Fundamentals of Digital Image Processing

Project Report | Image Compression

Autumn 2024

Mayank Kumar | 22B0933

Overview

1. Problem Statement
2. JPEG Image Compression Engine
3. Performance of Our Compression Engine
4. Comparison with JPEG (MATLAB)
5. Implementation of PCA based Compression Engine
6. Comparison with PCA based Compression Engine
7. Conclusion
8. Dataset

1. Problem Statement

Part 1: Implementing an Image Compression Engine [JPEG]

- Compute 2D DCT coefficients of non-overlapping image patches.
- Implementation of the quantization step
- Implementation of Huffman tree for encoding data
- Writing the data to file which can be read and image can be reconstructed later
- Simulate various quality factors and plot RMSE vs. BPP
  - **RMSE:** Relative Root Mean Squared Error between original and compressed images.
  - **BPP:** Bits per pixel (compressed image size divided by pixel count).

Part 2: Innovation and Advanced Features

- Comparison with existing JPEG implementations (MATLAB)
- Experiments on color images with different parameter settings.
- Implementation or study of a research paper.
- Comparison with PCA-based image compression techniques for specific image classes.

2. JPEG Image Compression Engine

Description of Implemented Algorithms

Computation of DCT Coefficients & Quantization

- For each 8x8 block, the 2D Discrete Cosine Transform (DCT) is computed using `dct2`.
- The resulting DCT coefficients are quantized by dividing them element-wise with quantization matrix Q and rounding the values.

```
base_Q = [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];
```

(a) Default Quantization Matrix

```
scale =  50 / quality_factors(qf_idx);
Q = max(round(base_Q * scale), 1);

quantized_coefs = zeros(rows_padded, cols_padded);
for i = 1:8:rows_padded
    for j = 1:8:cols_padded
        block = padded_img(i:i+7, j:j+7);
        dct_block = dct2(block);
        quantized_block = round(dct_block ./ Q);
        quantized_coefs(i:i+7, j:j+7) = quantized_block;
    end
end
```

(b) Code

Huffman Encoding

- Identification of unique symbols in the quantized coefficients matrix and calculates their frequency counts using `histcounts`.
- A Huffman dictionary is generated using the unique symbols and their normalized probabilities, derived from the frequency counts.
- The quantized coefficients are then encoded into a compressed bitstream using `huffmanenco`

```
symbols = unique(quantized_coefs);
counts = histcounts(quantized_coefs(:), [symbols; max(symbols)+1]);
huffman_dict = huffmandict(symbols, counts / numel(quantized_coefs));
compressed_stream = huffmanenco(quantized_coefs(:), huffman_dict);
```

Write to & Read from File

- The `save` command stores the compressed bitstream, Huffman dictionary, image dimensions (`rows` and `cols`), and quantization matrix (`Q`), which is saved in the `MATLAB.mat.24.1.0` format.
- The `load` command retrieves the saved data from the file for decoding and reconstruction of the original image.
- The file size of `compressed_data.mat` is determined using `dir` for BPP calculation.

```
save('compressed_data.mat', 'compressed_stream', 'huffman_dict', 'rows', 'cols', 'Q');

load('compressed_data.mat');
file_info = dir('compressed_data.mat');
compressed_bytes = file_info.bytes;
compressed_bits = compressed_bytes * 8;
```

Image Reconstruction

- The compressed bitstream is decoded using the Huffman dictionary, reshaped into the original padded dimensions.
- For each 8×8 block, inverse quantization and inverse DCT are applied to reconstruct the image.

```
decoded_stream = huffmandeco(compressed_stream, huffman_dict);
decoded_quantized = reshape(decoded_stream, rows_padded, cols_padded);

reconstructed_img = zeros(rows_padded, cols_padded);
for i = 1:8:rows_padded
    for j = 1:8:cols_padded
        quantized_block = decoded_quantized(i:i+7, j:j+7);
        dequantized_block = quantized_block .* Q; % Inverse quantization
        idct_block = idct2(dequantized_block);    % Inverse DCT
        reconstructed_img(i:i+7, j:j+7) = idct_block;
    end
end
```

Modifications for Colour Images

- Conversion from RGB to YCbCr
- Downsampling Cb and Cr channels
- Encoding each channel separately

```
img_ycbcr = rgb2ycbcr(uint8(img));
Y = img_ycbcr(:,:,1);
Cb = img_ycbcr(:,:,2);
Cr = img_ycbcr(:,:,3);
[rows, cols, ~] = size(img);
```

```
Cb_down = imresize(Cb, 0.5, 'bilinear');
Cr_down = imresize(Cr, 0.5, 'bilinear');
```

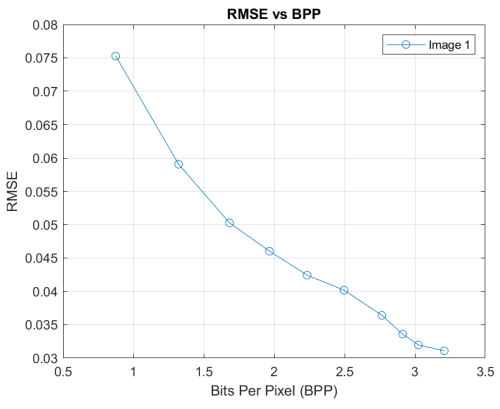
```
quantized_Y = dct_quantize_block(padded_Y, Q, rows_padded, cols_padded);
quantized_Cb_down = dct_quantize_block(padded_Cb_down, Q, down_rows_padded, down_cols_padded);
quantized_Cr_down = dct_quantize_block(padded_Cr_down, Q, down_rows_padded, down_cols_padded);
```

