Report on Image Compression using JPEG Algorithm

Problem Statement:

This project involves designing and implementing an image compression engine inspired by the JPEG algorithm. The objective is to compress grayscale images and evaluate the algorithm's performance in terms of compression quality and efficiency. The implementation is expected to follow the key steps of the JPEG compression process, achieving meaningful results and demonstrating practical application.

Description of the algorithm implemented for a single image:

Encoding:

Read the grayscale image and appropriately zero pad the images such that the image can be split into perfect 8x8 blocks.

For each block, calculate the DCT and elementwise divide with the quantization matrix, where the quantization matrix is created by taking the standard quantization matrix and elementwise multiply with 50/Q.

With the quantized blocks, create an array of values in a zig-zag pattern for each block.

Split the DC components separately, and AC components are run length encoded for each block separately. Create the Differential Pulse-code modulation for the DC components.

With the encoded DC and AC components, find the Huffman coding, for which we need the frequency of each component. Encode the DC and AC components using the calculated Huffman code.

Store height, width, number of channels, row padding, column padding, quantization table, size of the Huffman tables, encoded values size, and the Huffman table and encoded data as bit values details in a file.

Decoding

Read the metadata, Huffman table, and encoded values for the data.

Huffman decode the AC and the DC components. With the decoded values, create back the DC component and de-zigzag the AC components.

Then, with these values, recreate each block and multiply elementwise with the quantization table used in the encoder, performing inverse DCT on each block.

Description of the dataset:

We have used grayscale images from Kaggle on different file sizes with different pixel heights and width details. There are 20 images named from “image_01” to “image20”. These images are unrelated to each other in the context of the objects inside.

Sample Reconstructed images:

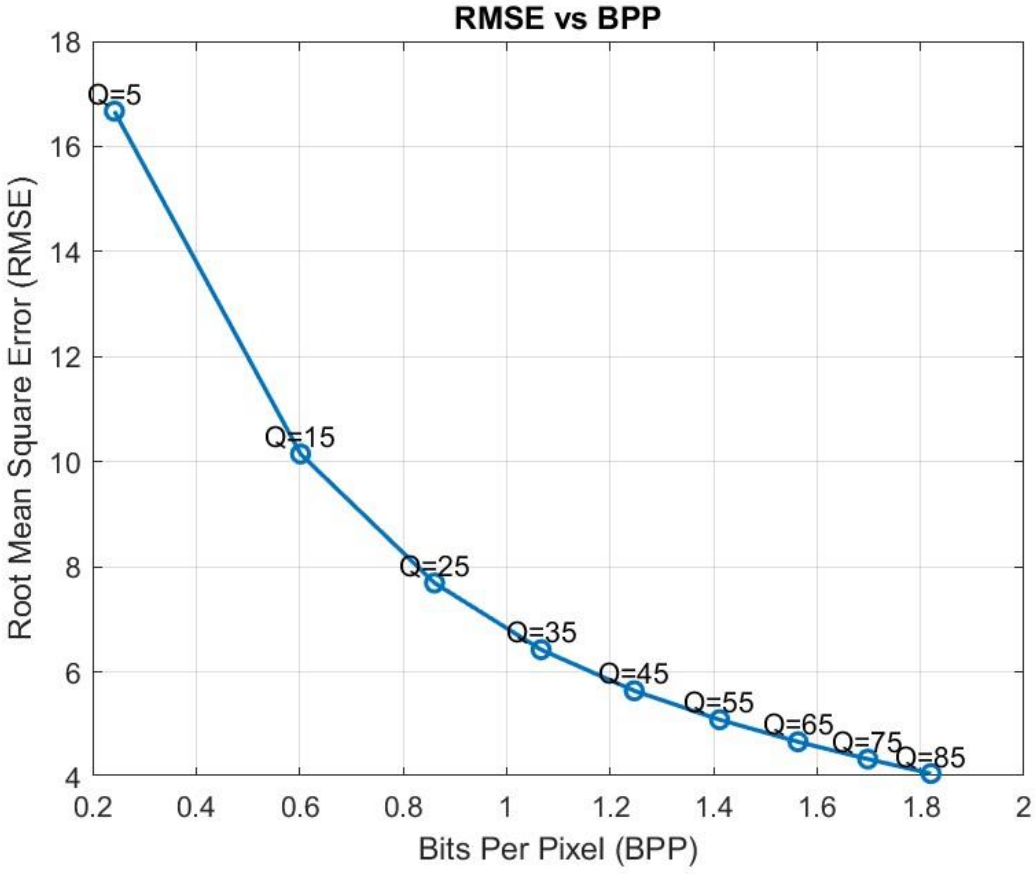
Q=5

Q=55

Q=85

RMSE vs BPP:

For each of the above images, we find the RMSE vs BPP plots with varying Q values. Below is the RMSE vs BPP plot created for the Barbara image.

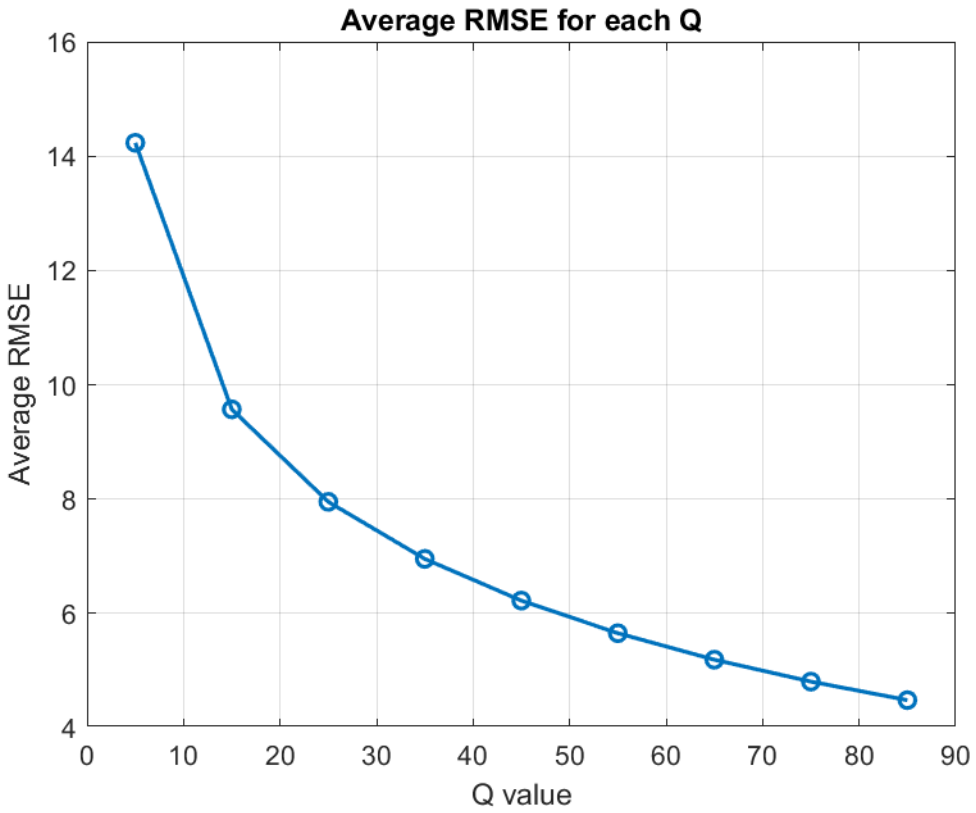


The graph plots the BPP values in the X-axis and RMSE values in the Y-axis for Barbara image each Q value. (for the rest of the images the please check the github).

As we have expected for the heavier compression that is the Q value is less, the BPP is less but the RMSE is very high, similarly when increasing the Q value BPP increases as expected as we storing more data per pixels thus reducing the RMSE.

