JPEG Image Compression

Nandan (22B0920) - Lohith (22B0947) - Varshith (22B0907)

CS 663 : Fundamentals of Digital Image Processing - Autumn 2024-25

1 Introduction

The JPEG (Joint Photographic Experts Group) standard is one of the most widely used image compression methods globally. It is a lossy compression technique designed to significantly reduce the size of image files while maintaining visual quality.

In this project, we implemented a basic JPEG-like compression algorithm focusing on key components for both grayscale and colour images. Additionally, we explored small enhancements to improve the compression process.

2 Main Steps in Compression Algorithm

Here we'll explain the important steps of the algorithm for grayscale images

- **Padding:** We would like to divide images into 8×8 patches. To do so, we want size of image array to be multiple of 8. This step adds zero padding accordingly to make the array size multiple of 8
- **Discrete Cosine Transform (DCT):** This step divides image into a set of disjoint 8×8 patches and applies Discrete Cosine Transform to each patch. This step is necessary to separate important information (low frequency) from other details (high frequency)

  The DCT matrix is given by,

$$C_{ij} = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } i = 0, \\ \sqrt{\frac{2}{N}} \cos\left(\frac{(2j+1)i\pi}{2N}\right) & \text{if } i > 0, \end{cases}$$

  where $i, j = 0, 1, \ldots, N-1$. (N = 8 here)

  To apply DCT transform to an 8×8 patch P,

$$P' = C \cdot P \cdot C^T$$

- **Quantization:** This is the lossy part of JPEG, where we divide a given 8×8 patch by a 8×8 matrix which can be brought for a given quality decided by user

  The standard quantization matrix for a quality factor of 50 is,

$$quant\_matrix_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

We compute quantization matrix for a given quality factor Q as,

$$quant\_matrix_Q = quant\_matrix_{50} \times \frac{50}{Q}$$

Then we round off the nearest integer after division

- **Ziz-zag encoding:** Elements in a patch are flattened using zig zag order. Elements are accessed in zig-zag so that in this way elements at the end of the of each flattened path all zeros will be accumulated

- **Runlength Encoding:** In each patch 1st element is considered as DC element and rest all elements are considered as AC elements. Concatenate all DC elements in one list. Now in DC elements list also store the first element as DC_first and store difference between each element and DC_first in the list. Now we will apply run-length encoding for the AC lists. In run length encoding for AC elements, we will store tuples (a,b) which represents where a represents number of consecutive zeros before this element in each patch. to represent ending of patch we will store (0,0) representing end of patch. Now this tuples list for all patches is concatenated and and flattened to get a single list for AC. The DC list, AC list after runlength encoding and DC_first is sent to next step

- **Huffman Encoding:** This is a lossless text compression technique widely used text compression. This involves building a binary tree for characters in text based on their frequency. We use this technique as DCT coefficients lie between [-1024, 1024] but number of pixels are like 1600000 (for a 400×400 image). The two list mentioned above are encoded using the huffman tree built based on the elements in the list. Fianl encoding will be a binary string.

- **Encoding Huffman tree:** Based on the fact that any node in Huffman tree has either 2 children or no child, and storing one traversal of tree (pre-order or inorder traversal), we can completely reconstruct the tree. This tree is encoded into binary

- **Final Compressed file:** This includes

  1. quality used for compression
  2. height of image
  3. width of image
  4. type of image (colour or grayscale)
  5. size of encoded huffman trees, encoded lists

6. encoded lists and encoded huffman trees (for both DC and AC lists)

All binary string are converted to bytes and stored in the final comppressed file of the image

3 Main Steps in De-compression Algorithm

- **Reading:** Read all the parameters from the file and store them
- **Huffman decoding:** First bring tree from its traversal. Then bring the Both AC and DC lists from bitstring using huffman tree
- **Runlength decoding:** Now from AC and DC lists construct all patches back by using the fact that [0,0] represents ending of the patch. Get all DC coefficients also from the DC_first value and AC and DC values to get final list of patches.
- **Reverse Quantization:** Multiply each patch with same matrix used to quantize. The precision lost at quantization step cannot be recovered (Hence JPEG is overall a lossy algorithm)
- **Inverse Discrete Cosine Transform (iDCT):** If C is same matrix as described above, we can get back the patch by,

$$P = C^T \cdot P' \cdot C$$

using the fact that C is orthogonal matrix

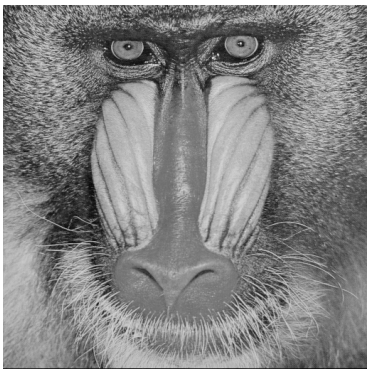
4 Data set for grayscale images

For evaluating our implementation, we have used 2 Datasets

- First one from Kaggle at [2], which are 150×150, images consisting of street, buildings, mountains, glaciers, trees, etc

- Second one, C4L data set of 512×512 size images (which we didn't had access from official site, so got them from github repo of [1])





5 Results for grayscale images

We have run our jpeg for 30 images from above two data sets for the quality factors [1, 3, 5, 10, 15, 20, 25, 40, 50, 70, 90] (These quality factors has been selected for better visuals in graphs)

