value=-2.5
count=215
```

Threshold	0.1	0.2	0.5	1.0	2.0	5.0	10.0	20.0	30.0	50.0	100.0
PSNR	89.2	80.0	68.2	59.4	51.4	42.9	37.7	33.2	30.9	28.4	25.5

Fig. 13(a) Original Lena image. (b) Reconstructed Image after DCT, thresholding and inverse DCT. The threshold value is assumed to be 30. (c) Difference between the original and reconstructed image. (d) Histogram of the difference image. (Table) The MSE for different values of Threshold.

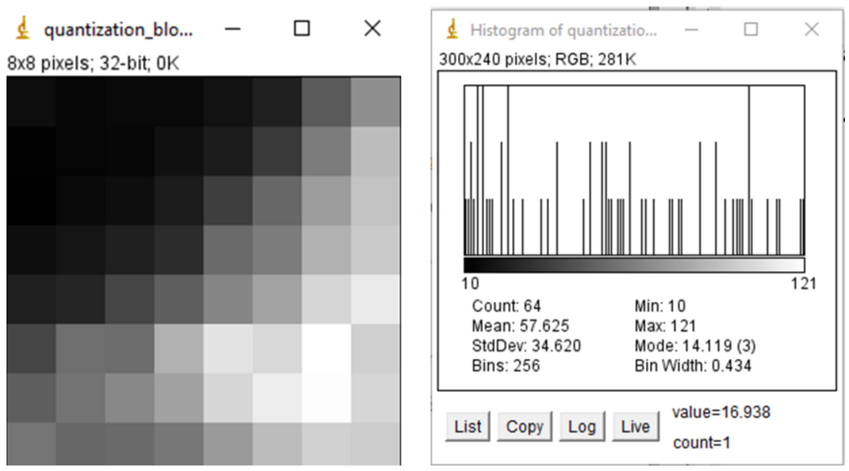
Session 4 : Basic encoding/decoding of images with quantized DCT coefficients.

Task 8: Lossy JPEG Image Approximation

8.1 The standard JPEG quantization weights Q matrix at 50 % quality is given by

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \qquad \text{Eq. 7}$$

The last row and right columns (i.e. bottom right) have larger values compared to the top left of the matrix. This is because DC components are at the lop left and high frequency components are at the bottom right. While encoding, we perform an element by element division of the DCT with the Q matrix We want to minimize the high frequency components and round them to zero for the RLE encoding (and thereby JPEG image compression), and hence the Q values at the bottom right are high. The quantization matrix and its histogram is shown in Fig. 14.

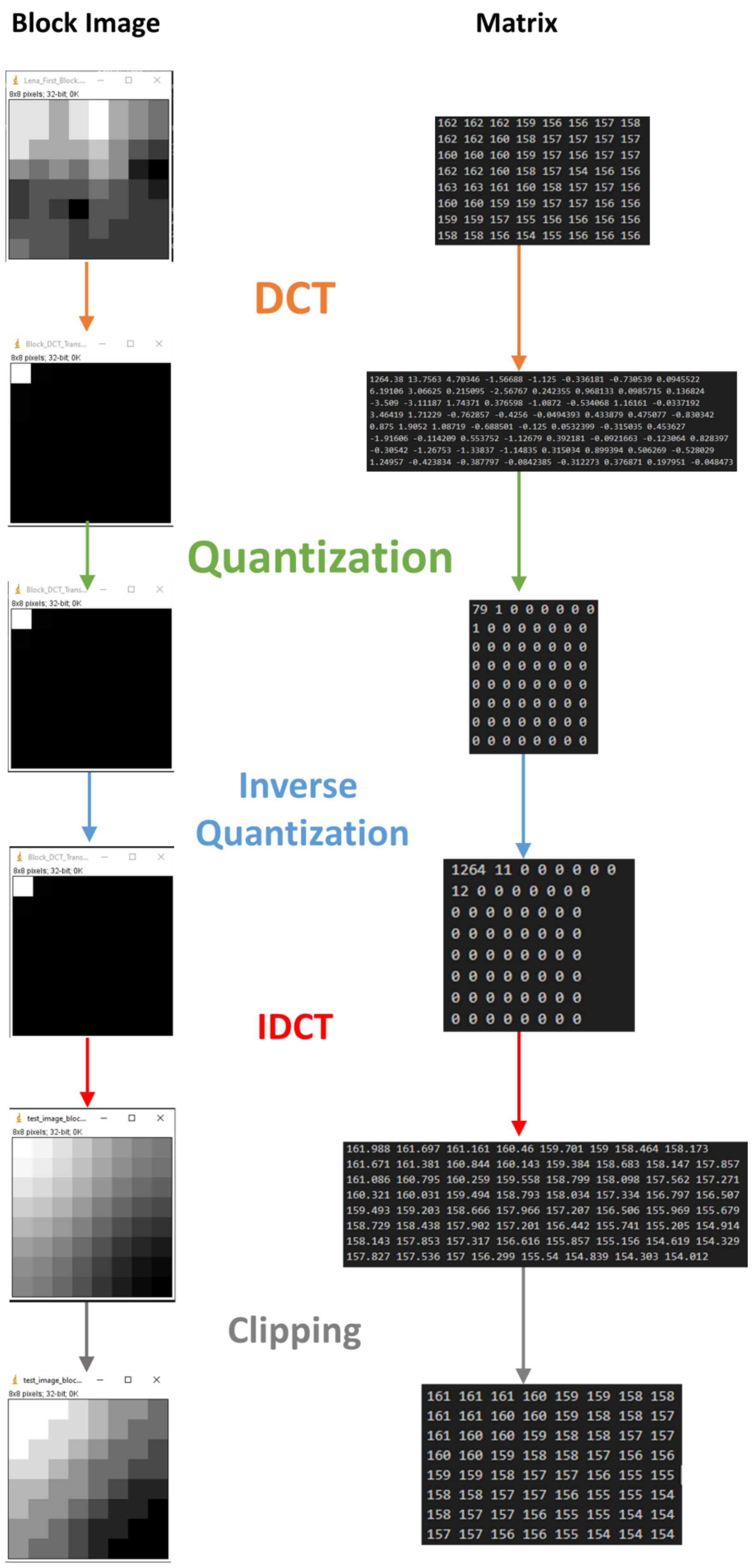


Count: 64
Mean: 57.625
StdDev: 34.620
Bins: 256
Min: 10
Max: 121
Mode: 14.119 (3)
Bin Width: 0.434
value=16.938
count=1

Fig. 14 The 8×8 quantization matrix and its histogram.

8.2-8.3 I have written an function (called *approximate_clip_image)* that applies for each block the following : DCT, Quantization, Inverse Quantization, Inverse DCT and clipping to integers between 0 and 255. As an example, I have shown the intermediate images (in ImageJ) and their corresponding matrix values (in Visual code) for the first block of the Lena Image in Fig. 15. The following are my observations during the process.

- DCT coefficients are dominated by low frequency coefficients, while the high frequency components are very small.
- After quantization, all the coefficients are reduced to smaller integers, as they are divided by the elements in the quantization matrix and rounded. Note that the high frequency components are mostly reduced to zero in this process.
- The inverse quantization process restores the DCT coefficients (mainly the low frequency components), while ensuring that high frequency components are mainly zero.
- The inverse DCT reconstructs an approximate image, but the elements are in floating point.
- The clipping process clips the floating values to integers and reconstructs the image.