3. Performance of Our Compression Engine

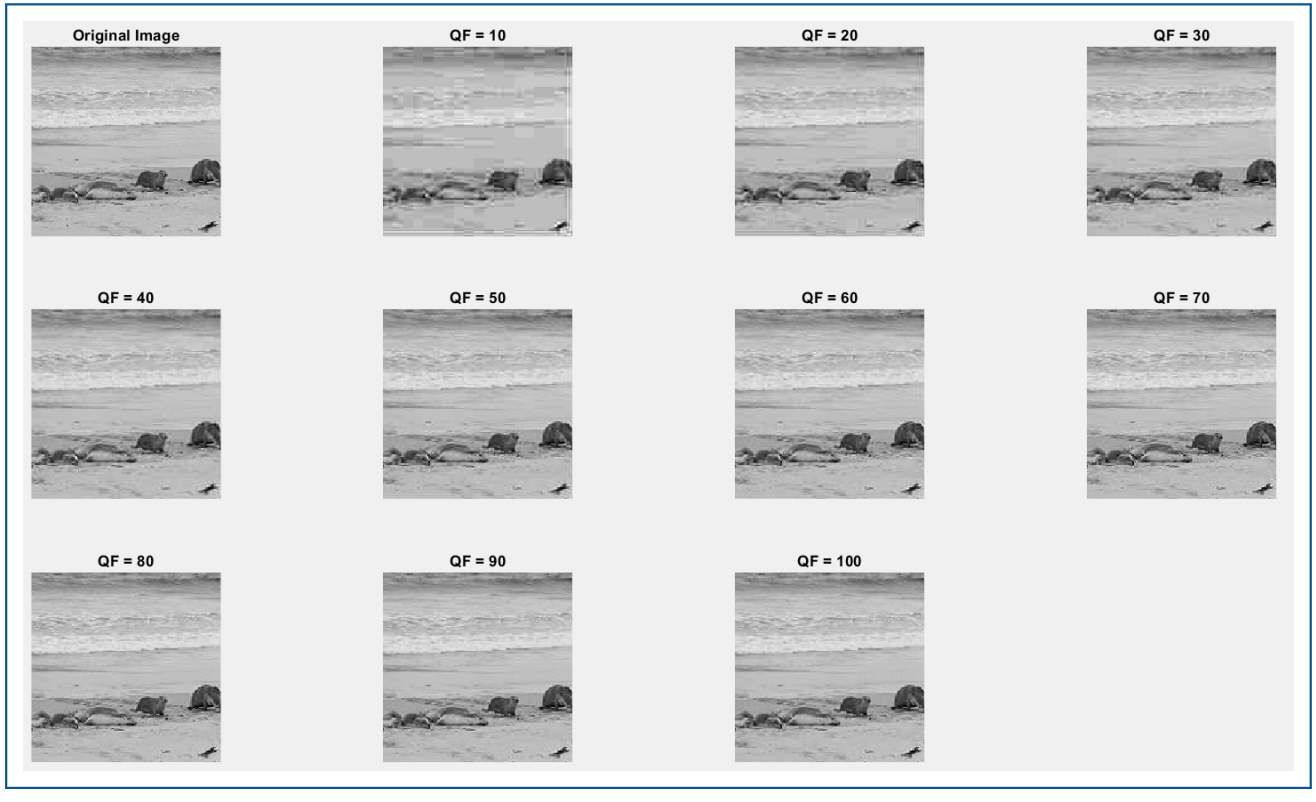
RMSE v/s BPP for Grayscale Images

RMSE v/s BPP Plots & Image Reconstructions [Grayscale]

- RMSE varies from 7.5% for Qf = 10 to 3% for Qf = 100

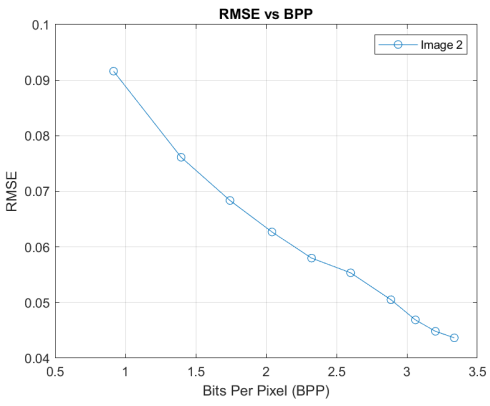


Qf = 10:10:100



RMSE v/s BPP Plots & Image Reconstructions [Grayscale]

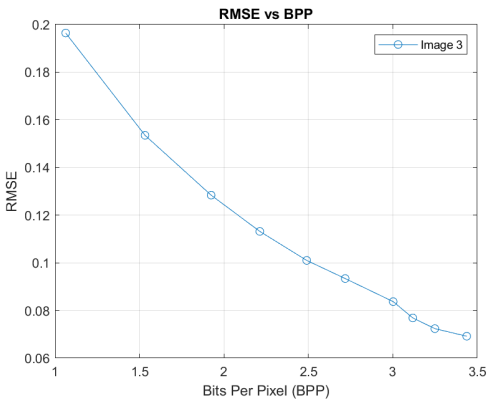
- RMSE varies from 9% for Qf = 10 to 4% for Qf = 100