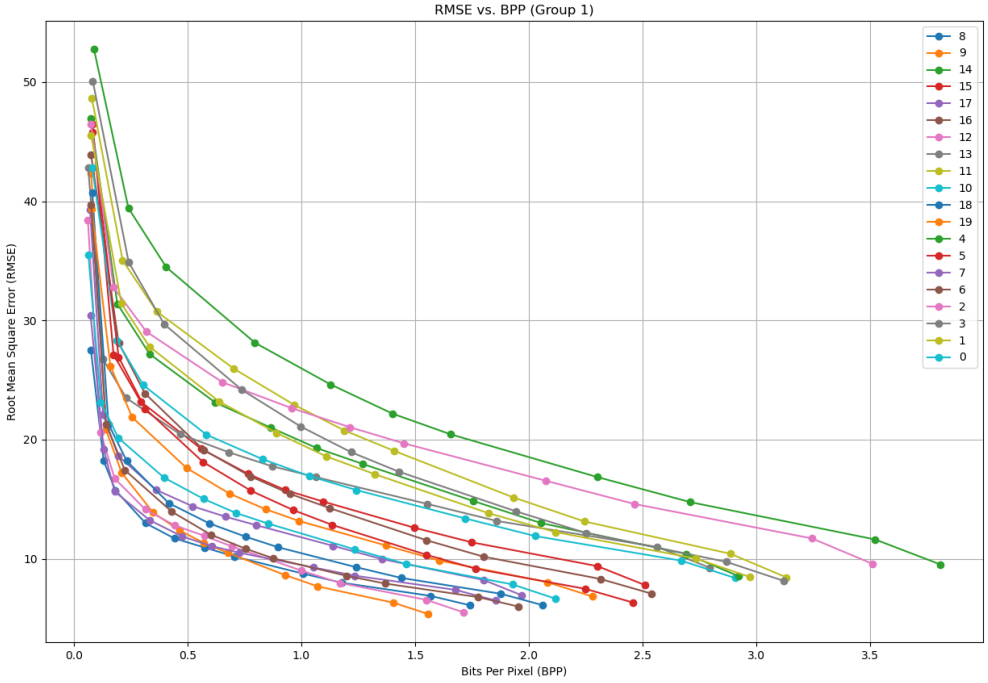


Figure 3: RSME vs BPP plot for 20 images of size 150×150

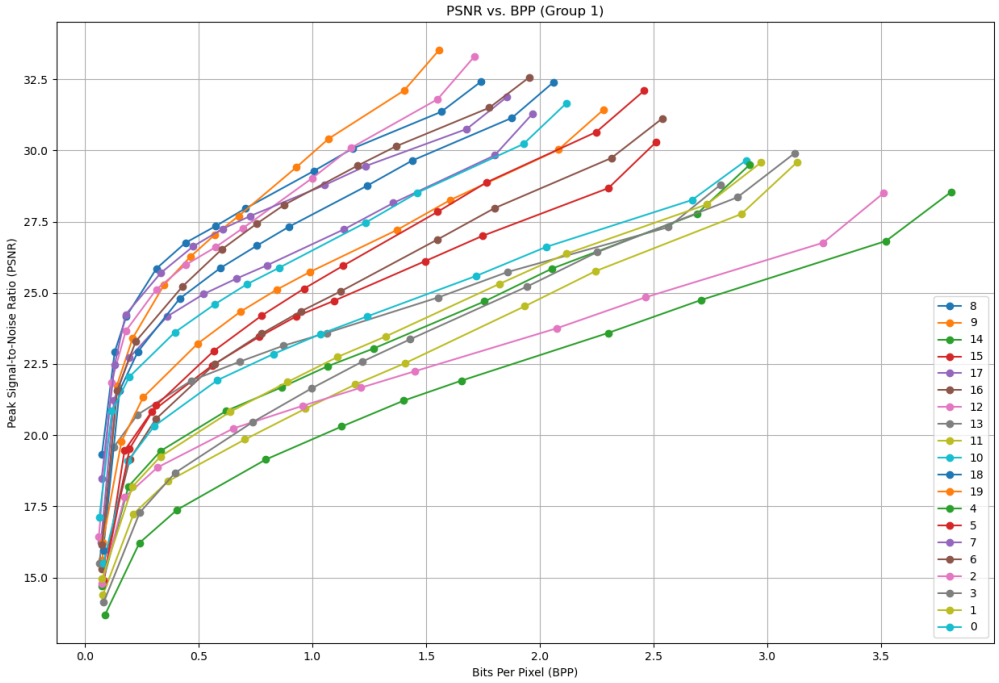


Figure 4: PSNR vs BPP plot for 20 images of size 150×150

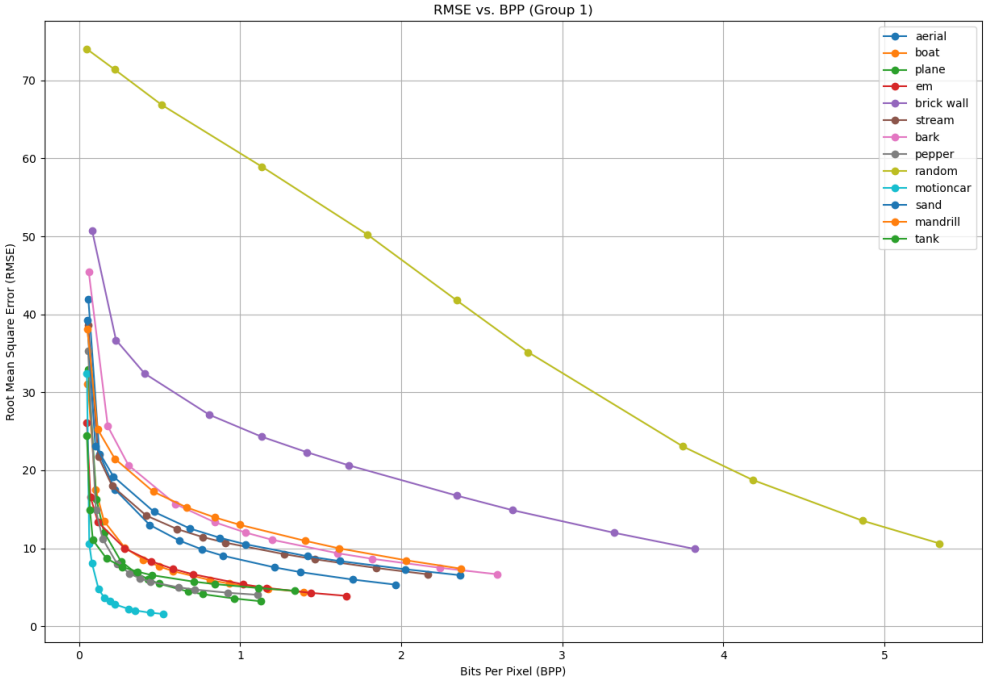


Figure 5: RSME vs BPP plot for 10 images of size 512×512

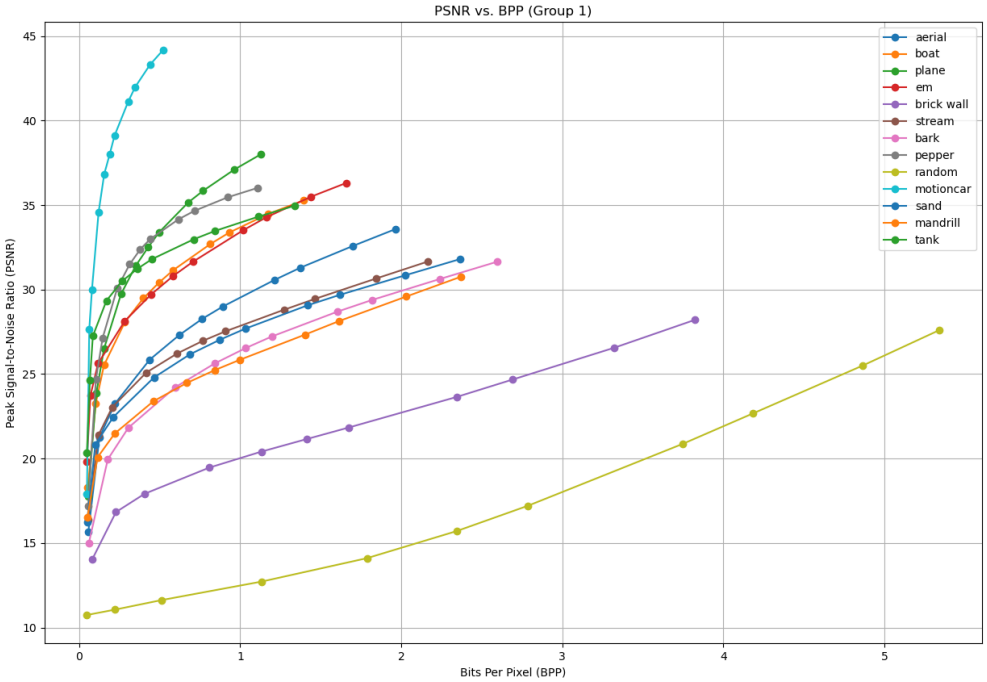


Figure 6: PSNR vs BPP plot for 20 images of size 512×512

6 Changes for Colour image compression

The primary difference between colour and grayscale images is that colour images consist of three channels (RGB), while grayscale images have a single channel

- **RGB to YCbCr conversion:** We convert the RGB channels (which are correlated to each other) to YCbCr (luminance, Chrominance) which are de-correlated. Because quantizing correlated channels causes artifacts in compressed images
- **Downsample Cb and Cr channels:** $N \times M$ size Cb, Cr channels are downsampled to $\lceil \frac{N}{2} \rceil \times \lceil \frac{M}{2} \rceil$. This significantly reduces final compressed image size, with visual quality being intact. This is another lossy part apart from quantization
- **Two Huffman trees:** We use two trees, one for Y channel patches and other for combined Cb and Cr channel
- **Upsampling at reconstruction:** This upsamples the $\lceil \frac{N}{2} \rceil \times \lceil \frac{M}{2} \rceil$ Cb and Cr channels to $N \times M$. The actual values cannot be recovered again

7 Data set for Colour images

For evaluating our implementation, we have used images from Microsoft Copilot training dataset at [3], which has $(640, 480, 3)$ size images each



8 Results for Colour images

We have run the algorithm for 33 images each with quality factors [1, 3, 5, 10, 15, 20, 25, 40, 50, 70, 90]