```
162 162 162 159 156 156 157 158
162 162 160 158 157 157 157 157
160 160 160 159 157 156 157 157
162 162 160 158 157 154 156 156
163 163 161 160 158 157 157 156
160 160 159 159 157 157 156 156
159 159 157 155 156 156 156 156
158 158 156 154 155 156 156 156
```

```
1264.38 13.7563 4.70346 -1.56688 -1.125 -0.336181 -0.730539 0.0945522
6.19106 3.06625 0.215095 -2.56767 0.242355 0.968133 0.0985715 0.136824
-3.509 -3.11187 1.74371 0.376598 -1.0872 -0.534068 1.16161 -0.0337192
3.46419 1.71229 -0.762857 -0.4256 -0.0494393 0.433879 0.475077 -0.830342
0.875 1.9052 1.08719 -0.688501 -0.125 0.0532399 -0.315035 0.453627
-1.91606 -0.114209 0.553752 -1.12679 0.392181 -0.0921663 -0.123064 0.828397
-0.30542 -1.26753 -1.33837 -1.14835 0.315034 0.899394 0.506269 -0.528029
1.24957 -0.423834 -0.387797 -0.0842385 -0.312273 0.376871 0.197951 -0.048473
```

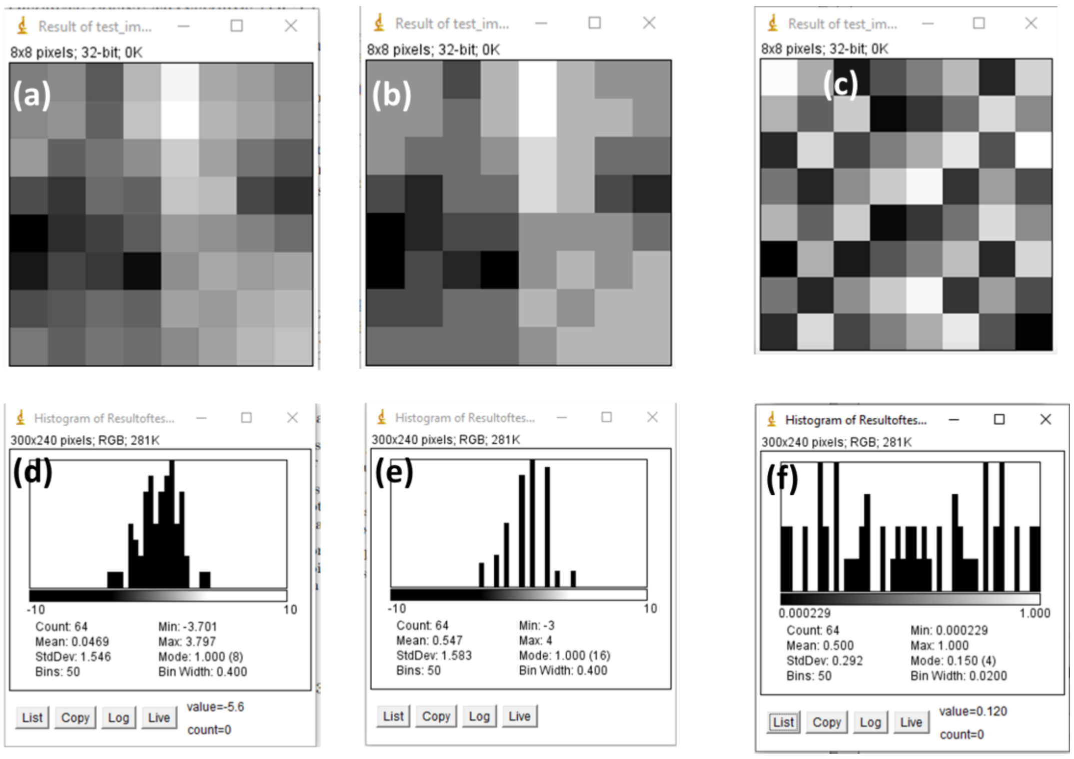
```
79 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
1264 11 0 0 0 0 0 0
12 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
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
161.988 161.697 161.161 160.46 159.701 159 158.464 158.173
161.671 161.381 160.844 160.143 159.384 158.683 158.147 157.857
161.086 160.795 160.259 159.558 158.799 158.098 157.562 157.271
160.321 160.031 159.494 158.793 158.034 157.334 156.797 156.507
159.493 159.203 158.666 157.966 157.207 156.506 155.969 155.679
158.729 158.438 157.902 157.201 156.442 155.741 155.205 154.914
158.143 157.853 157.317 156.616 155.857 155.156 154.619 154.329
157.827 157.536 157 156.299 155.54 154.839 154.303 154.012
```

```
161 161 161 160 159 159 158 158
161 161 160 160 159 158 158 157
161 160 160 159 158 158 157 157
160 160 159 158 158 157 156 156
159 159 158 157 157 156 155 155
158 158 157 157 156 155 155 154
158 157 157 156 155 155 154 154
157 157 156 156 155 154 154 154
```

Fig. 15 The first 8×8 block of the lena image subject to approximation (DCT, Quantization, Inverse Quantization & IDCT) and clipping. The resulting images in imageJ and numerical values as obtained from the C++ code are shown.

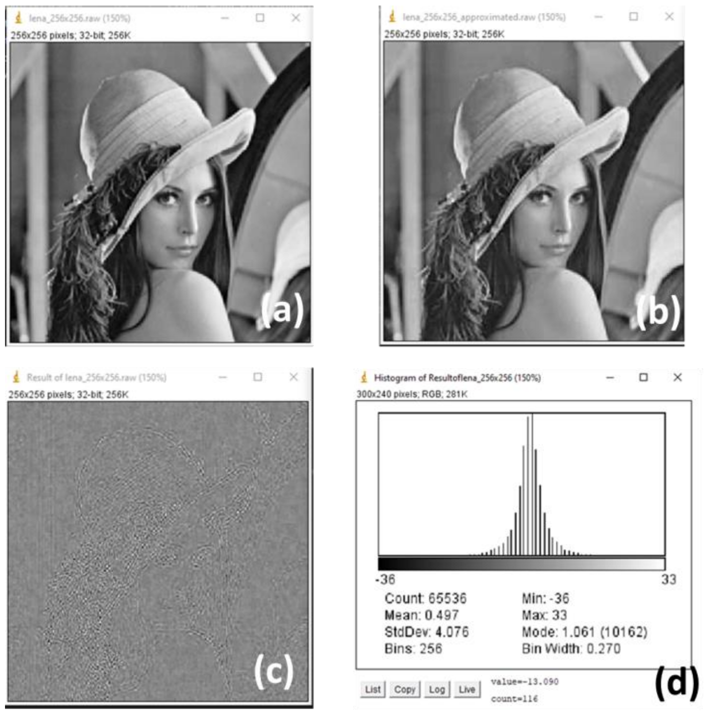