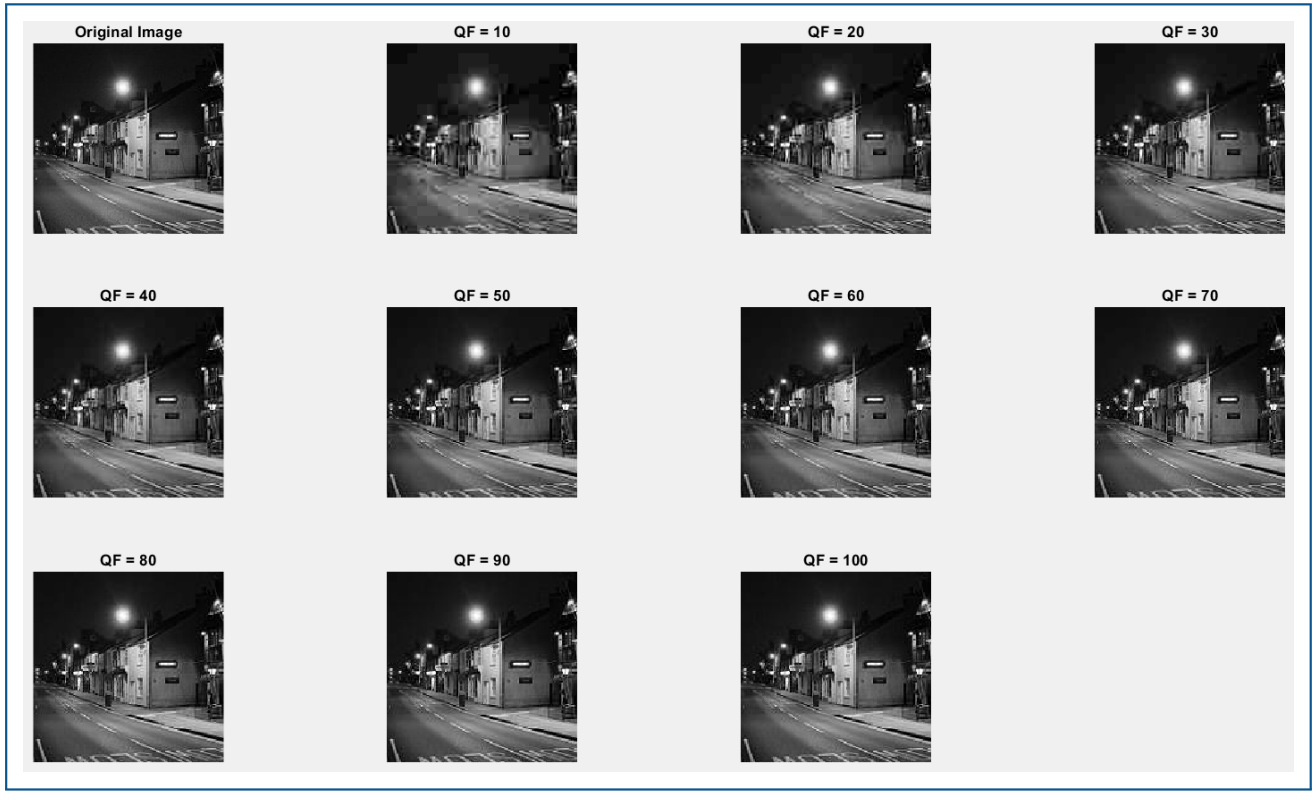
Qf = 10:10:100

RMSE v/s BPP Plots & Image Reconstructions [Grayscale]