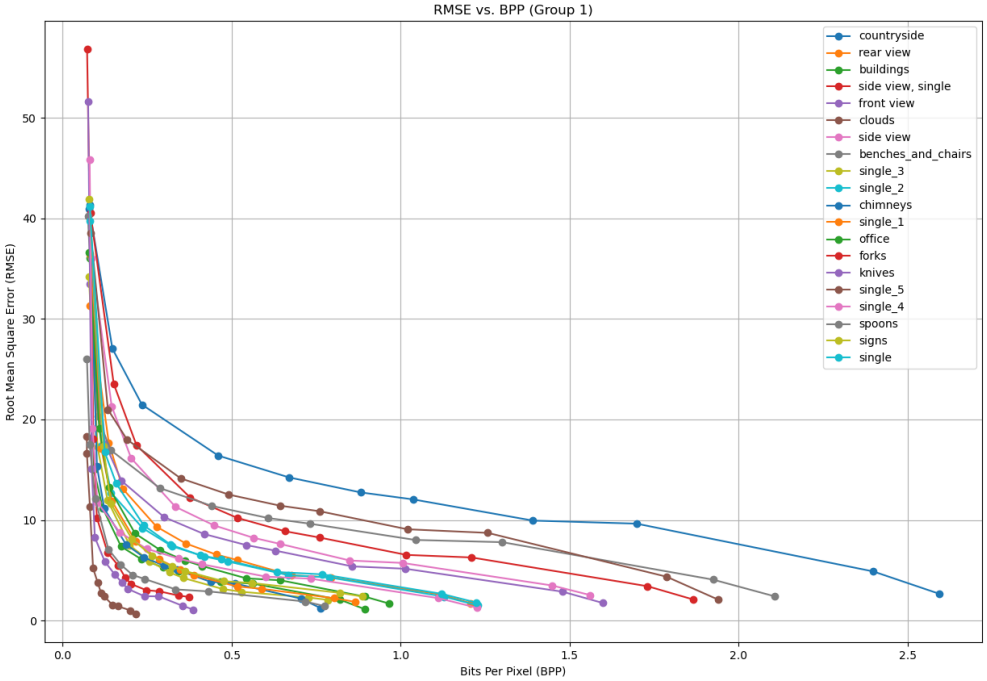


Figure 8: RSME vs BPP plot for 20 colour images

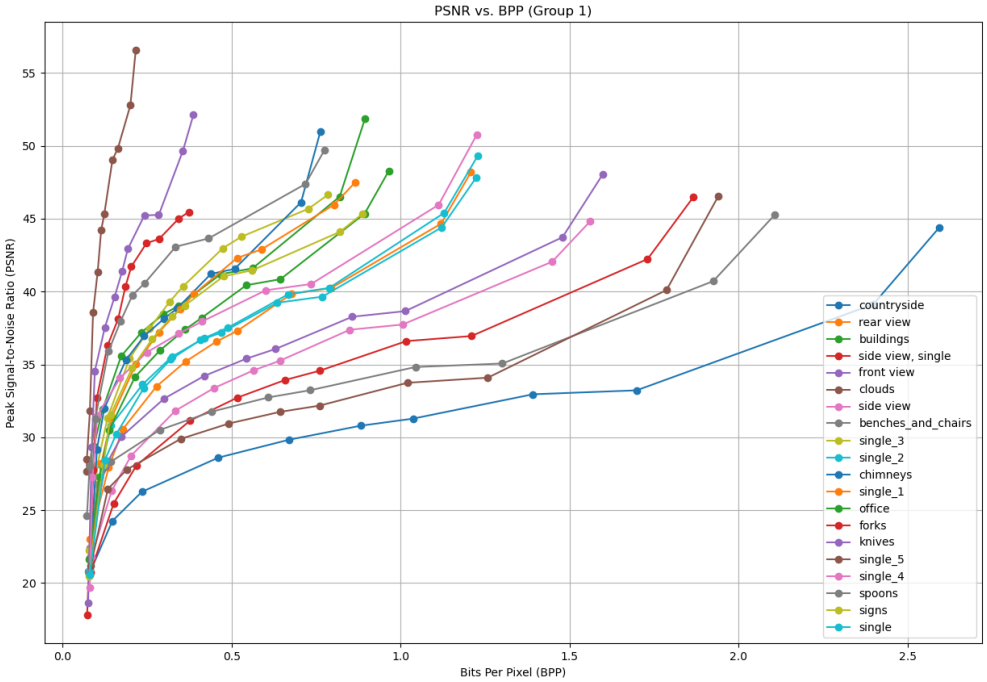


Figure 9: PSNR vs BPP plot for 20 colour images

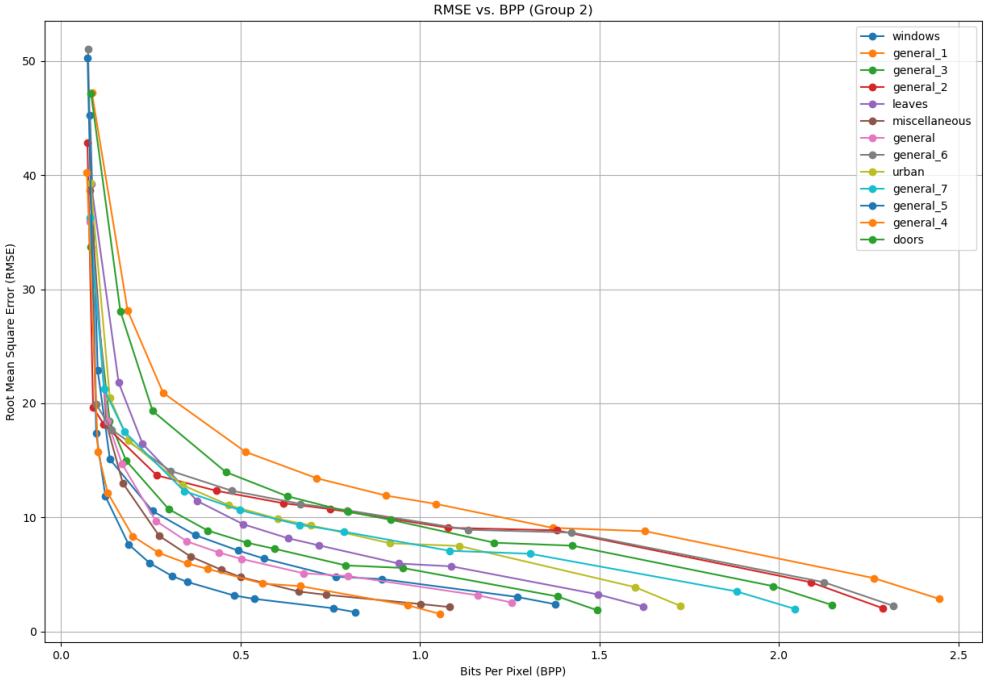


Figure 10: RSME vs BPP plot for 13 colour images

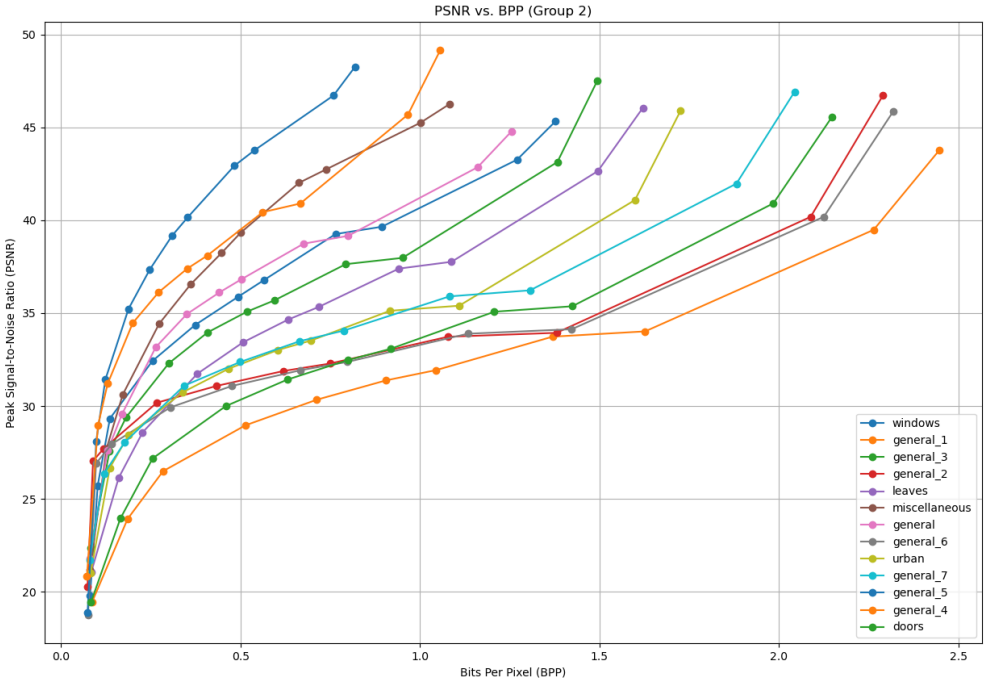
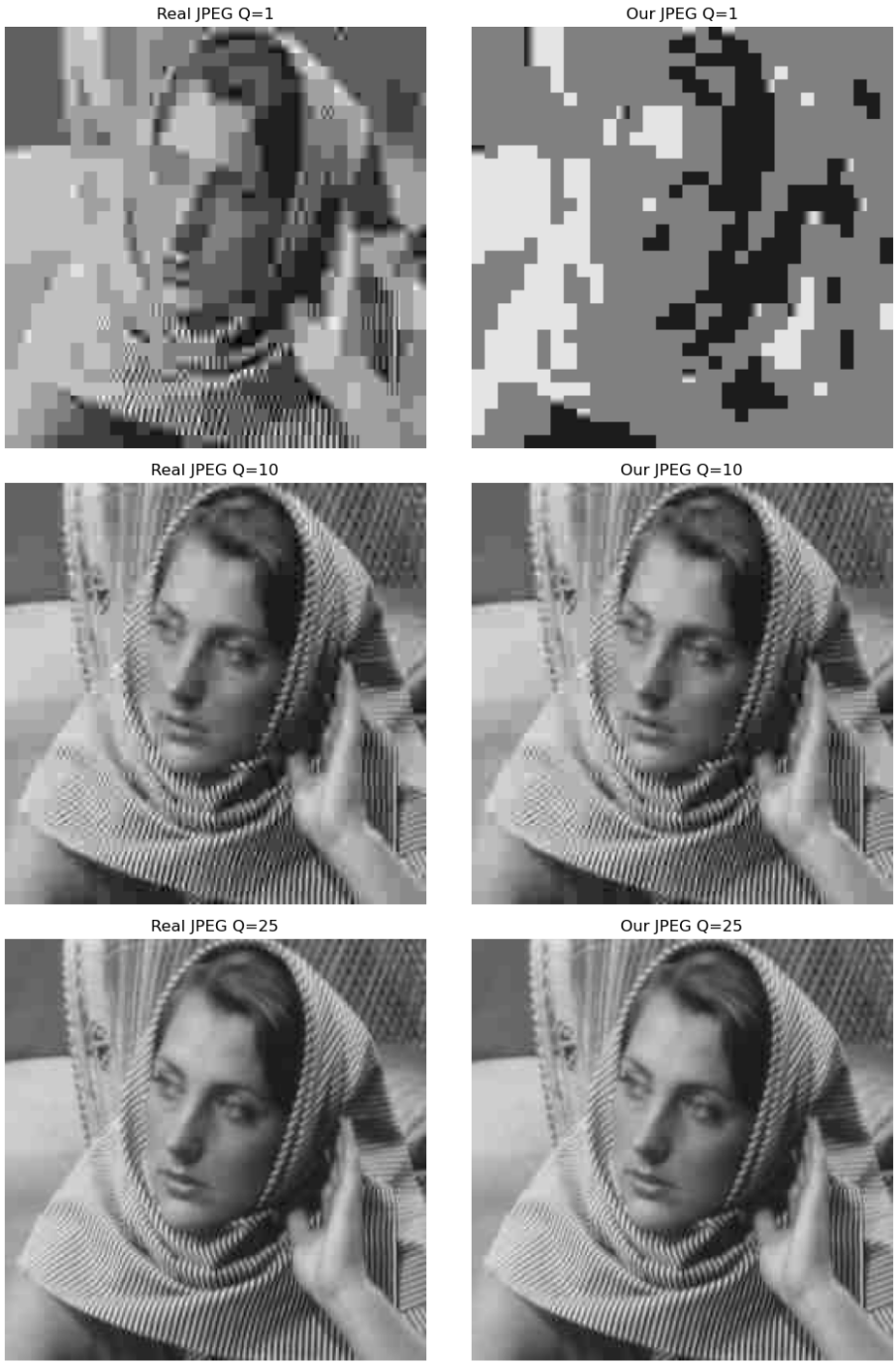
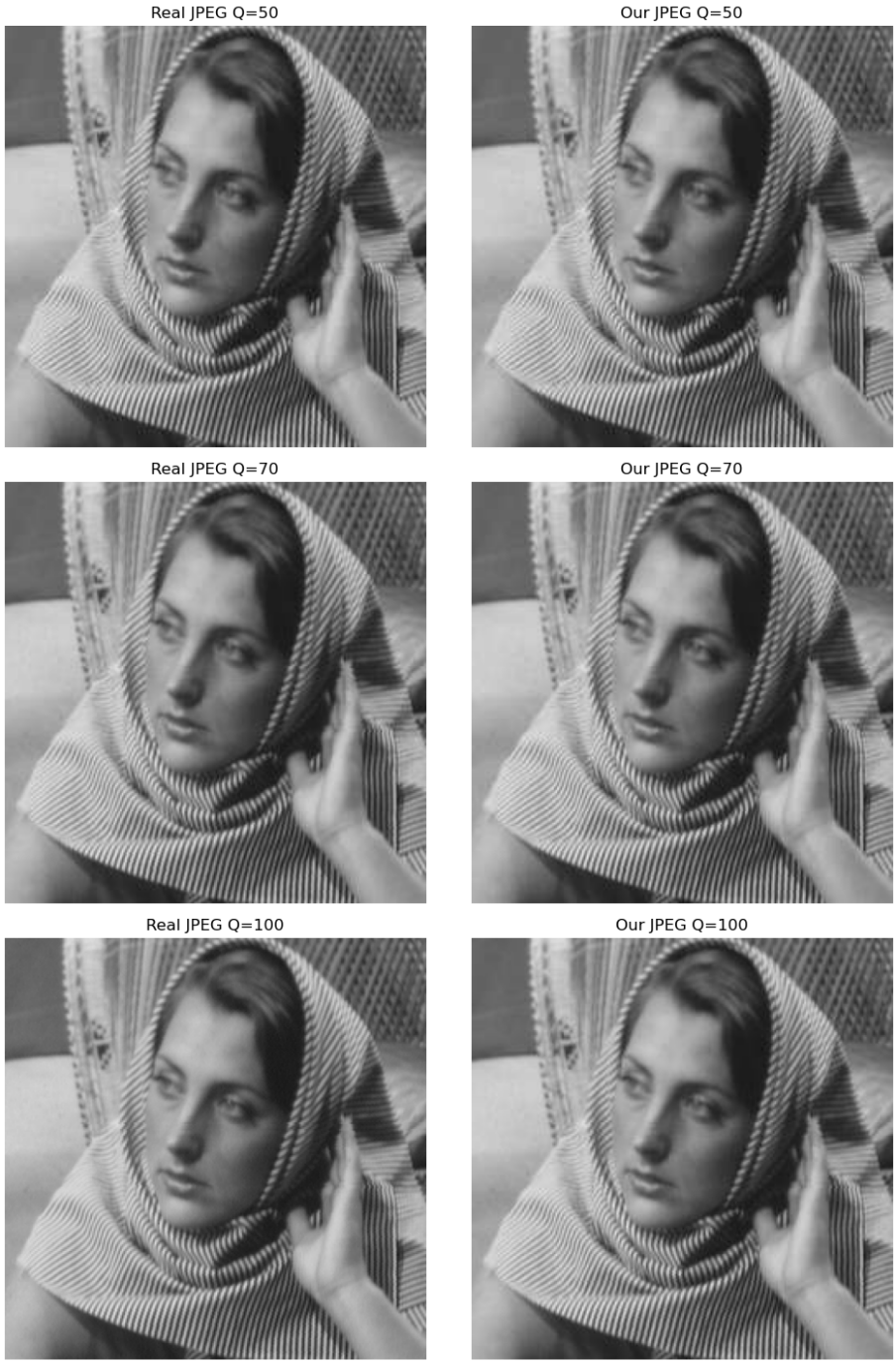