```
(d)
-10                              10
Count: 64          Min: -3.701
Mean: 0.0469       Max: 3.797
StdDev: 1.546      Mode: 1.000 (8)
Bins: 50           Bin Width: 0.400
value=-5.6
count=0

(e)
-10                              10
Count: 64          Min: -3
Mean: 0.547        Max: 4
StdDev: 1.583      Mode: 1.000 (16)
Bins: 50           Bin Width: 0.400

(f)
0.000229                         1.000
Count: 64          Min: 0.000229
Mean: 0.500        Max: 1.000
StdDev: 0.292      Mode: 0.150 (4)
Bins: 50           Bin Width: 0.0200
value=0.120
count=0
```

Fig. 15 Differences between (a) the first block of lena image and its approximation based on DCT, Quantization, Inverse Quantization and IDCT. (b) the first block of lena image and the clipped image after approximation. (c) the approximated image and clipped image. (d),(e) and (f) corresponds to the histogram of the differences estimated in (a), (b) and (c) respectively.

Fig. 16 plots the difference between the first block of the lena image, and the approximated/clipped blocks.

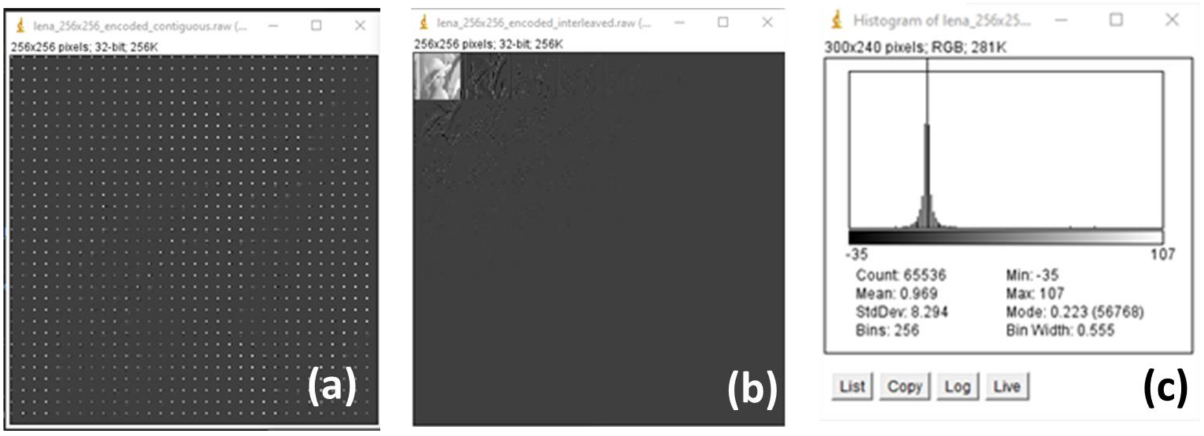
8.4 The above procedure was only shown for the first block of the lena image. This is repeated for all the 8*8 blocks in the image, and the image is reconstructed (Fig. 16b). The difference between the original image and the reconstructed image is plotted in Fig. 16c, and the histogram of the difference is plotted in Fig 16d. The finite difference between the two images is due to the rounding step when the DCT coefficients are quantized. We also note that the MSE of the reconstructed image is $4.076^2 = 16.61$, resulting in a PSNR of ~ 35.9 dB.


lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_approximated.raw (150%)
256x256 pixels; 32-bit; 256K
(b)
Result of lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflena_256x256 (150%)
300x240 pixels; RGB; 281K
(d)
List
Copy
Log
Live

```
-36                              33
Count: 65536       Min: -36
Mean: 0.497        Max: 33
StdDev: 4.076      Mode: 1.061 (10162)
Bins: 256          Bin Width: 0.270
value=-13.090
count=116
```

Fig. 16 (a) Original Lena image. (b) Reconstructed Image after performing DCT, Q, IQ, IDCT and clipping. (c) Difference between the original and reconstructed image. (d) Histogram of the difference image.

8.5 I have split the *approximate_clip_image* function into *encode_image* and *decode_image* functions, and have been able to reproduce the same results as Fig. 16 after encoding and decoding. The quantized DCT coefficients of the lena image are shown in Fig. 17a (contiguous representation) and its histogram and characteristics are plotted in Fig. 17c.


lena_256x256_encoded_contiguous.raw (...
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_encoded_interleaved.raw (...
256x256 pixels; 32-bit; 256K
(b)
Histogram of lena_256x25...
300x240 pixels; RGB; 281K
-35
107
List
Copy
Log
Live
(c)

```
Count: 65536      Min: -35
Mean: 0.969       Max: 107
StdDev: 8.294     Mode: 0.223 (56768)
Bins: 256         Bin Width: 0.555
```

Fig. 17 (a) Contiguous representation (b) Interleaved representation and (c) The histogram of the quantized DCT coefficients.

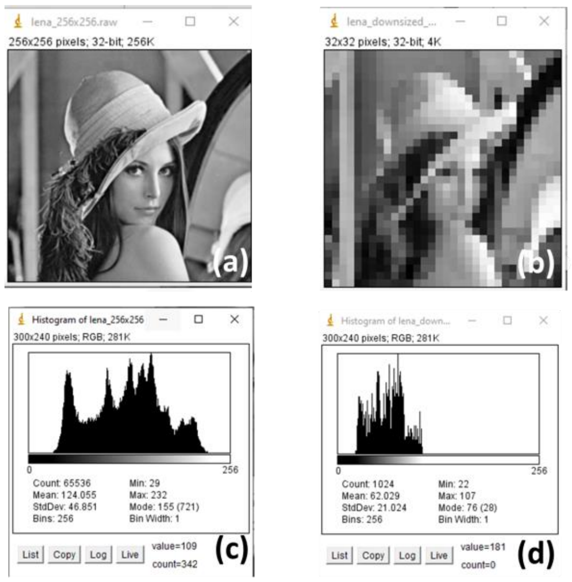
Task 9: Packing DCT coefficients

9.1-9.2 I wrote functions (*contiguous_layout_matrix and interleaved_layout_matrix*) to layout the quantized with a 32× 32 grid of 8× 8 contiguous coefficients, as well as with a 8× 8 grid of 32× 32 interleaved coefficients. The two layouts are compared side-by-side in Fig. 17a and Fig. 17b. The interleaved layout is a better representation, as each set of (32× 32) interleaved coefficients correspond to a specific frequency component of the 8× 8 blocks. E.g. the first 32× 32 set in the interleaved representation stores the DC components of all the 8× 8 blocks. Hence, in the interleaved representation, the high frequency components of all the 8× 8 blocks will be clubbed together, which will be approximated as zero due to quantization. This allows for good compression with RLE encoding in Session 5 and Session 6.

Session 5 : Advanced encoding/decoding of the quantized DCT coefficients with Delta Encoding (for DC coefficients) and Run Length Encoding (for AC Coefficients)

Task 10: Delta Encoding of DC coefficients

10.1 The first 32× 32 elements in the interleaved representation corresponds to the DC coefficients of all the 8× 8 blocks. The original and downsized image from the DC coefficients are compared in Fig. 18a and Fig. 18b respectively.


lena_256x256.raw
256x256 pixels; 32-bit; 256K
(a)
lena_downsized_...
32x32 pixels; 32-bit; 4K
(b)
Histogram of lena_256x256
300x240 pixels; RGB; 281K
0
256
List
Copy
Log
Live
(c)
Histogram of lena_down...
300x240 pixels; RGB; 281K
0
256
List
Copy
Log
Live
(d)

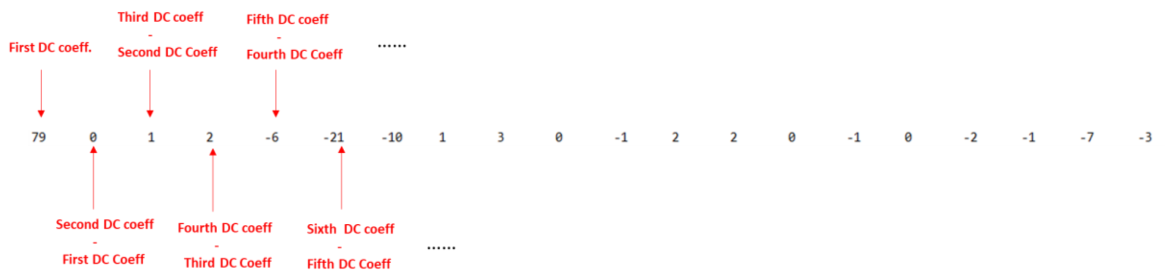
```
Count: 65536      Min: 29
Mean: 124.055     Max: 232
StdDev: 46.851    Mode: 155 (721)
Bins: 256         Bin Width: 1
value=109
count=342
```