- RMSE varies from 20% for Qf = 10 to 6% for Qf = 100

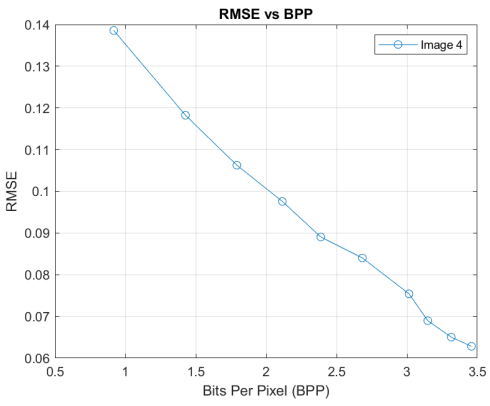


Qf = 10:10:100

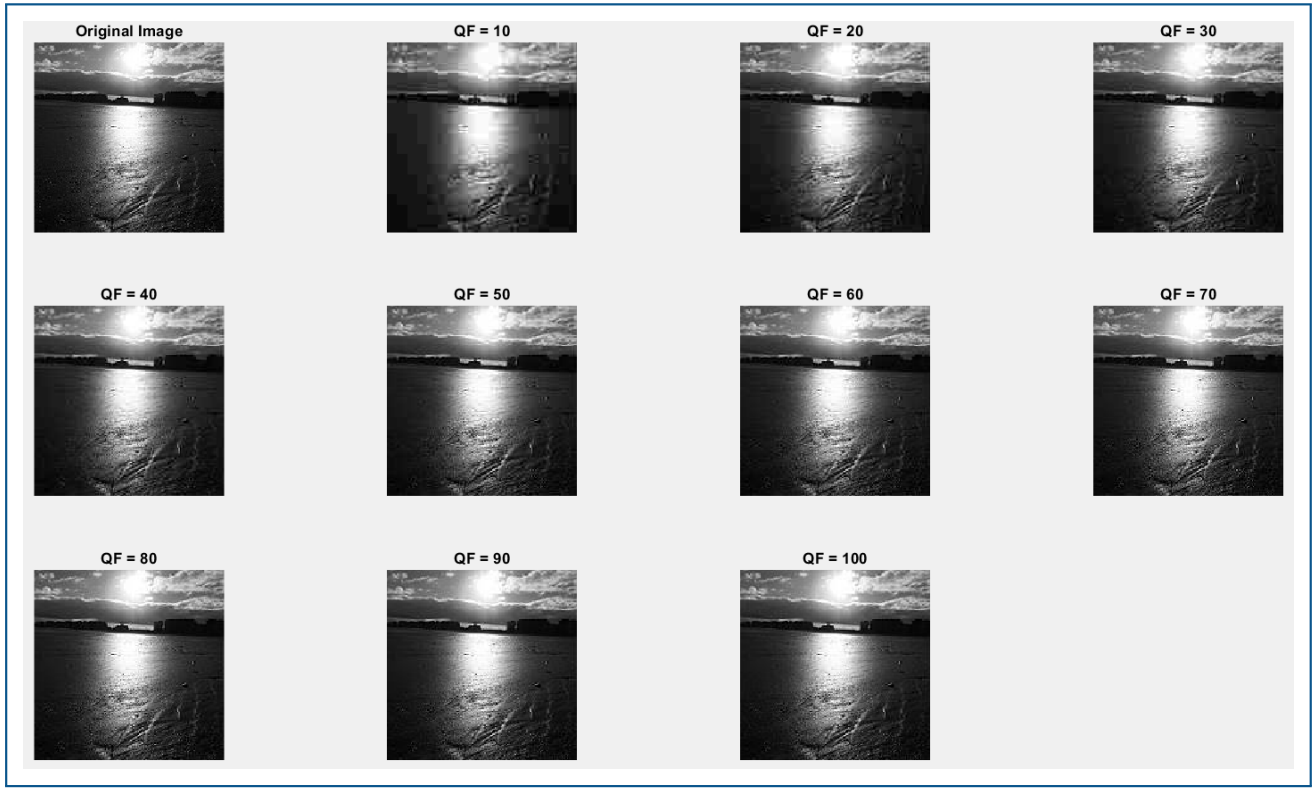


RMSE v/s BPP Plots & Image Reconstructions [Grayscale]

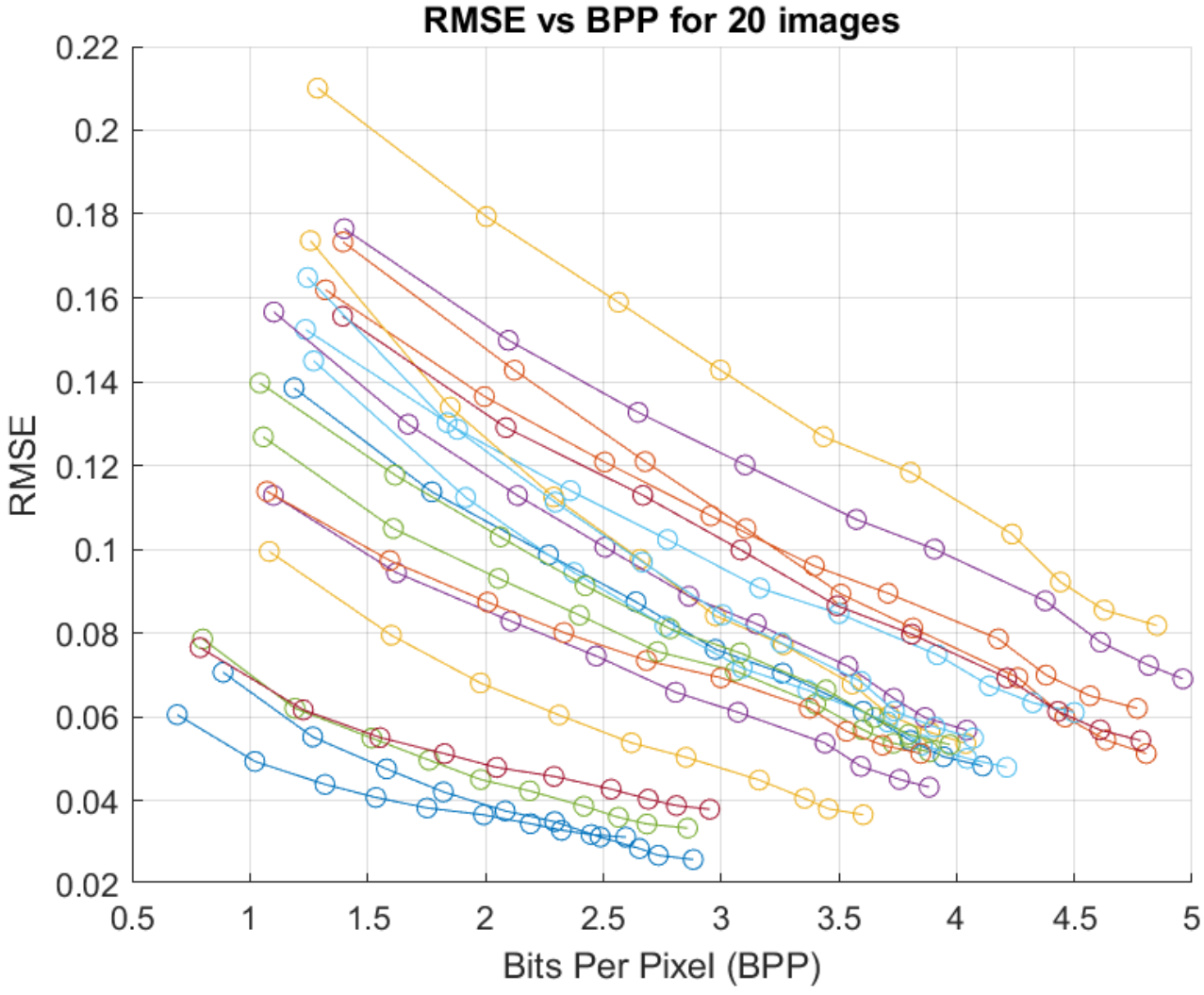
- RMSE varies from 14% for Qf = 10 to 6% for Qf = 100