Figure 11: PSNR vs BPP plot for 13 colour images

9 Comparison with actual JPEG

9.1 Grayscale images - 1




Real JPEG Q=50
Our JPEG Q=50
Real JPEG Q=70
Our JPEG Q=70
Real JPEG Q=100
Our JPEG Q=100

Quality	Real JPEG	Our JPEG	RSME Real	RSME Our	PSNR Real	PSNR Our
1	41.43	139.74	20.442	38.753	21.920	16.365
3	38.71	69.50	19.519	20.690	22.322	21.816
5	31.46	42.23	16.354	16.795	23.859	23.627
10	20.15	21.66	12.357	12.447	26.293	26.230
15	15.49	15.41	10.134	10.194	28.015	27.964
20	13.05	12.44	8.678	8.725	29.362	29.315
25	11.50	10.76	7.698	7.775	30.403	30.316
40	8.95	7.98	6.003	6.053	32.563	32.492
50	7.91	7.08	5.357	5.448	33.552	33.407
70	6.05	5.76	4.114	4.560	35.845	34.952
90	3.46	4.99	2.363	4.013	40.663	36.061

Table 1: Compression rates of Real JPEG and Our JPEG for Various Qualities

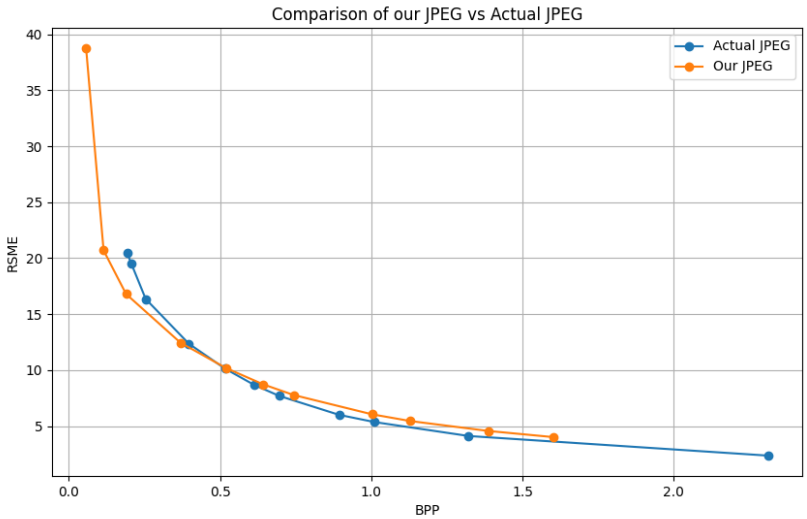


Figure 12: Barbara image RMSE vs BPP plot

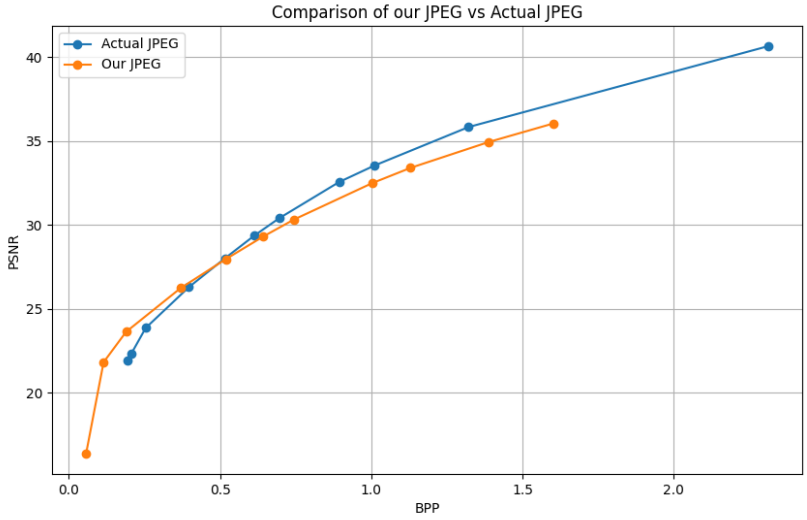
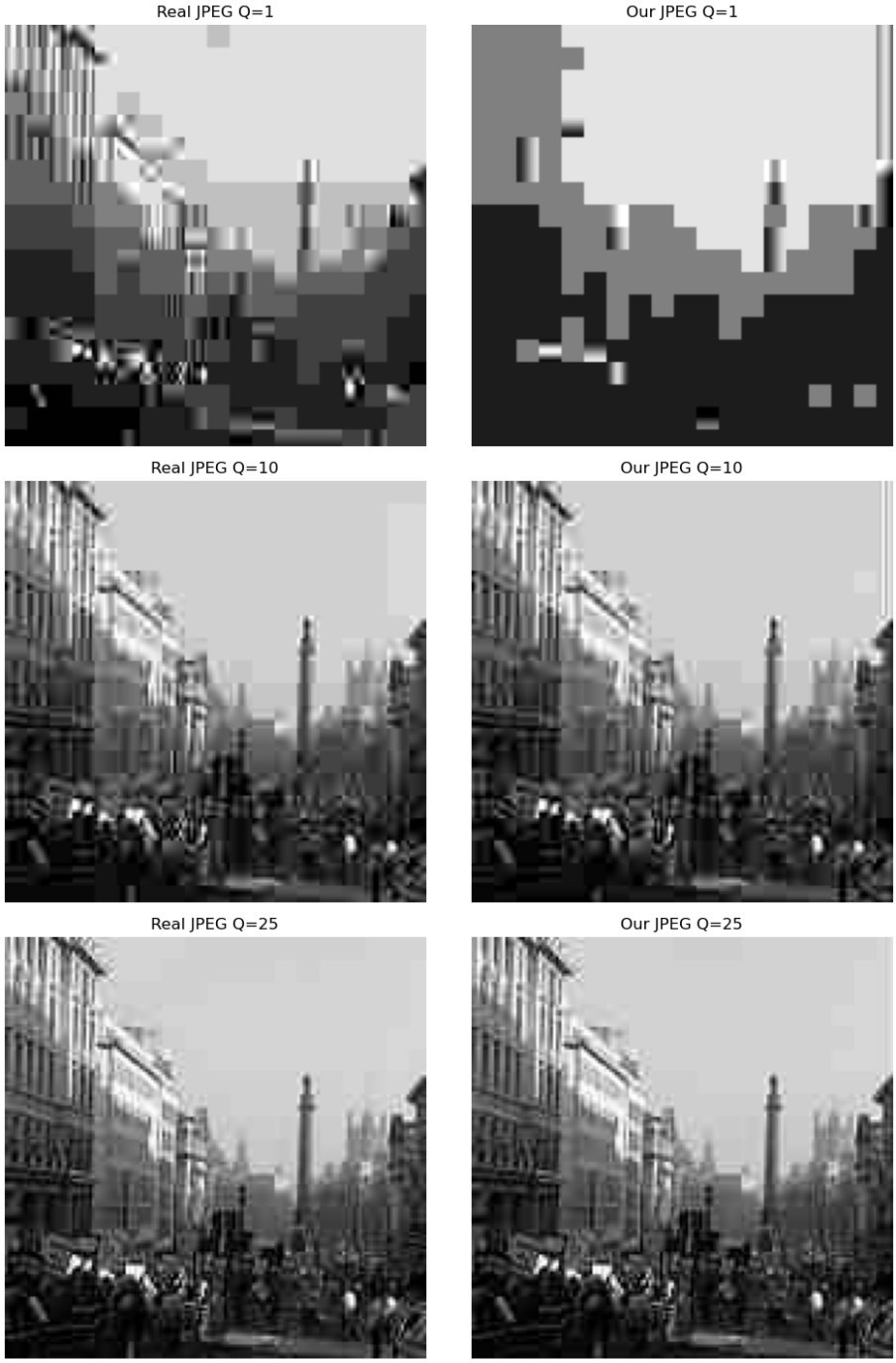


Figure 13: Barbara image PSNR vs BPP plot

9.2 Grayscale images - 2




Real JPEG Q=50
Our JPEG Q=50
Real JPEG Q=70
Our JPEG Q=70
Real JPEG Q=100
Our JPEG Q=100

Quality	Real JPEG	Our JPEG	RSME Real	RSME Our	PSNR Real	PSNR Our
1	27.44	101.35	25.742	39.395	19.918	16.222
3	26.07	49.56	24.883	26.174	20.213	19.773
5	21.87	31.16	21.039	21.885	21.670	21.328
10	14.75	16.14	17.315	17.622	23.362	23.210
15	11.66	11.69	15.260	15.477	24.460	24.337
20	10.03	9.50	14.029	14.164	25.190	25.107
25	8.78	8.08	13.045	13.178	25.822	25.734
40	6.68	5.83	11.001	11.121	27.302	27.208
50	5.80	4.98	9.705	9.854	28.391	28.259
70	4.38	3.84	6.994	8.040	31.237	30.026
90	2.55	3.51	0.813	6.840	49.926	31.430