Average RMSE value for each Q value:

For each of the test image we found the Average RMSE value for different Q values and the plot is shown below.

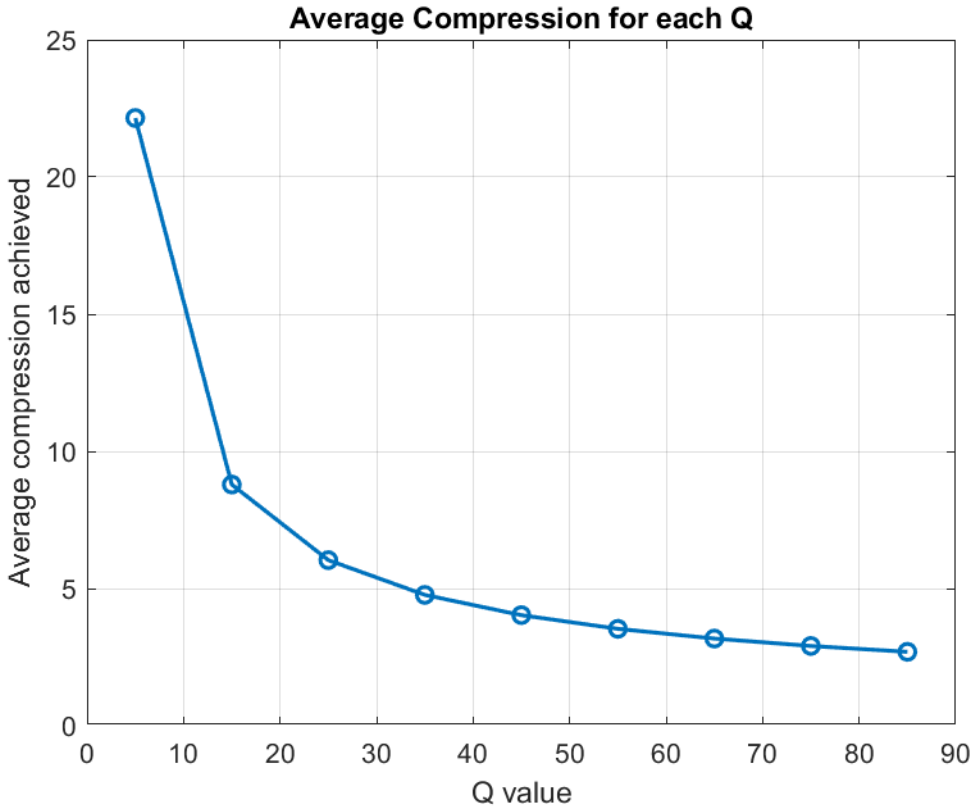


As we have expected, the RMSE value goes down when the Q value is increased because with the increasing Q value, we are giving less weight to the quantization.

Average Compression Ratio for each Q value:

For each of the test image we found the Average compression ratio for different Q values and the plot is shown below.

As we have expected, the compression ratio goes down when the Q value is increased because with the increasing Q value, we are storing more data of the image.



Color image processing:

Description of the algorithm implemented for a single image encoder:

Encoding:

Convert the RGB image to YCbCr layers, then apply the greyscale compression for each YCbCr layer with the functionality we have created above.

After compression, similarly store the compressed encoded data in a file.

Decoding:

Read all the three YCbCr layers that we have stored and decode/decompress using the functionality we have created above.

After decoding/decompression, convert the YCbCr layers to the RGB image.

Description of the dataset:

We have used some famous color images from Kaggle, which contains 17 images of foxes, each labeled as "img01" to "img17". Those images include foxes of various colors, such as red foxes, arctic foxes, and other variations of fox species, providing diversity in texture and colors.