Qf = 10:10:100



Cumulative RMSE v/s BPP Plot [Grayscale]

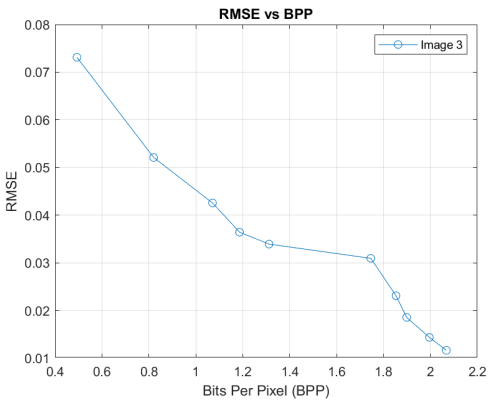


3. Performance of Our Compression Engine

RMSE v/s BPP for Coloured Images

RMSE v/s BPP Plots & Image Reconstructions [Coloured]

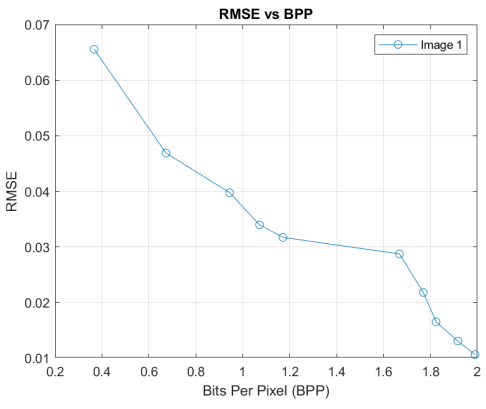
- RMSE varies from 8% for Qf = 10 to 1% for Qf = 100



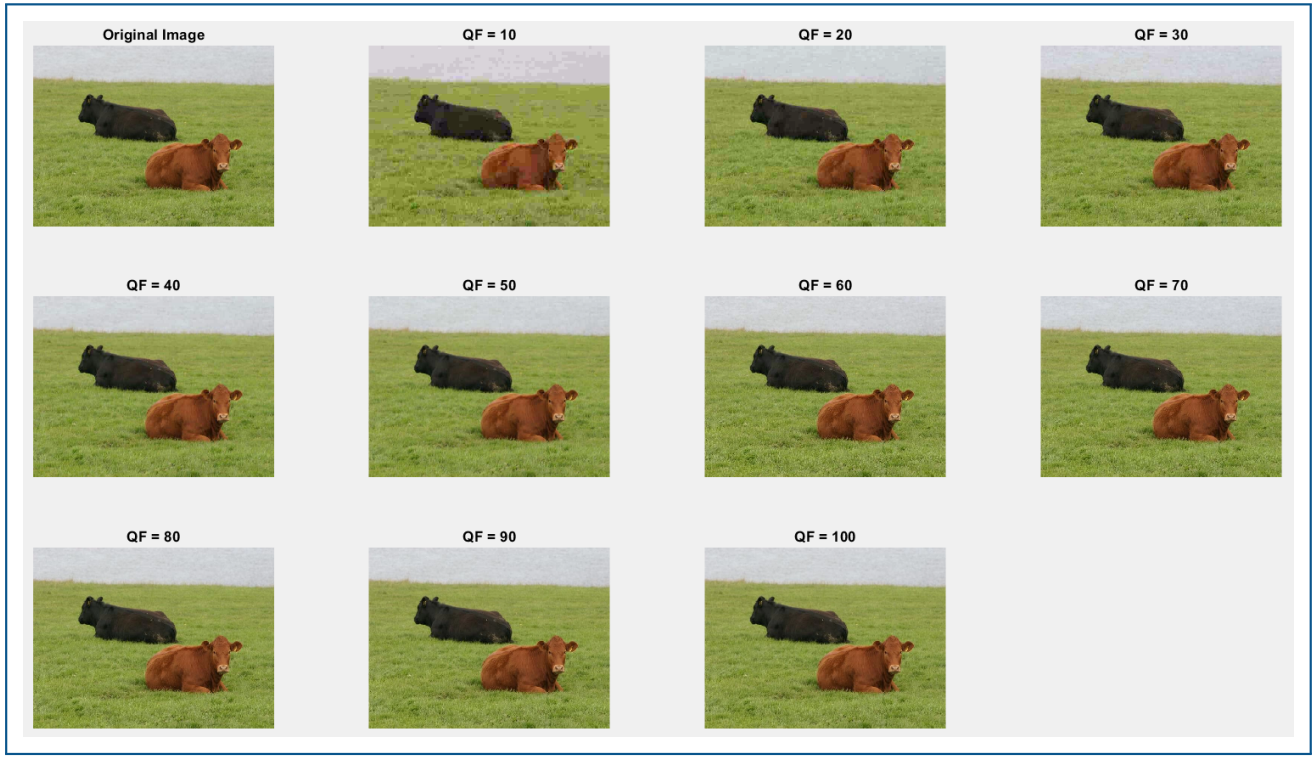
Qf = 10:10:100

RMSE v/s BPP Plots & Image Reconstructions [Coloured]