Table 2: Compression rates of Real JPEG and Our JPEG for Various Qualities

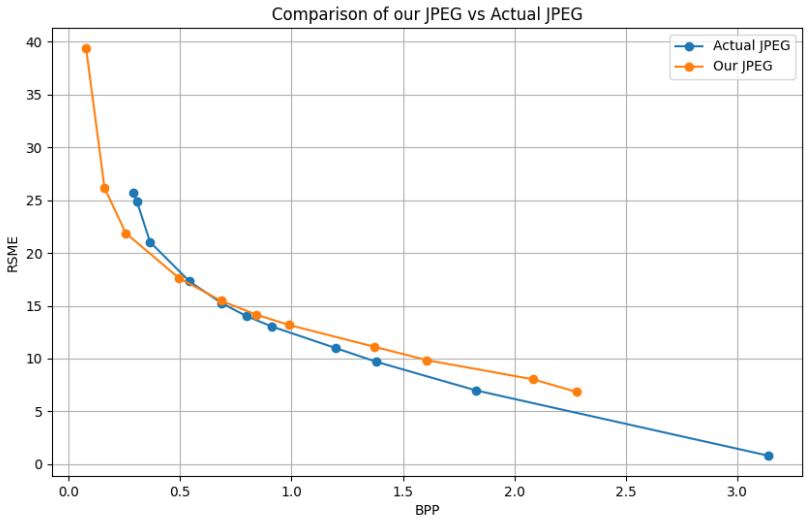


Figure 14: Grayscale image RMSE vs BPP plot

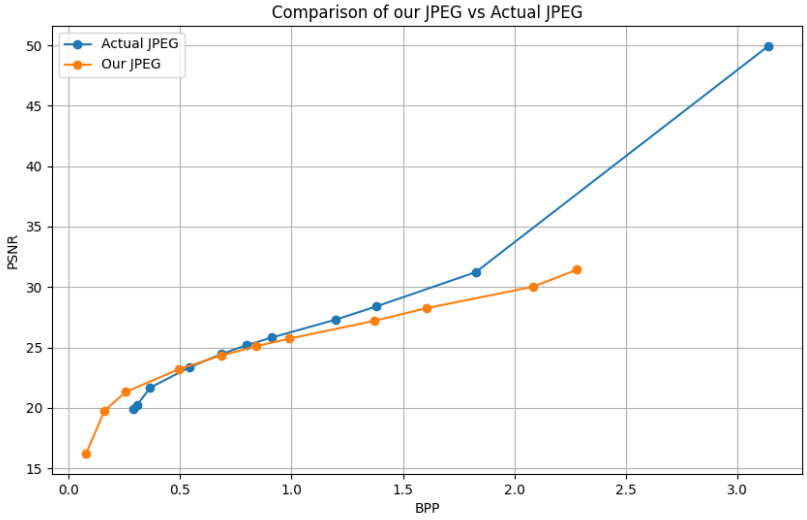
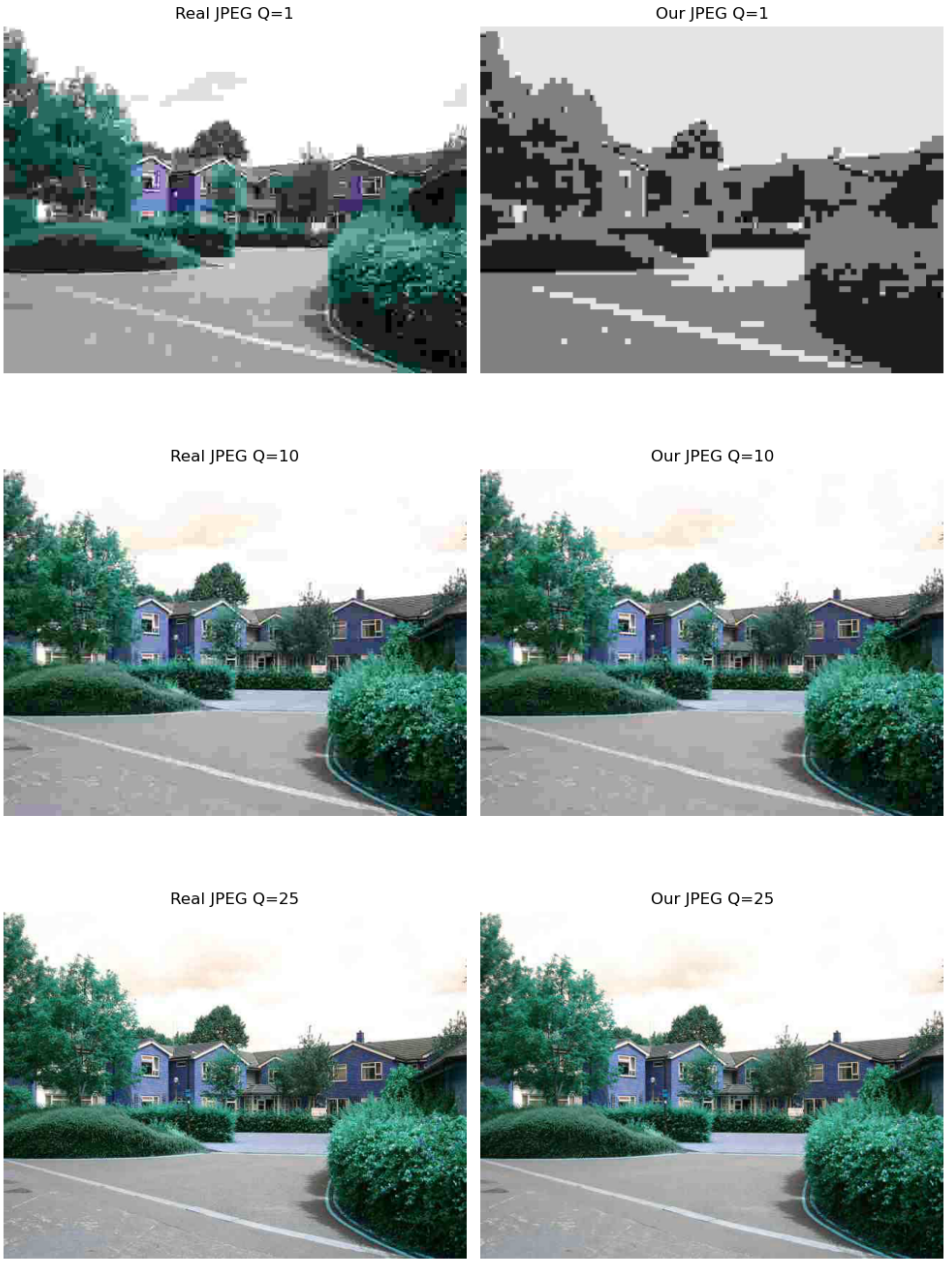


Figure 15: Grayscale image PSNR vs BPP plot

9.3 Colour images



Real JPEG Q=50

Our JPEG Q=50

Real JPEG Q=70

Our JPEG Q=70

Real JPEG Q=100

Our JPEG Q=100

Quality	Real JPEG	Our JPEG	RSME Real	RSME Our	PSNR Real	PSNR Our
1	138.42	291.37	20.784	39.274	26.547	21.020
3	129.67	176.59	20.047	20.419	26.861	26.701
5	105.18	128.00	16.792	16.678	28.400	28.459
10	69.15	71.27	12.800	12.744	30.758	30.796
15	55.71	51.91	11.069	11.002	32.020	32.073
20	45.54	39.97	10.000	9.875	32.902	33.011
25	37.44	34.65	9.453	9.299	33.390	33.533
40	31.53	26.27	7.943	7.730	34.902	35.139
50	21.60	21.73	7.624	7.495	35.259	35.406
70	17.17	15.03	2.214	3.859	45.998	41.173
90	11.57	13.92	1.579	2.195	48.935	46.075

Table 3: Compression rates of Real JPEG and Our JPEG for Various Qualities

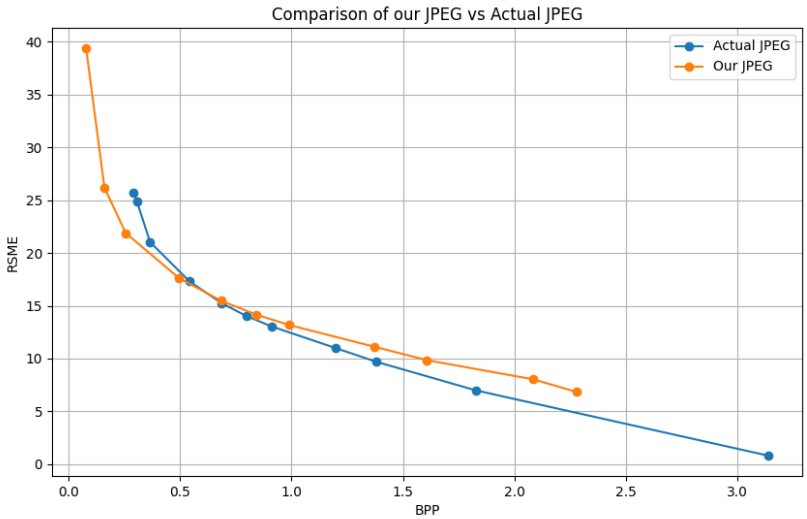


Figure 16: Colour image RMSE vs BPP plot

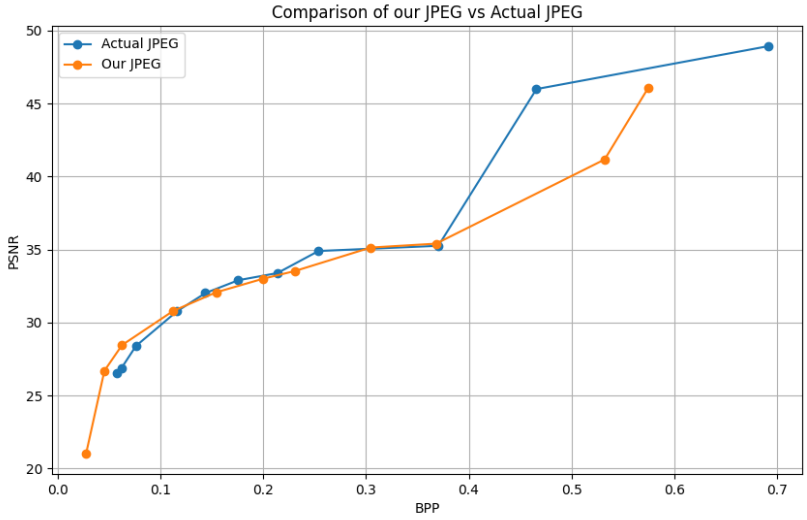


Figure 17: Colour image PSNR vs BPP plot

10 Some extensive Experiments on JPEG

10.1 Experiment 1

For colour images, we changed the quantization matrix only for luminance as below. This penalizes higher frequencies more

$$new\_luminance\_quant\_matrix_{50} = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

(a) Old Quantization matrix (b) New Quantization matrix

We experimented with quality rate 60. The compression rate for old matrix is **21.33** and with old quantization matrix is **42.15**.
But the RSME for old quantization matrix is **3.422** and new quantization matrix is **6.039**

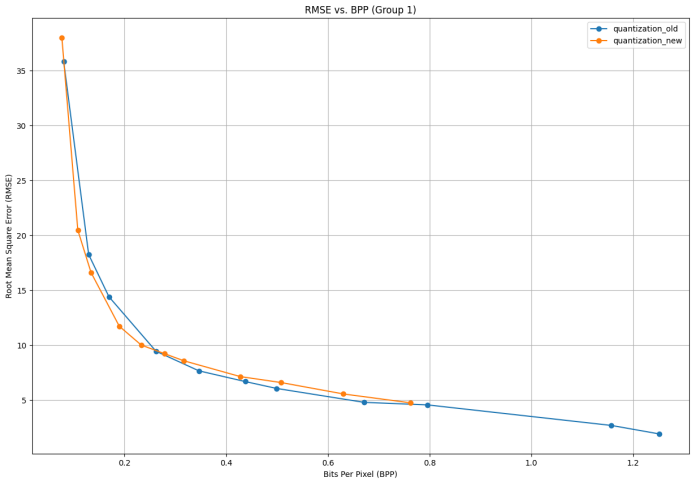


Figure 19: RSME vs BPP plot

10.2 Experiment 2

In runlength encoding if each tuple (a,b) where a represents number of zeros before it is restricted to 4 bits versus not restricting it, the second case is giving better compression. In class we did by restricting it to 4 bits and we did not and got better results
Reason: If an patch has many zeros, storing it in many (15,0) tuples is not a good option as mostly all zeroes will be at end and directly keeping (0,0) is saving lot of space

Example compression rate for colour image: **56.15 without restricting, 33.86 with restricting original JPEG:65.38**
Example compression rate for grayscale images: **5.08 without restricting, 4.58 with restricting original JPEG:5.92**
These are done for quality 30

10.3 Experiment 3

Instead of storing 2 separate trees for DC and AC, if we merge them then compression is slightly decreasing for example from **14.65 to 13.65** making separate trees for AC and DC better
Storing 2 seperate trees might increase space for storing 2 trees but **average length of content is decreasing** and making it more affective. This shows the distribution of AC and DC coefficients is much different

10.4 Experiment 4

If We dont go in zig-zag order in each patch then the results are worse
Reason: Ziz-zag order ensures that all zeros will be end increases the compression rate
Example compression rate for colour image: **56.15 without restricting, 51.5 with restricting original JPEG:65.38**
Example compression rate for grayscale images: **5.08 without restricting, 4.74 with restricting original JPEG:5.92**

11 PCA-based compression on a class of images

We are testing the compression of ORL dataset face images using PCA and storing the eigenfaces and coefficients for each face This Dataset contains images of 40 people, 10 per person. Each image is of size 112x92, which is 10304 pixels per image. Let the number of pixels be D, and the number of images is N.