```
Count: 1024       Min: 22
Mean: 62.029      Max: 107
StdDev: 21.024    Mode: 76 (28)
Bins: 256         Bin Width: 1
value=181
count=0
```

Fig. 18 (a) Original Lena image (b)Downsized Lena image from DC coefficients of quantized DCT. (c), (d) correspond to the histograms of (a) and (b) respectively.

From Fig. 18a and 18b, it is evident that the quantized DC terms of each 8×8 block (i.e. average of the block in real domain), is a downsized version of the original image. The histograms of the two images are also shown in Fig. 18c and 18d. All the characteristics of the two images are summarized in the histogram. It can be inferred that the downsized image has a much smaller contrast when compared to the original image, and is evident from the reduced mean, standard deviation, minimum and maximum values. This is due to the averaging involved when estimating the DC coefficients in DCT.

10.2 All the DC coefficients are taken, and encoded (Delta Encoding) into a text file containing successive differences. The function *delta_encoding_DC* performs this in the submitted code. While printing all the 1024 (=32*32) DC coefficients is impractical in this report, I just show the first 20 Delta Encoded DC coefficients in the text file in Fig. 19.


First DC coeff.
Third DC coeff - Second DC Coeff
Fifth DC coeff - Fourth DC Coeff
......
Second DC coeff - First DC Coeff
Fourth DC coeff - Third DC Coeff
Sixth DC coeff - Fifth DC Coeff
......

```
79  0  1  2  -6  -21  -10  1  3  0  -1  2  2  0  -1  0  -2  -1  -7  -3
```

Fig. 19 First 20 delta encoded DC coefficients in the text file.

10.3 I have also been able to read the delta encoded DC terms from the text file and reconstruct the low definition image. Most of this functionality is performed by the *delta_decoding_DC* function in the code. The downsized image, the reconstructed image and their difference are shown in Fig. 20.

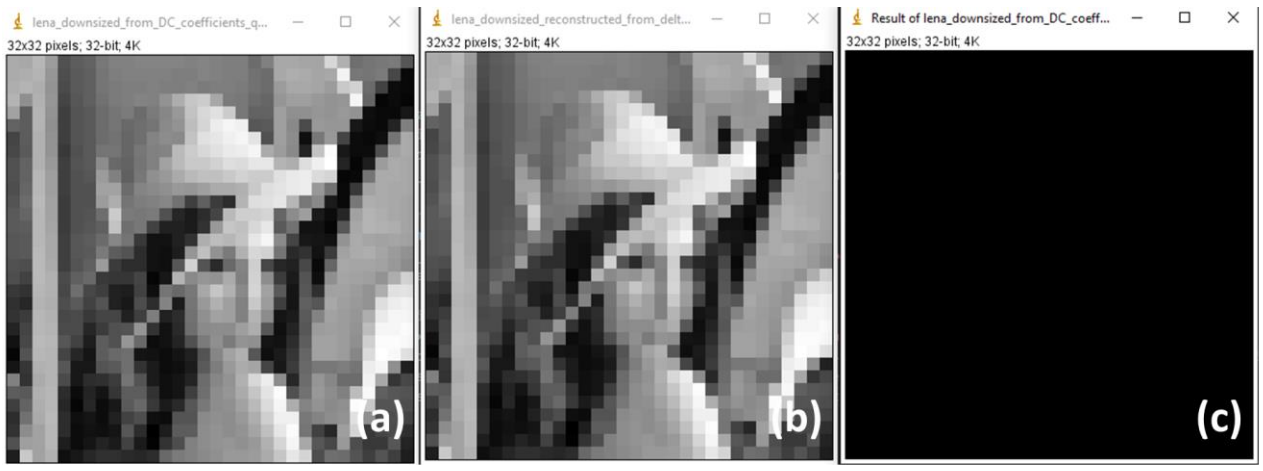

lena_downsized_from_DC_coefficients_q...
32x32 pixels; 32-bit; 4K
(a)
lena_downsized_reconstructed_from_delt...
32x32 pixels; 32-bit; 4K
(b)
Result of lena_downsized_from_DC_coeff...
32x32 pixels; 32-bit; 4K
(c)

Fig. 20. (a) The low definition lena image obtained from the DC coefficients of the quantized DCT. (b) Reconstructed low definition lena image based on Delta encoding followed by Decoding. (c) The difference between the low definition image and reconstructed image indicating that both the images are the same.

Task 11: Run Length Encoding of AC coefficients

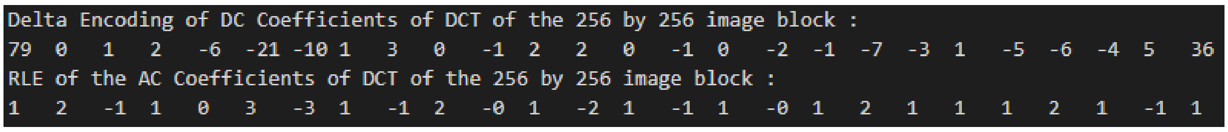
11.1 For encoding the AC terms, the following methodology has been adopted :

- We start with the interleaved representation of the quantized DCT matrix.
- The coefficients are scanned in a zigzag manner, wherein the DC coefficients (the first 32*32 grid) are omitted. The result of the zigzag scan will then be a 1-dimentional array. The code for the zigzag scan has been inspired and modified from the website : https://www.geeksforgeeks.org/print-matrix-zag-zag-fashion/

- We define a *run* as a set of repeated continuous values in the 1D array and store it with two numbers : (i) the value that is repeated continuously (ii) the number of repetitions of the value in the run.
- Hence, for each run in the array, we store two numbers, and perform it for all the runs in the 1D array (which in turn is obtained from the zigzag scan).

As an example, if the output of the zigzag scan is (5,5,5,5,2,3,3,10,100,5,0,0,0,0,0,0), it will be encoded as 5,4,2,1, 3,2, 10,1, 100,1,5,1,0,6.

11.2 I have modified the *encode_image* function to also print a sequence of the encoded runs from the AC coefficients with the above methodology. This function also estimates the length of RLE, which is 19758 in this case. For completeness, I have also included the printing of the Delta encoded DC terms in the function. The screenshot (from Visual code) of the first few encoded coefficients is shown in Fig. 21.