- RMSE varies from 7% for Qf = 10 to 1% for Qf = 100

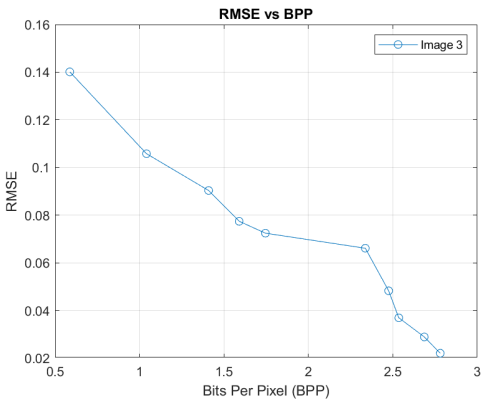


Qf = 10:10:100

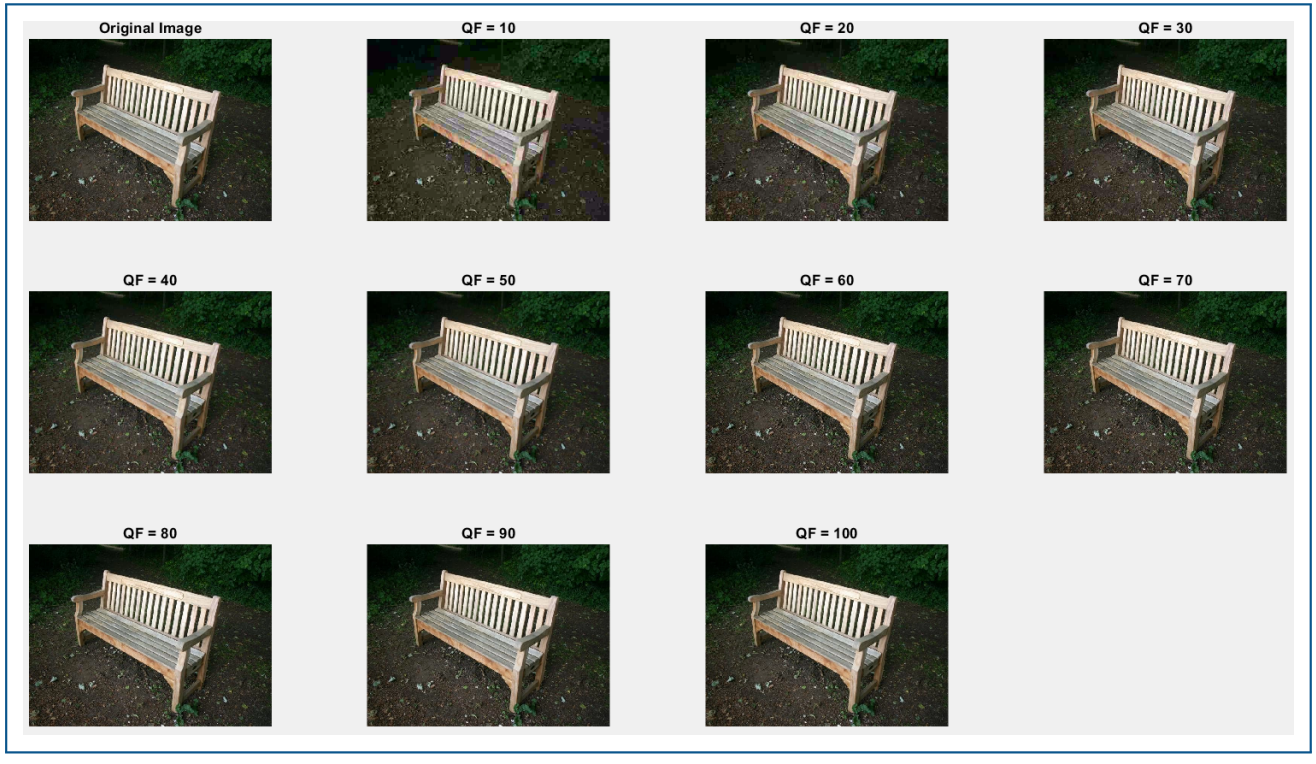


RMSE v/s BPP Plots & Image Reconstructions [Coloured]