11.1 Compression mechanism

Take all the images in the dataset as columns of a matrix X. Now we have to find eigenvectors for the covariance matrix of X. Dimension of XX^T will be dxd, which is huge and highly inefficient to find covariance of this matrix. Therefore, we see the covariance of X^TX, which is NxN and find eigenvectors for this matrix. and to find the covariance matrix of XX^T, we multiply X with eigenvectors of X^TX to get top N eigenvectors of XX^T.
Now, for the compression of a set of images, we find the top K eigen faces for the set of images, k eigen coefficients for each image and the mean face.

11.2 Compression Rate

If we take k eigenvectors, the compression rate is

$$\frac{N * d * 8}{32 * d * k + 32 * N * k + 32 * d}$$

$$\frac{1}{\frac{4k}{N} + \frac{4k}{d} + \frac{4}{N}}$$

Which in our case will approximately be $\frac{N}{4k}$

In our experiment we took K in same order of N but if images are many then compression rate will also be high

In our case, we took N=400 and k ranging from 150 to 300

11.3 Results

- Case1: K = 150. Comparison of our JPEG, and original JPEG in terms of RSME and PSNR

Figure 20: k=150

The top row is the original image, the second row is the PCA compressed image, and third row is our JPEG compressed image for quality = 30
The compression rate of PCA for these images = 0.63

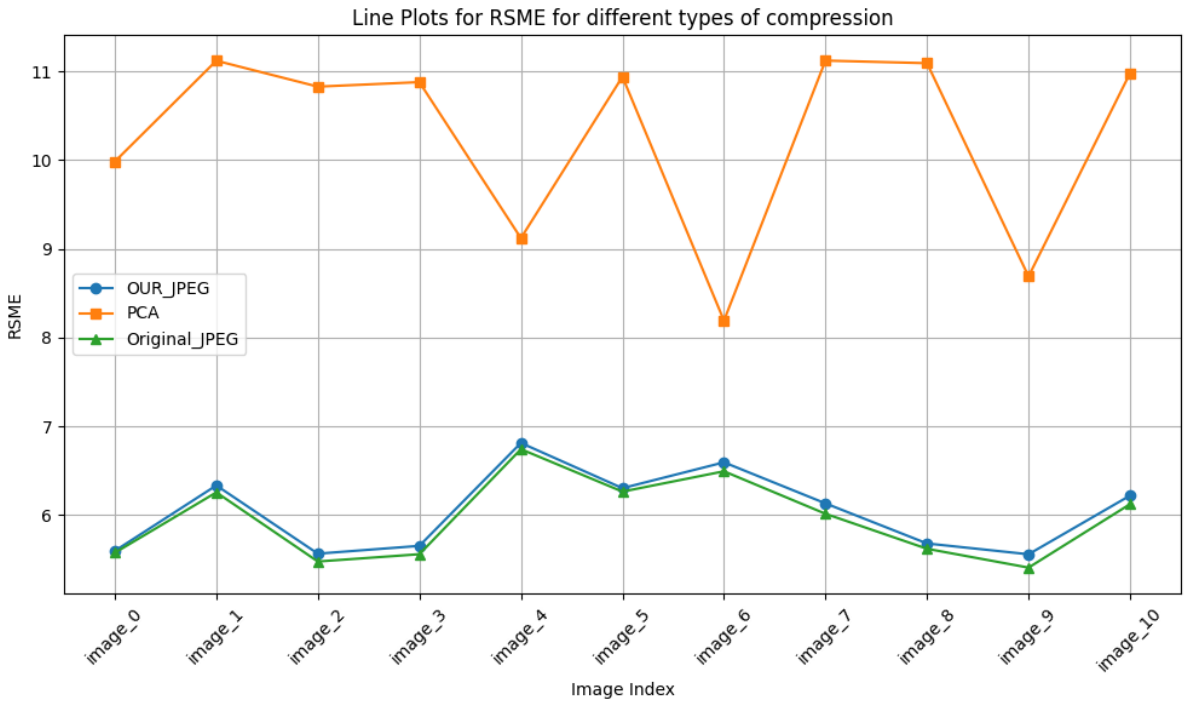


Figure 21: k=150

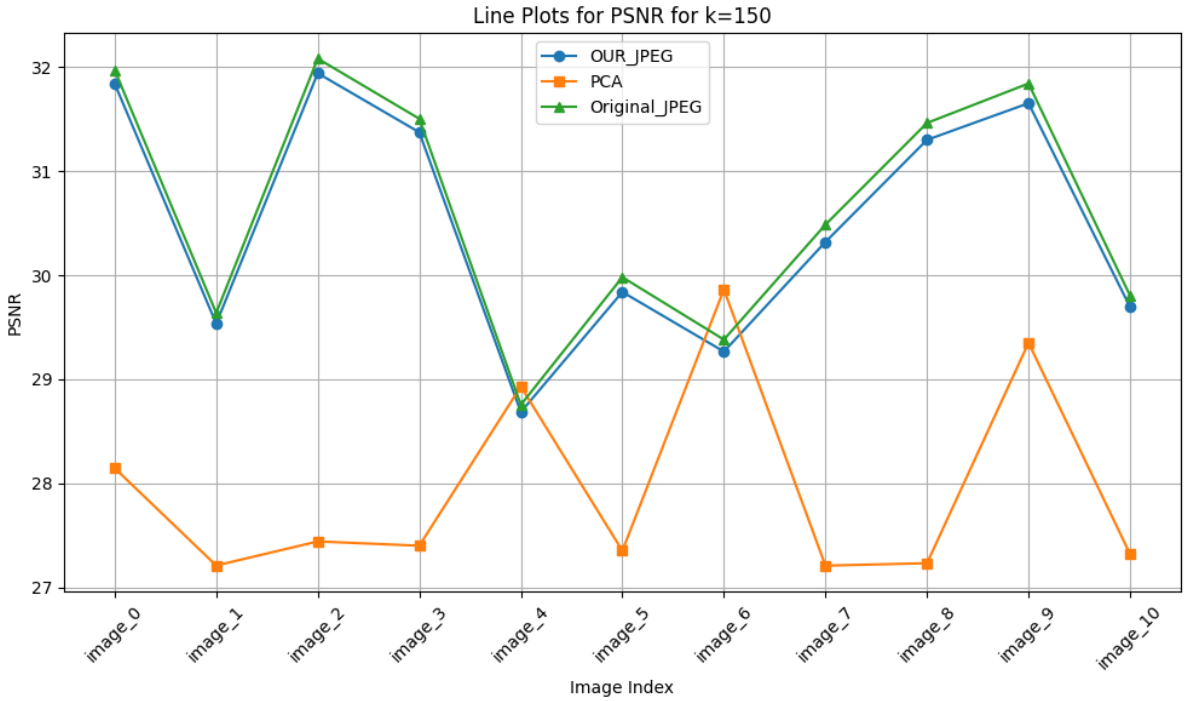


Figure 22: k=150

- Case2: K = 225. Comparison of our JPEG, and original JPEG in terms of RSME and PSNR

Figure 23: k=225

The top row is the original image, the second row is the PCA compressed image, and third row is our JPEG compressed image for quality = 30
The compression rate of PCA for these images = 0.43