```
Delta Encoding of DC Coefficients of DCT of the 256 by 256 image block :
79  0   1   2   -6  -21 -10 1   3   0   -1  2   2   0   -1  0   -2  -1  -7  -3  1   -5  -6  -4  5   36
RLE of the AC Coefficients of DCT of the 256 by 256 image block :
1   2   -1  1   0   3   -3  1   -1  2   -0  1   -2  1   -1  1   -0  1   2   1   1   1   2   1   -1  1
```

Fig. 21. Screenshot of the first few encoded DC and AC coefficients with Delta Encoding and Run Length Encoding (RLE) respectively.

11.3 The *decode_image* function has been further modified to take the encoded DC (Delta encoding) and AC (RLE) encoded integers, decode them separately, and reconstruct the image back. The reconstructed image, the difference between the original and reconstructed image, and the histogram of the difference is shown in Fig. 22b, 22c and 22d respectively. The MSE of the reconstructed image is $4.065^2 = 16.52$, resulting in a PSNR of ~ 35.95 dB.

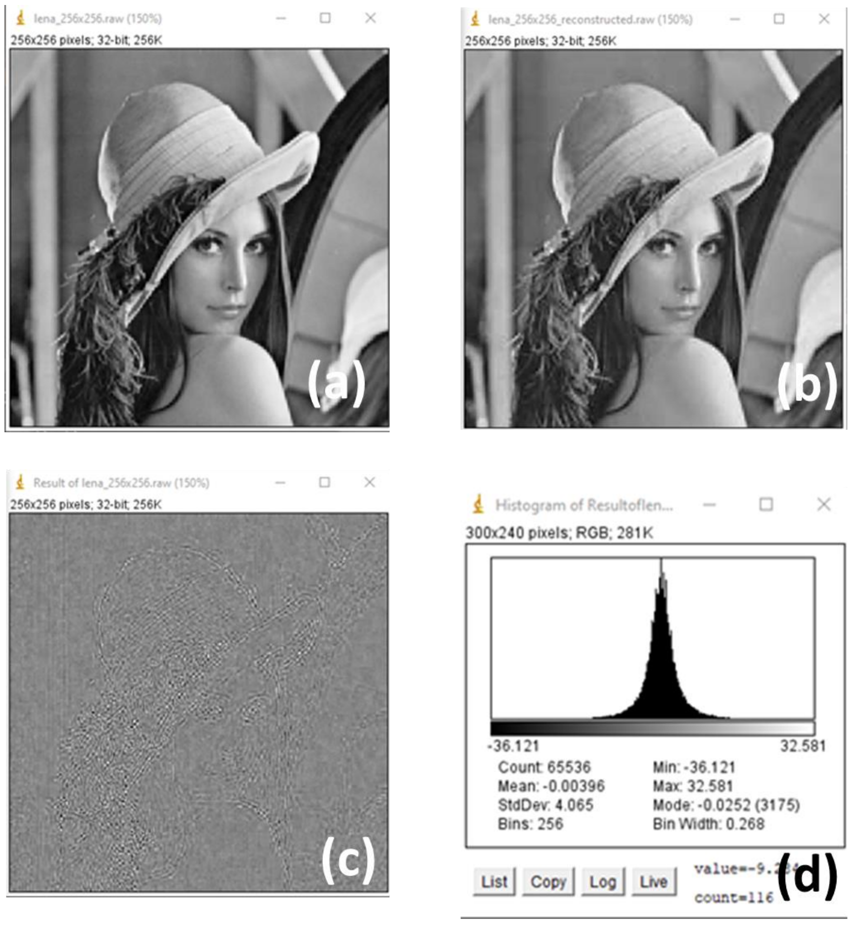

lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_reconstructed.raw (150%)
256x256 pixels; 32-bit; 256K
(b)
Result of lena_256x256.raw (150%)
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflen...
300x240 pixels; RGB; 281K
-36.121
32.581
Count: 65536
Mean: -0.00396
StdDev: 4.065
Bins: 256
Min: -36.121
Max: 32.581
Mode: -0.0252 (3175)
Bin Width: 0.268
List
Copy
Log
Live
value=-9.2
count=116
(d)

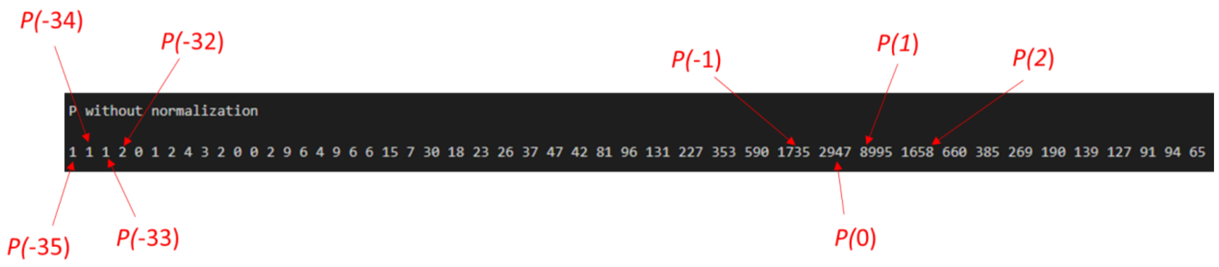
Fig. 22 (a) Original Lena image. (b) Reconstructed Image after decoding the encoded AC and DC integer coefficients. (c) Difference between the original and reconstructed image. (d) Histogram of the difference image.

Task 12: Discrete Probability Density Functions

12.1 We will now postprocess the RLE of AC coefficients and understand its distribution. In the code, this is performed by the function *process_AC_encoding.* As each run is represented by a pair of numbers (value and repetition), the number of runs required for encoding the AC coefficients is the length of the RLE divided by 2. The length of RLE has been calculated in Task 11.2 and is 19758. Hence, the number of runs M to encode all AC coefficients = 19758/2 = 9879.

We calculate the minimum and maximum value in the RLE encoding to be -35 and 19018 respectively. Hence, the maximum run length N in the encoding is 19018. Furthermore, there are 1538 unique symbols in the encoding.

For the distribution, we estimate ***P(i)*** (also called *count_val[]* in the code) such that $P(i)$ is the number of occurrences of i in the encoding. E.g. *P*(-35) corresponds to number of occurrences of -35, *P*(-34) corresponds to number of occurrences of -34 , *P*(19018) corresponds to the number of occurrences of 19018 in the encoding and so on. A snapshot of the first few elements of P for our RLE encoding is shown in Fig. 23.

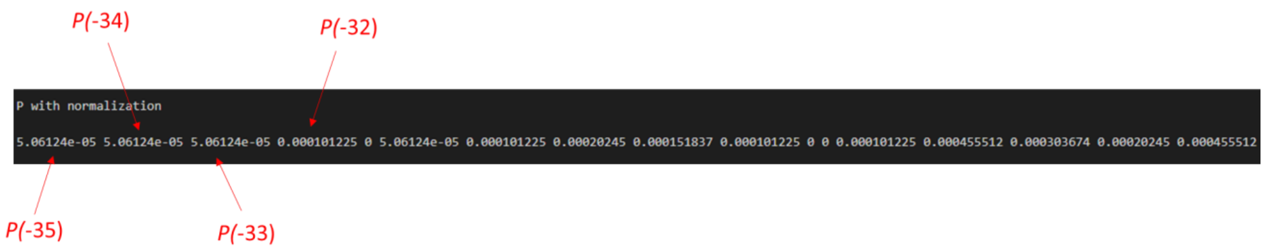