- RMSE varies from 14% for Qf = 10 to 2% for Qf = 100

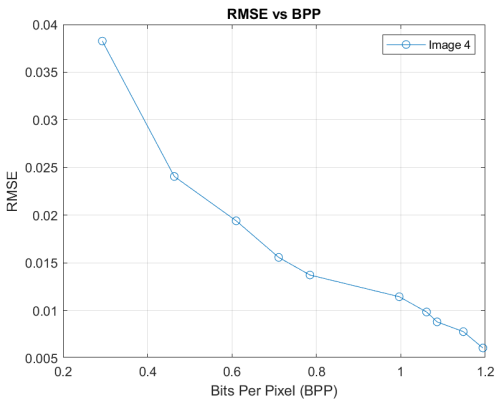


Qf = 10:10:100



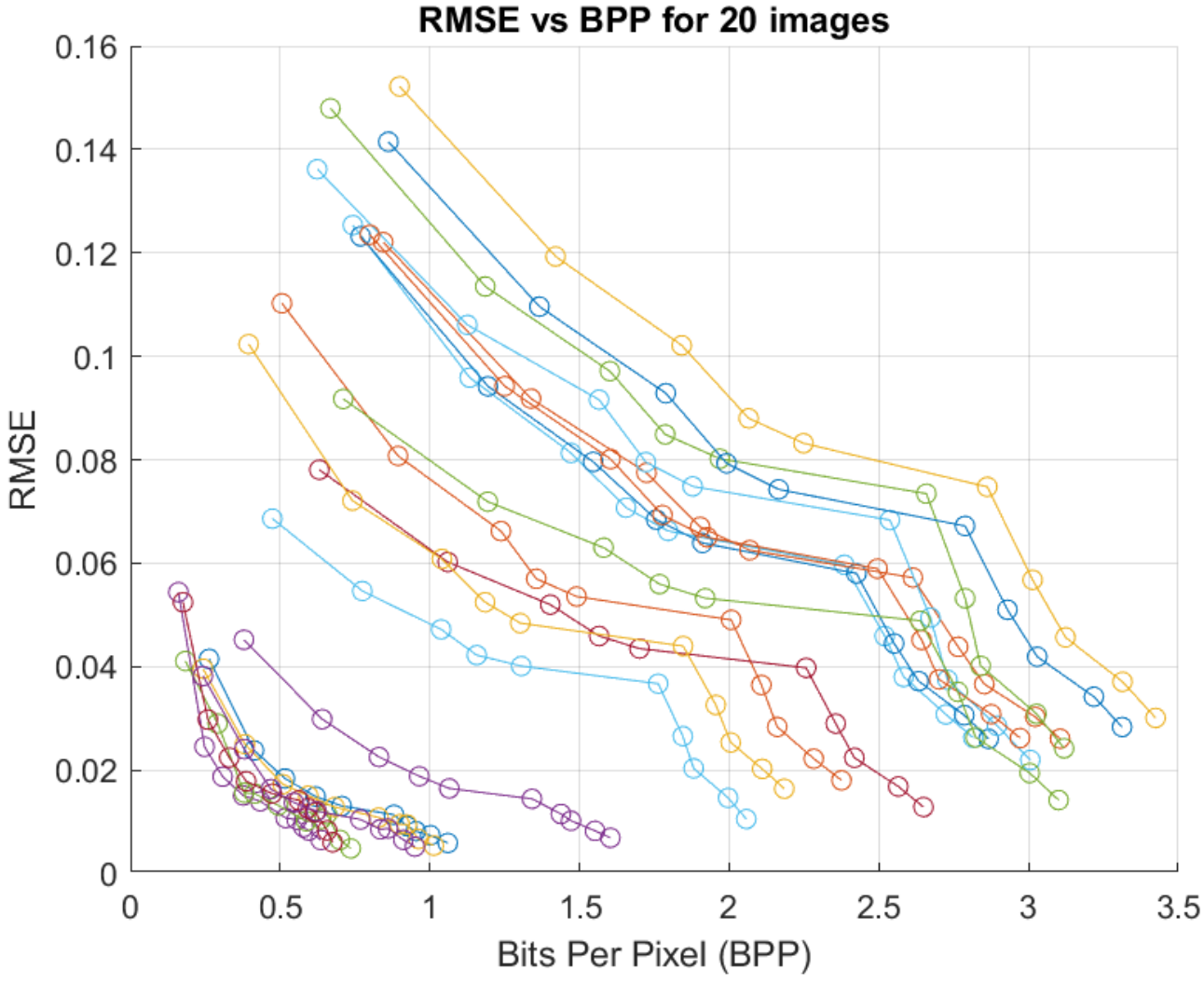
RMSE v/s BPP Plots & Image Reconstructions [Coloured]

- RMSE varies from 4% for Qf = 10 to 0.5% for Qf = 100



Qf = 10:10:100

Cumulative RMSE v/s BPP Plot [Coloured]



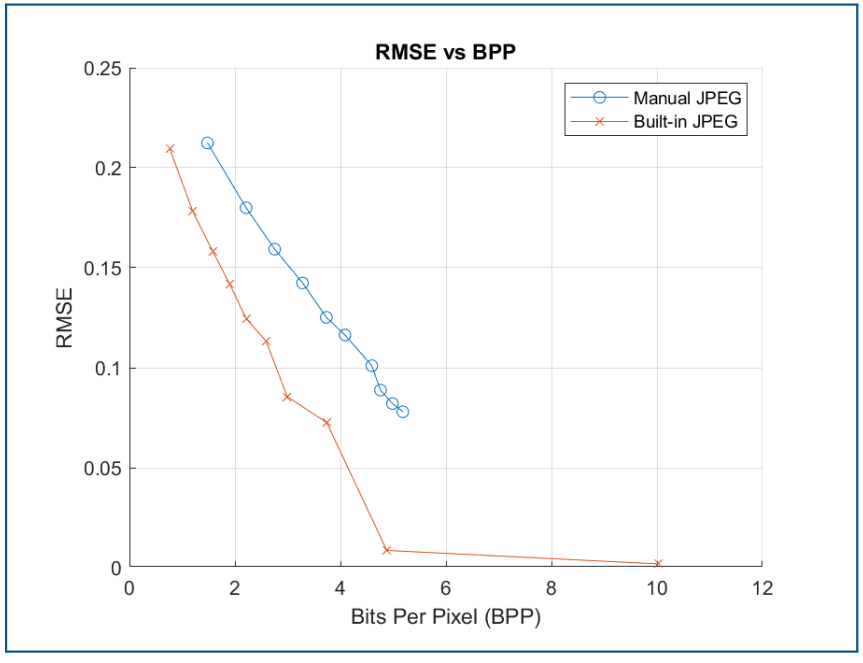
Observations and Analysis

- The images which have more pixels with higher intensity values have lower RMSE values for same Qf because for these images the quantization step results in less loss of information.
- RMSE value is less than 30% with atleast 85% compression for all images when Qf $= 10$.
- The plots for grayscale images is almost linear with RMSE decreasing as BPP increases.
- JPEG artifacts like Seam artifact and Colour artifact are observed for lower values of Quality factor.