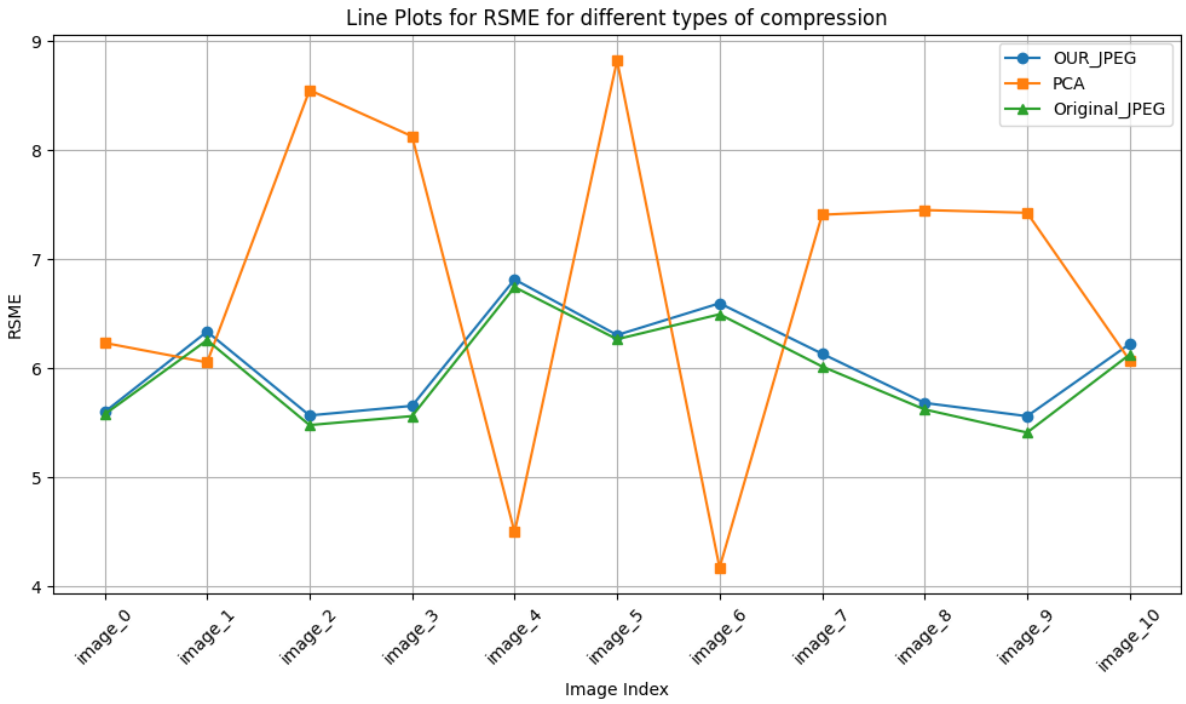


Figure 24: k=225

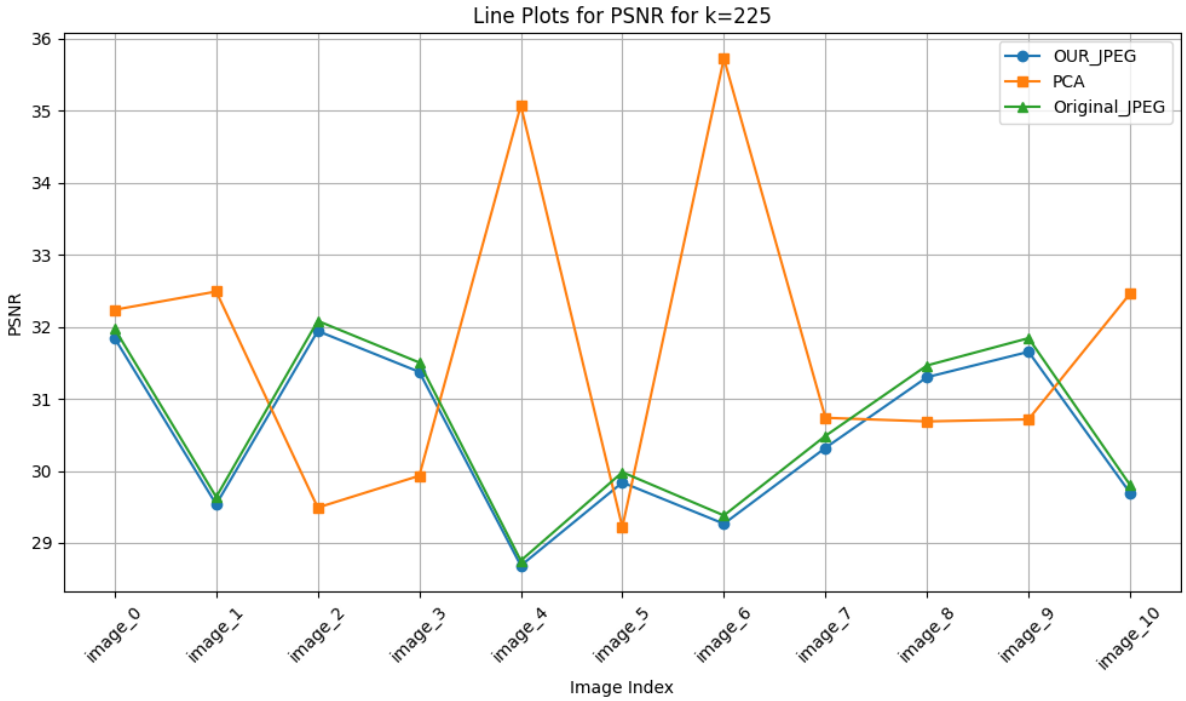


Figure 25: k=225

- Case3: K = 300. Comparison of our JPEG, and original JPEG in terms of RSME and PSNR

Figure 26: k=300

The top row is the original image, the second row is the PCA compressed image, and third row is our JPEG compressed image for quality = 30
The compression rate of PCA for these images = 0.32

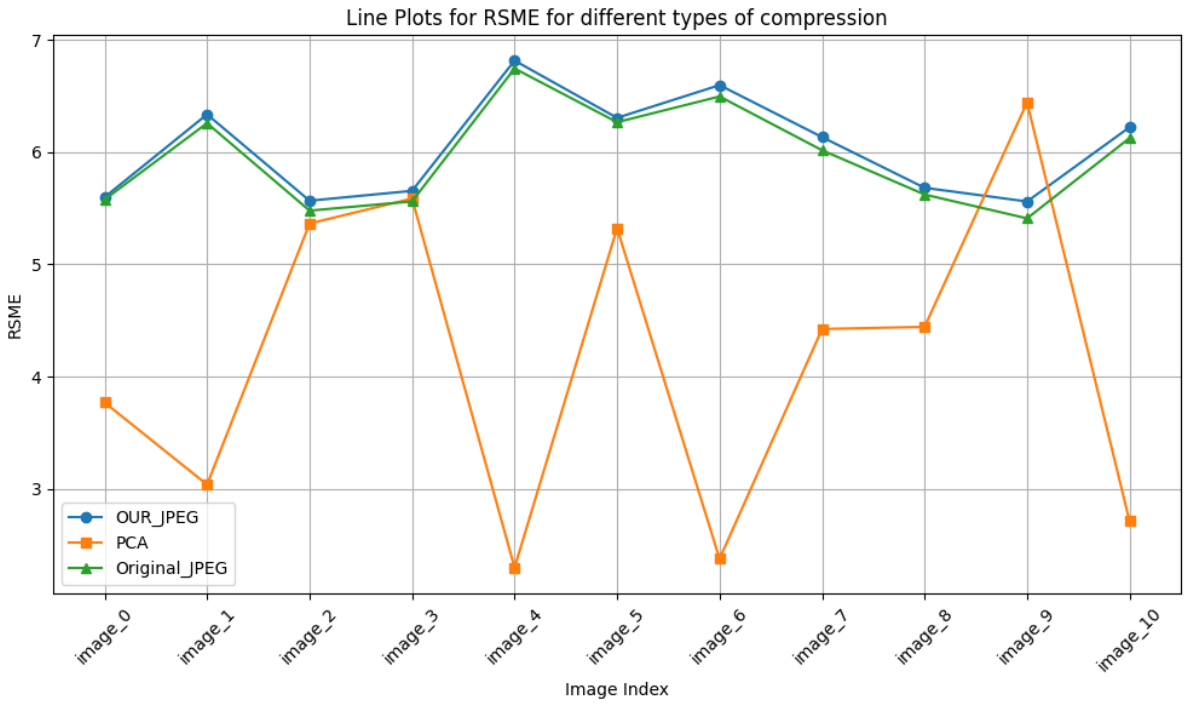


Figure 27: k=300

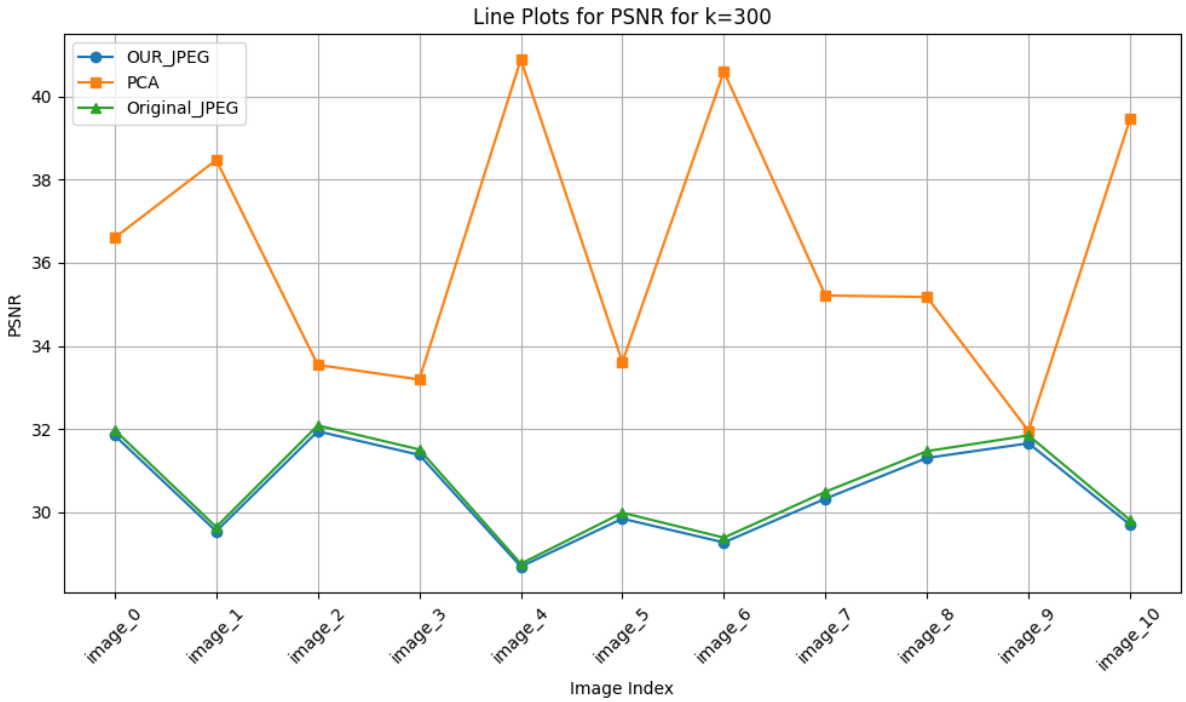


Figure 28: k=300

11.4 Inferences

PCA-based compression is better regarding compression rate if k « N. In our case, as N=400, we could not test for this case and didn't get a reasonable compression rate.
But RSME and PSNR values are better for PCA-based compression for k=225 and k=300.
In general, these parameters are better for PCA-based compression than for JPEG.

12 Implementing Paper : Edge-Based Image Compression with Homogeneous Diffusion

Introduction

This paper describes a lossy compression method based on the fact that edges contains semantically important information of images.We explicitly used this method for cartoon-like images where we observe that storing information of pixels near edges and location of edges is enough to generate images of high quality.The unstored pixels using steady state of homogeneous diffusion process

Compression Algorithm

The main steps of Compression Algorithm

- **Detecting Edges:** First we will detect the edges of an image used the **Canny Edge** detector and stored this bit-image in a image
- **Storing the mask:** The bi-level edge image(means that there are only 2 different intensities in the image) obtained is compressed using a method called **JBIG**(Joint Bi-Level Image Expert Group).We use a lossless version in our case.
- **Encoding the pixels:** We store the intensities of pixels present adjacent to where edges are there.We take a window of the edges(We used a window size of 1) around the edge pixel and store only the intensities of these values.We store all the values in a `jpg` file.
- **Subsampling:** We can further only store a some pixels around the edge instead of storing all pixels which are present along the edge since the intensity difference between neighbouring pixels is small(definition of edge) and we can further quantize the intensity values to a set of 2^q values.
- **Creating a archive:** We compress the `jbg` file (obtained from **JBIG** compression) and `jpg` file we created for storing neighoburs of edges to a single archive using a lossless compression method called **ZPAQ**.