```
P without normalization
1 1 1 2 0 1 2 4 3 2 0 0 2 9 6 4 9 6 6 15 7 30 18 23 26 37 47 42 81 96 131 227 353 590 1735 2947 8995 1658 660 385 269 190 139 127 91 94 65
```

Fig. 23. Number of occurrences of each symbol in the encoding.

It is evident from Fig. 23 that out of 19758 values (length of RLE encoding), over 75 % of the values belong to the set [-1,1,0,2] and the remainder is a small portion. I further note that the number of occurrences for large values (> 50) is mostly zero.

12.2 To obtain the normalized probability distribution of *P*, we divide each element in *P* by the length of RLE (i.e. 19758). The first few elements of the normalized probability distribution are shown in Fig. 24.



```
P with normalization
5.06124e-05 5.06124e-05 5.06124e-05 0.000101225 0 5.06124e-05 0.000101225 0.00020245 0.000151837 0.000101225 0 0 0.000101225 0.000455512 0.000303674 0.00020245 0.000455512
```

Fig. 24. Normalized probability of each symbol in the encoding.

12.3 An entropy function (called *calculate_entropy* in the code) was written to estimate the theoretical number of bits per symbol. The entropy is estimated using the following formula

$$\text{Entropy} = \text{Minimum number of Bits per Symbol} = \sum_{i\ \epsilon\ \text{all symbols}} p_i \log_2(1/p_i) \qquad \text{Eq. 2}$$

For our RLE encoding, we estimate an entropy of 3.073 minimum bits per symbol. To calculate the minimum file size, we multiple the entropy (3.073) with the length (19758) of the encoding.

Hence, minimum filesize for encoding the AC coefficients with RLE is 3.073*19758 = 60713 bits.

Session 6 : Compression of images by combining binary Golomb codes with the encoding methods in Session 5.

Task 13: Exponential-Golomb Variable Length Code (VLC)

13.1 A golomb function (called *golomb* in the code) that maps a signed integer to a Golomb string was written. Prior to this, we need to calculate the Golomb code of an integer without considering sign, with the procedure below :

- We first add 1 to the number
- Convert the resulting number into its binary format and estimate its $length$. Note : For the binary representation of negative numbers, I assume the 2's complement.
- Append ($length$ -1) zeros in front of the most significant bit in the binary representation.

The Golomb code of the signed integer n can then be obtained as follows :

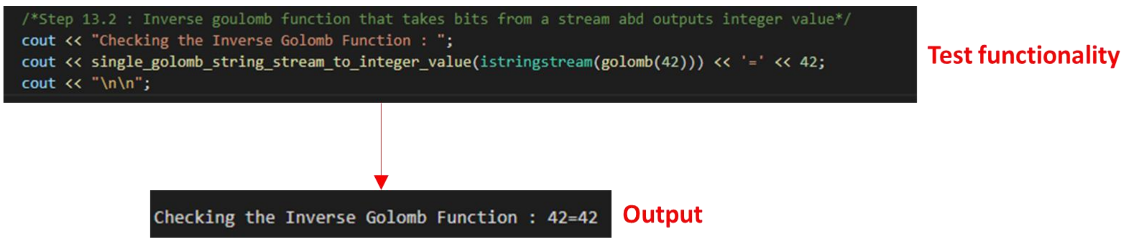
- IF n is positive, its Golomb code would that of the Golulomb code of $2n$ -1 without considering sign.
- If n is negative, its Golomb code would that of the unsigned Golulomb -$2n$ without considering sign.

I tried a few examples with the code. The results are tabulated below and are as expected.

Number	Golomb Code Without Considering Sign	Golomb Code After Considering Sign
-5	-	0001011
-4	-	0001001
-3	-	00111
-2	-	00101
-1	-	011
0	1	1
1	010	010
2	011	00100
3	00100	00110
4	00101	0001000
5	00110	0001010

Fig. 25. Golomb code of few integers with and without considering sign.

13.2 An inverse golomb function (called *single_golomb_string_stream_to_integer_value* in the code) that maps an input Golomb code string stream to an integer was written. This is based on reversing algorithm used in Task 13.2. Based on the example in the *ivt_exercises_2020.pdf* document, I tested the output of the inverse golomb code in Fig. 26 and provides expected results.



```
/*Step 13.2 : Inverse goulomb function that takes bits from a stream abd outputs integer value*/
cout << "Checking the Inverse Golomb Function : ";
cout << single_golomb_string_stream_to_integer_value(istringstream(golomb(42))) << '=' << 42;
cout << "\n\n";
```