Sample Reconstructed images:

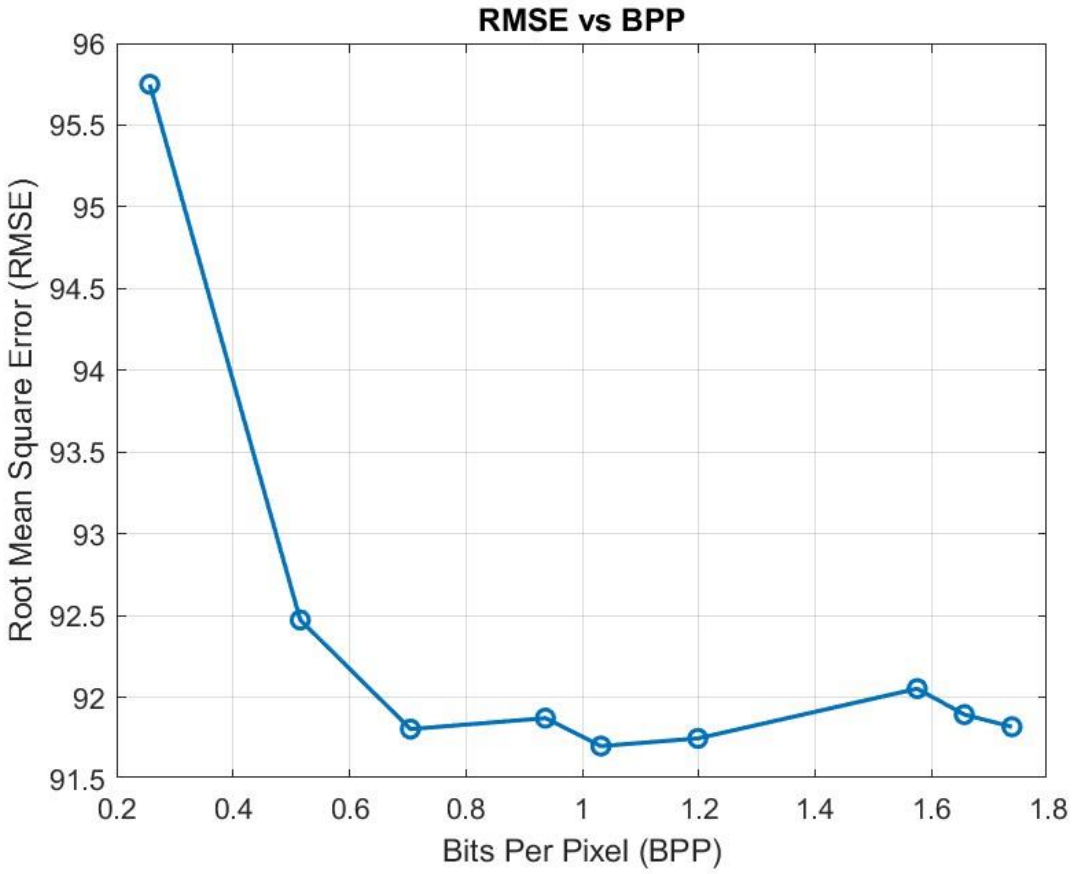
Q=5

Q=55

Q=85

RMSE vs BPP:

For each of the above images, we find the RMSE vs BPP plots with varying Q values. Below is the RMSE vs BPP plot created for the Barbara image.