Decompression Algorithm

The main steps of Decompression Algorithm

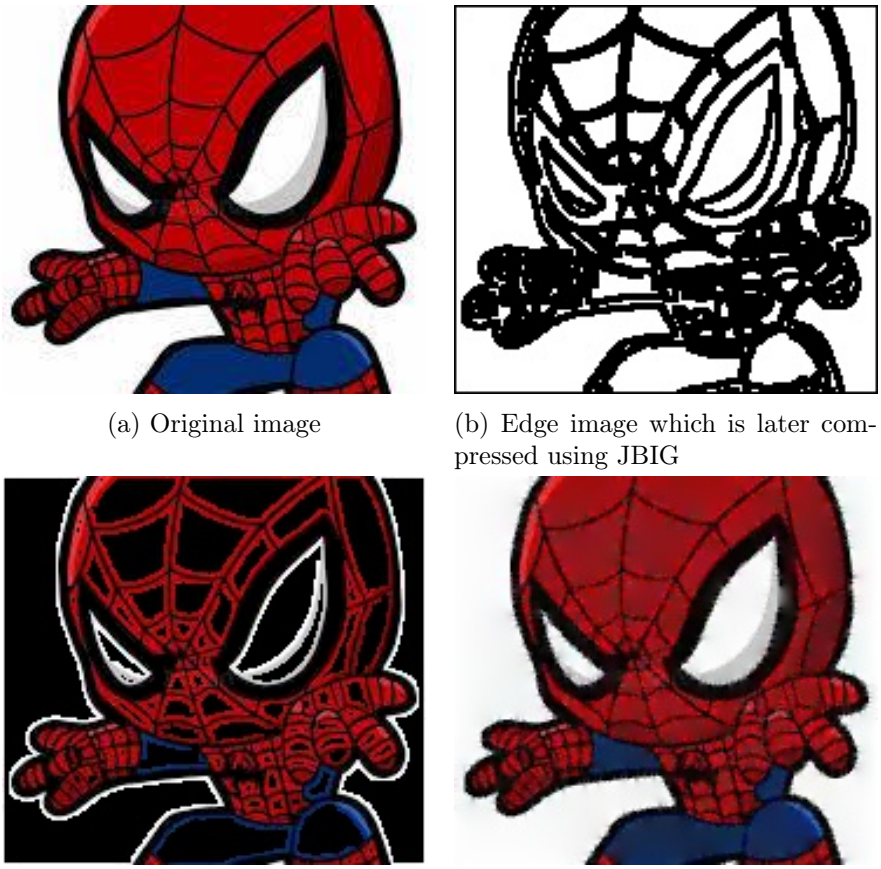
- **Supersampling:** We first reconstruct the actual intensities of quantised values present in image and reconstructed the missing from the stored values using a linear interpolation along th edge.
- **Reconstructing the image:** Now that we have intensities of image along intensities in the neighbourhood of edge,we compute the rest of the missing values using image in-painting.We use homogeneous diffusion for interpolation.

  $$image+ = \delta \times mask \times \texttt{Laplacian}(image)$$

  Above step is run for large number of iterations (depends on value of δ) and we get the resultant image($\delta = 1500/\texttt{num\_of\_iterations}$)

Implementation and Results

The following are the images obtained in various steps of our algorithm as mentioned.



(a) Original image

(b) Edge image which is later compressed using JBIG

(c) Colors next to edges and image boundary

(d) Image reconstructed using Homogeneous Diffusion

Figure 29: Images and masks obtained from `spider.jpeg`

Below is the table containing Compression ratios achieved by our algorithm and by our implementation of JPEG and corresponding images obtained.

Image	Compression Ratio by our JPEG	Compression Ratio by Edge based
`im1.png`	16.04	3.60
`im2.png`	40.58	9.93
`im3.png`	40.09	6.19
`spider.jpeg`	47.05	11.53

Table 4: Compression ratios obtained for various images

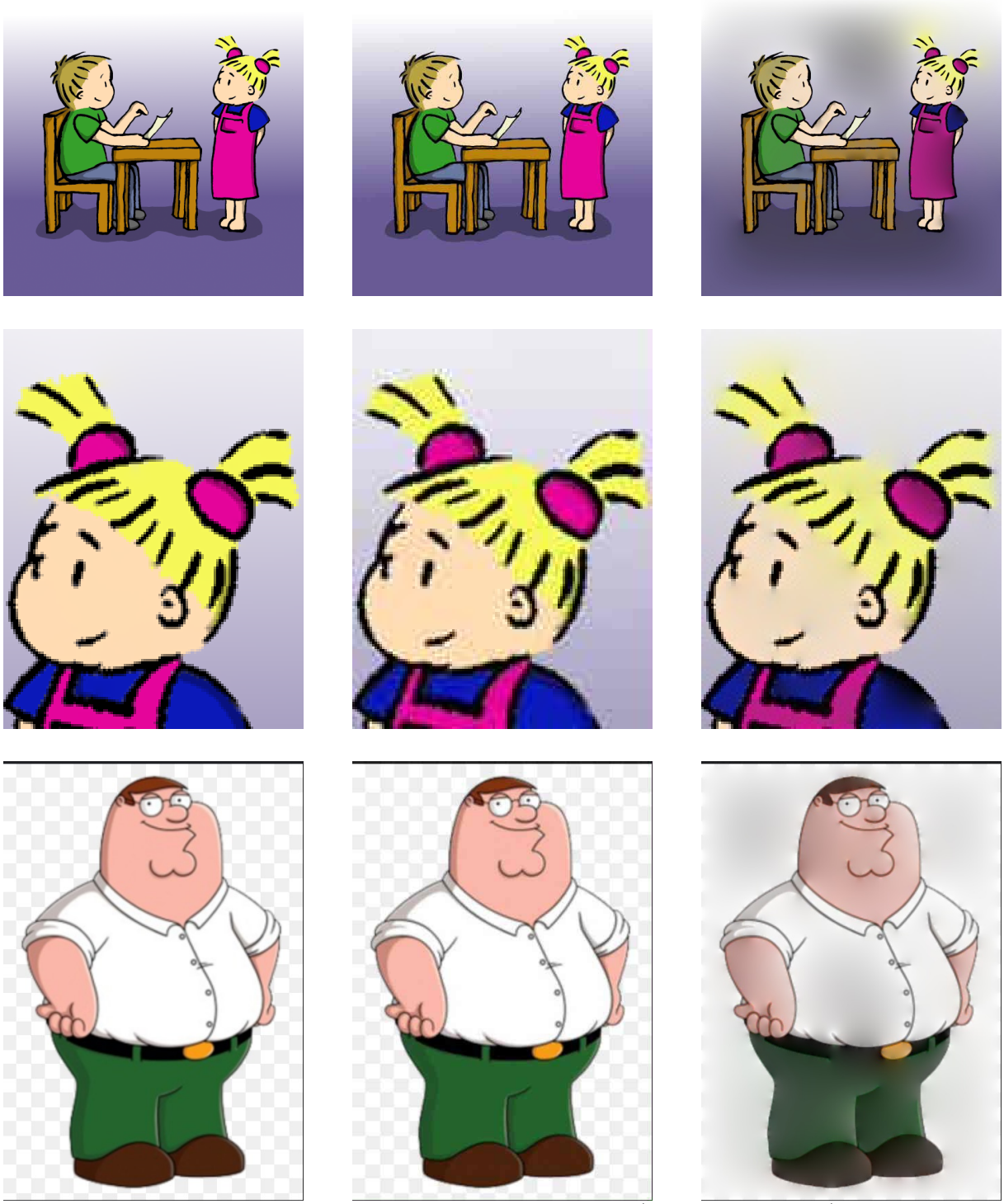


Figure 30: Images are there in following order in each row a)Original Image b)Restored image after our JPEG compression c)Restored image using the edge based compression

Conclusion

As you can see in the images as well as by our compression ratios,the images obtained by our edge based compression is not too good.By increasing the number of iterations or changing the value of delta in the decompressing part,some images got worsened while some remained the same and these are without using subsampling and supersampling. We were not able to include subsampling and supersampling due to various other constraints(mainly time and lack of quality of obtained images).By including we might improve Compression ratios but quality of image doesn't improve)

More about JBIG and ZPAG

JBIG (Joint Bi-level Image Experts Group) is a lossless image compression standard for binary images. **ZPAQ** is an open-source command-line archiver and file compression format that allows users to create compressed archives and encrypted backups.ZPAQ uses a variety of algorithms, including LZ77, BWT, and context mixing, to compress files.We used a lossless variation of **ZPAQ** and **ZPAG** archives will have smaller space than other formats be like **ZIP** and **RAR**.

12.1 Data Set Drive links

- Graysacle images : `https://iitbacin-my.sharepoint.com/:f:/g/personal/22b0920_iitb_ac_in/Ei3iM_rCuz1MmW4My11rcnwB38_v_9hXo8YeO3HmhO1KKw?e=TCe10D`
- ORL data set : `https://iitbacin-my.sharepoint.com/:f:/g/personal/22b0920_iitb_ac_in/EjWnVhjHVPxOj8fspbsGB_QBiuTCiEPKsBiWqpgkkxzmzQ?e=YI9uZI`
- Colour images used : `https://iitbacin-my.sharepoint.com/:f:/g/personal/22b0920_iitb_ac_in/EsRcvKw_OwZAvjBHjRTPvzkBJSJX-79eUSrMVCYczqALAQ?e=dQxDwt`
- Cartoon images : `https://iitbacin-my.sharepoint.com/:f:/g/personal/22b0920_iitb_ac_in/ElQvi8yErURMmUNtveh96p4BPCLMdd0fCVeAsBSXQW8RIg?e=jMdngg`

13 Contribution of teammates

All of them contributed equally to project and helped each other in every subpart
Main parts done by each teammates

- **Nandan Manjunath:** Lossless compression in JPEG, PCA based compression, report.
- **Varshit:** Lossy compression part of JPEG (both for grayscale and colour), report, Extensive experiments.
- **Lohith:** Research paper implementation, Extensive experiments, report.

14 How to Run (Running instructions)

14.1 Basic JPEG implementation

- Add image to **data** directory
- Open **test.py** file
- Based on grayscale/colour image add path at test_gray/test_color function
- Change **quality** variable as required
- Run **python3 test.py** to get comparison plots

14.2 PCA based compression

- Add **ORL** data into **data/ORL** directory
- Open **test.py** file
- Add mutiples of 10 indices (any 3) to be compared from ORL data
- Uncomment **line 7** to get plots of image comparision
- Uncomment **line 8** to get plots of rsme, psne comparision
- Run "python3 test.py"

14.3 Edge Based Compression

- Add image to **data** directory
- Open **test.py** file
- Add path to the variable **original_image_path**
- Run **python3 test.py** to get compressed image at directory **reconstructed_images**

References

[1] Jeffrey Bush. *C4L Image Dataset*. https://github.com/coderforlife/c4l-image-dataset. 2020.

[2] Bhabuk Ghimire. *Landscape color and grayscale images*. https://www.kaggle.com/datasets/theblackmamba31/landscape-image-colorization. 2020.

[3] *Microsoft Research Cambridge Object Recognition Image Database*. https://www.microsoft.com/en-us/download/details.aspx?id=52644. 2016.