```
Checking the Inverse Golomb Function : 42=42
```

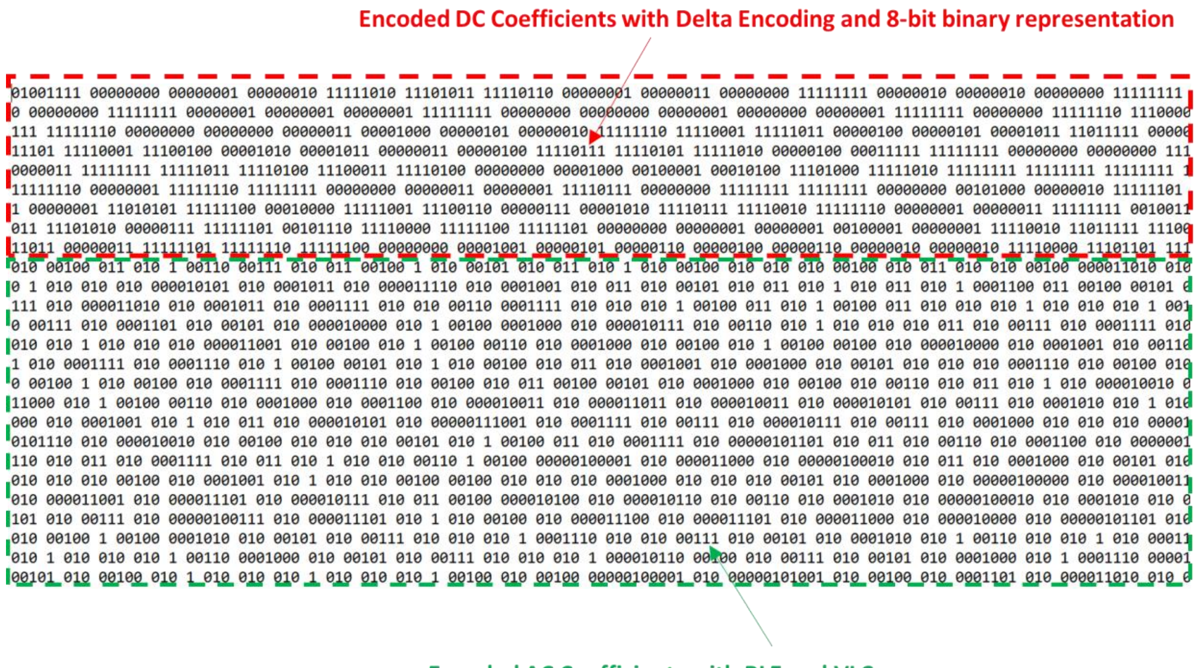
Fig. 26. Test of the inverse Golomb code with console output

Task 14: Lossy Grayscale Image Compression

14.1 I have written a *compress* function which performs the following steps on an image :

- Calculate the quantized DCT matrix and arrange it in the interleaved representation.
- Encode the DC coefficients by Delta Encoding, and represent the resulting integers by their 8-bit binary equivalents.
- Encode the AC coefficients by RLE encoding, and represent the resulting integers by the Golomb Variable Length Code.
- Store the binary representations of the 1024 (i.e. 32*32) encoded DC coefficients into the first part of a text file.
- Write the Golomb representations of the encoded AC coefficients into the text file, after the DC part.

A screenshot of the generated text file is shown below.


Encoded DC Coefficients with Delta Encoding and 8-bit binary representation
Encoded AC Coefficients with RLE and VLC

```
01001111 00000000 00000001 00000010 11111010 11101011 11110110 00000001 00000011 00000000 11111111 00000010 00000010 00000000 11111111
0 00000000 11111111 00000001 00000001 00000001 11111111 00000000 00000000 00000001 00000000 00000001 11111111 00000000 11111110 1110000
111 11111110 00000000 00000000 00000011 00001000 00000101 00000010 11111110 11110001 11111011 00000100 00000101 00001011 11011111 00000
11101 11110001 11100100 00001010 00001011 00000011 00000100 11110111 11110101 11111010 00000100 00011111 11111111 00000000 00000000 111
0000011 11111111 11111011 11110100 11100011 11110100 00000000 00001000 00100001 00010100 11101000 11111010 11111111 11111111 11111111 1
11111110 00000001 11111110 11111111 00000000 00000011 00000001 11110111 00000000 11111111 11111111 00000000 00101000 00000010 11111101
1 00000001 11010101 11111100 00010000 11111001 11100110 00000111 00001010 11110111 11110010 11111110 00000001 00000011 11111111 0010011
011 11101010 00000111 11111101 00101110 11110000 11111100 11111101 00000000 00000001 00000001 00100001 00000001 11110010 11011111 11100
11011 00000011 11111101 11111110 11111100 00000000 00001001 00000101 00000110 00000100 00000110 00000010 00000010 11110000 11101101 111
010 00100 011 010 1 00110 00111 010 011 00100 1 010 00101 010 011 010 1 010 00100 010 010 010 00100 010 011 010 010 00100 000011010 010
0 1 010 010 010 000010101 010 0001011 010 000011110 010 0001001 010 011 010 00101 010 011 010 1 010 011 010 1 0001100 011 00100 00101 0
111 010 000011010 010 0001011 010 0001111 010 010 00110 0001111 010 010 010 1 00100 011 010 1 00100 011 010 010 010 1 010 010 010 1 001
0 00111 010 0001101 010 00101 010 000010000 010 1 00100 0001000 010 000010111 010 00110 010 1 010 010 010 011 010 00111 010 0001111 010
010 010 1 010 010 010 000011001 010 00100 010 1 00100 00110 010 0001000 010 00100 010 1 00100 00100 010 000010000 010 0001001 010 00110
1 010 0001111 010 0001110 010 1 00100 00101 010 1 010 00100 010 011 010 0001001 010 0001000 010 00101 010 010 010 0001110 010 00100 010
0 00100 1 010 00100 010 0001111 010 0001110 010 00100 010 011 00100 00101 010 0001000 010 00100 010 00110 010 011 010 1 010 000010010 0
11000 010 1 00100 00110 010 0001000 010 0001100 010 000010011 010 000011011 010 000010011 010 000010101 010 00111 010 0001010 010 1 010
000 010 0001001 010 1 010 011 010 000010101 010 00000111001 010 0001111 010 00111 010 000010111 010 00111 010 0001000 010 010 010 00001
0101110 010 000010010 010 00100 010 010 010 00101 010 1 00100 011 010 0001111 010 00000101101 010 011 010 00110 010 0001100 010 0000001
110 010 011 010 0001111 010 011 010 1 010 010 00110 1 00100 00000100001 010 000011000 010 00000100010 010 011 010 0001000 010 00101 010
010 010 010 00100 010 0001001 010 1 010 010 00100 00100 010 010 010 0001000 010 010 010 00101 010 0001000 010 00000100000 010 000010011
010 000011001 010 000011101 010 000010111 010 011 00100 000010100 010 000010110 010 00110 010 0001010 010 00000100010 010 0001010 010 0
101 010 00111 010 00000100111 010 000011101 010 1 010 00100 010 000011100 010 000011101 010 000011000 010 000010000 010 00000101101 010
010 00100 1 00100 0001010 010 00101 010 00111 010 010 010 1 0001110 010 010 00111 010 00101 010 0001010 010 1 00110 010 010 1 010 00011
010 1 010 010 010 1 00110 0001000 010 00101 010 00111 010 010 010 1 000010110 00100 010 00111 010 00101 010 0001000 010 1 0001110 00001
00101 010 00100 010 1 010 010 010 1 010 010 010 1 00100 010 00100 00000100001 010 00000101001 010 00100 010 0001101 010 000011010 010 0
```

Fig. 27. Snapshot of the file encoding the DC and AC coefficient with 1's and 0's. The first part corresponds to the DC coefficients and the remainded to the AC coefficients.

I also calculate the <u>total number of bits used for the encoding to be 83498</u>, of which 52082 were 0's and 31416 were 1's

The number of bits per pixel with this encoding scheme can then be calculated as

$$\textbf{Bits per Pixel} = \frac{\textbf{Total Number of Bits}}{\textbf{Number of Pixels}} = \frac{\mathbf{83498}}{\mathbf{256 \times 256}} = \mathbf{1.27}$$

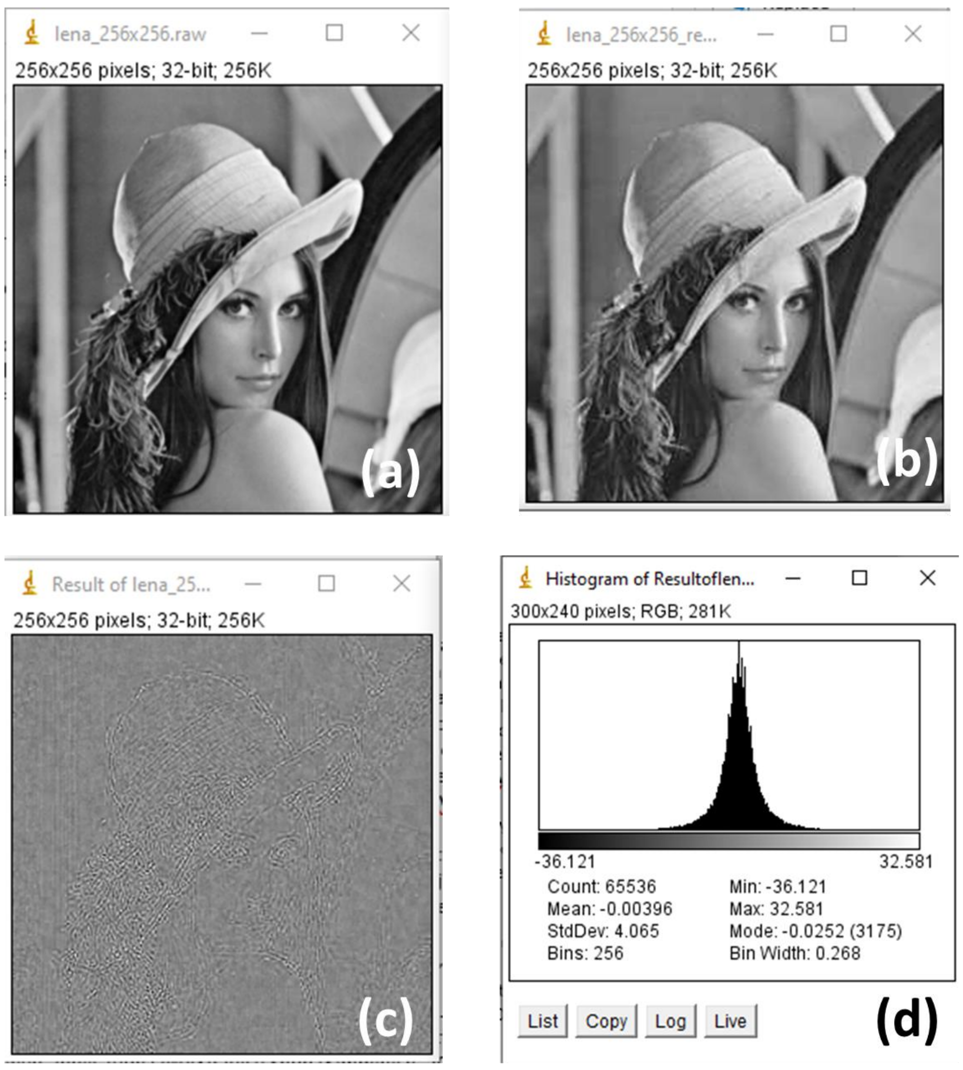
Also note that the RAW image required 32 bits per pixel. Hence, <u>the encoding scheme (with 50% JPEG Quality) improved compression by a factor of compression ratio = 32/1.27 = 25.2, when compared to the RAW Lena image</u>.

14.2 I have also written a decompress function which performs the following steps on the text file :

- Read the AC and DC encoded binary strings from the textfiles as streams.
- Convert the binary string streams to their respective integer values (8-bit Binary to Decimal for DC and inverse Golomb for AC).
- Perform inverse Delta Decoding and inverse RLE Decoding for DC and AC coefficients respectively.
- Populate the Quantized DCT matrix (interleaved) with the coefficients.

- De-inteleave to get the Quantized DCT in Contiguous format.
- De-Quantize and perform IDCT of the different 8×8 blocks to reconstruct the image.

As a test, I compressed the Lena image into the text file, and decompressed/reconstructed it. The reconstructed image, the difference between the RAW and reconstructed image and the histogram of the difference is shown in Fig. 28b, 28c and 28d respectively.


lena_256x256.raw
256x256 pixels; 32-bit; 256K
(a)
lena_256x256_re...
256x256 pixels; 32-bit; 256K
(b)
Result of lena_25...
256x256 pixels; 32-bit; 256K
(c)
Histogram of Resultoflen...
300x240 pixels; RGB; 281K
-36.121
32.581
List
Copy
Log
Live
(d)