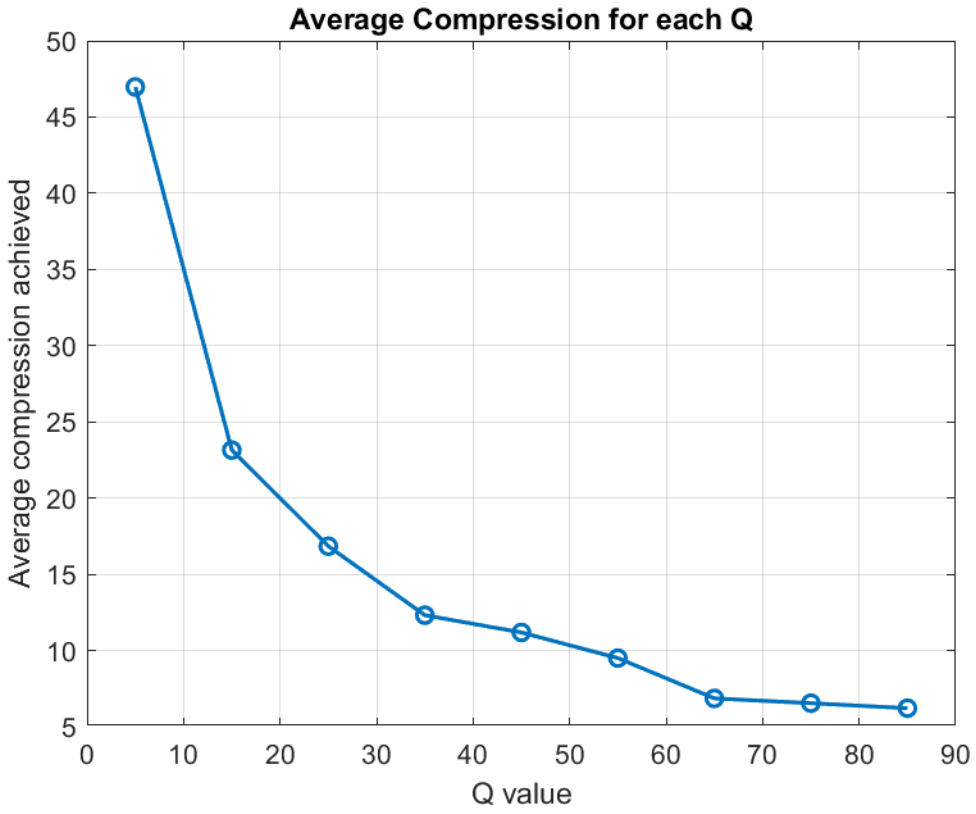
The graph plots the BPP values in the X-axis and RMSE values in the Y-axis for an image each Q value. (for the rest of the images the please check the github).

As we have expected for the heavier compression that is the Q value is less, the BPP is less but the RMSE is very high, similarly when increasing the Q value BPP increases as expected as we storing more data per pixels thus reducing the RMSE.

Average Compression Ratio for each Q value:

For each of the test image we found the Average compression ratio for different Q values and the plot is shown below.

As we have expected, the compression ratio goes down when the Q value is increased because with the increasing Q value, we are storing more data of the image.



Compression using Bitplane slicing:

Description:

We know the MSB carries more important information about the images than the LSB. So, we have tried to implement the compression using bitplane slicing.

Take the 4 MSBs of each pixel, perform Differential Pulse-code modulation and perform Huffman encoding on the data, store the encoded value in bits in a file.

The decoder will read the file and decode the image.

Results of this method:

File sizes:

```
size of JPEG file created by Matlab : 34515
size of original PNG image : 155129
size of JPEG file created by our code : 49606
```

The compression is good, mostly similar to the JPEG compression.

RMSE and compression ratio:

```
RMSE : 4.051522e+01
Compression ratio : 3.127223
```

Even though the compression was good, the RMSE was very high.

Images:

Original image

bitplane compressed image

Compression using the combination of PCA and JPEG Algorithm

I know this will not work, because of the added additional constraints on the image, the image might get more errors. Out of curiosity I tried below.

Divide the image into 8x8 patches. Using the 8x8 blocks, find the eigen coefficients and compress them using the JPEG algorithm.

Decode/decompress the eigen coefficients again and recreate the original image using eigen coefficients.

Result:

Below is the image recreated using this method. We can see several patches visible because of the loss that occurred in the PCA.

The link to all the files are in both drive and github

Drive:

https://drive.google.com/drive/folders/12XlCRhdnMSF26JGOuzKzy1K1xdPFviJy?usp=drive_link

Github: https://github.com/Mani-iitb/CS663