```
Count: 65536        Min: -36.121
Mean: -0.00396      Max: 32.581
StdDev: 4.065       Mode: -0.0252 (3175)
Bins: 256           Bin Width: 0.268
```

Fig. 28 (a) Original Lena image. (b) Reconstructed Image after compression and decompression. (c) Difference between the original and reconstructed image. (d) Histogram of the difference image.

Fig. 28 is very similar to Fig. 22 (which doesn't assume Golomb code), and hence is a verification that the encoding and decoding process works well. Note that the compression ratio (CR = 25.2) was calculated in Task 14.1.

14.3 Improve Compression over Current Scheme

I note that the total number of bits (83498 in Step 14.1) used for the encoding is still ~ 37% larger than the theoretical limit (60713, as calculated in Section 12.3) set by entropy for RLE. Looking back, there are few improvements that could be made in order to be closer to the theoretical limit.

For the AC coefficients, I currently encode the symbol "0" as its Golomb code 1. The symbol "1" is encoded as its Golomb code 010. However, we note from Fig. 23 that number of occurrences of the symbol "1" (8995 occurrences) in the encoding is far greater than that of the symbol "0" (2947 occur-rences). Hence, by encoding the symbol "1" as 1 (rather then 010) and symbol "0" as 010 (rather 1), we could save $2 \times (8995 - 2947) = 12096$ bits. This would reduce the total number of bits to 71402, rather than the current value of 83498 bits. By adopting similar strategies to the other symbols, where their encoding is selected based on their probability of occurrence, we can get even closer to the theoretical limit.

Conclusion

The lab exercises has involved the use of C++ and ImageJ to analyse, process and understand images. Methods behind JPEG compression such as Discrete Cosine Transform, Quantization, Delta Encoding, Run Length Encoding and Variable Length Codes has been well understood, implemented and validated. I have also proposed a potential method to improve the compression ratio over the current implementation.

Source Code

As mentioned previously, the C++ code for each of the 6 sessions are placed in separate folders named by their respective sessions. The name of the C++ files are also as per their respective session (Session1.cpp, Session2.cpp, Session3.cpp, etc.). I have been able to run all the C++ files without any trouble on Visual Studio Code. To understand the code, it may be beneficial to start from the main function, where I perform each of the task (also commented) in the session sequentially. I also have tried to provide detailed comments on the purpose of different functions where appropriate.