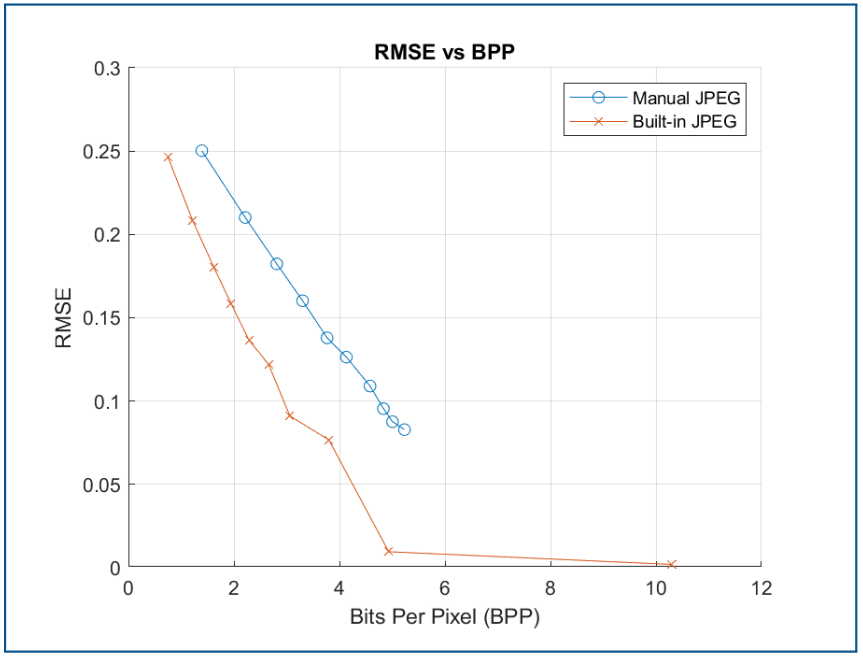
4. Comparison with JPEG (MATLAB)

Grayscale Images

RMSE v/s BPP Plot | Our Engine with MATLAB JPEG

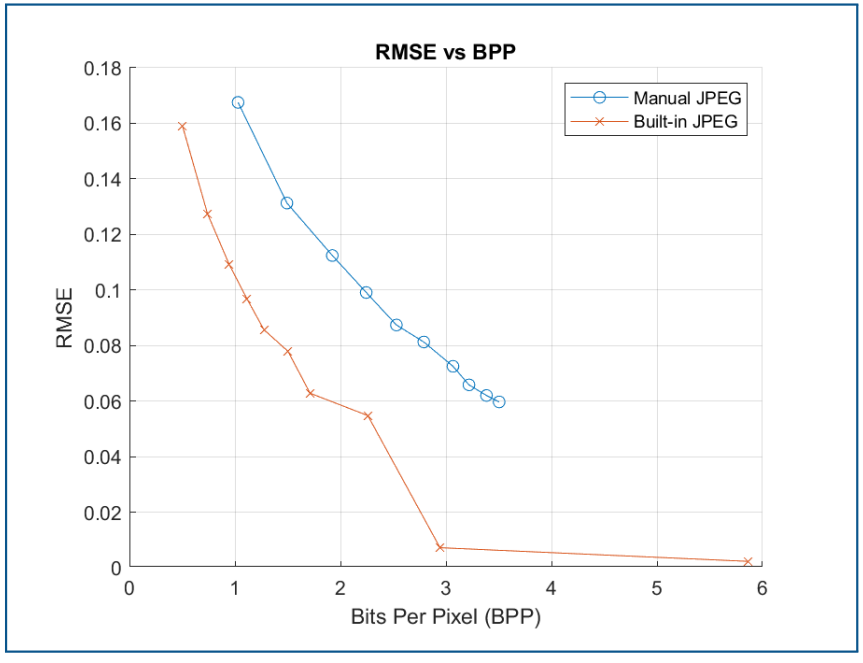


Qf = 10:10:100

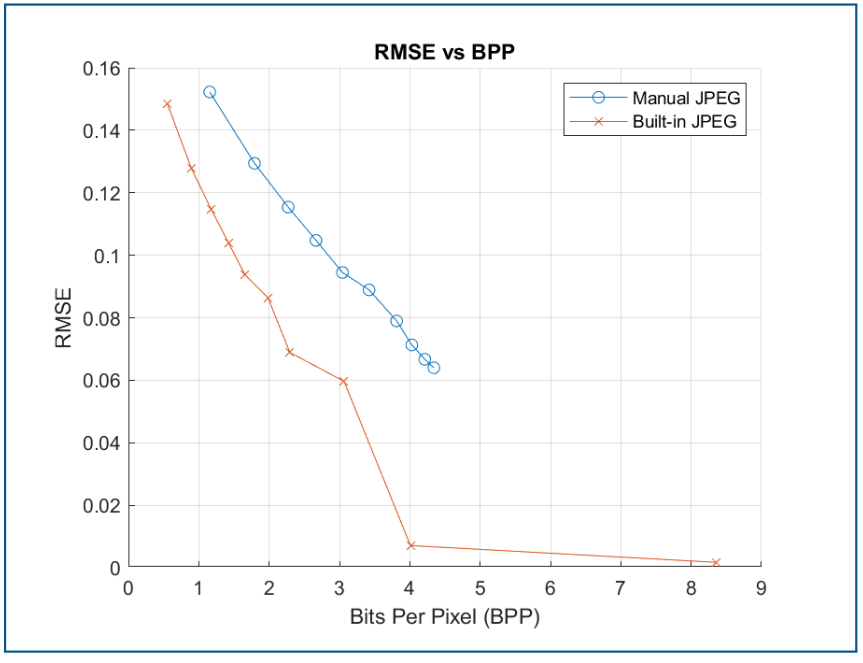


Qf = 10:10:100

RMSE v/s BPP Plot | Our Engine with MATLAB JPEG



Qf = 10:10:100

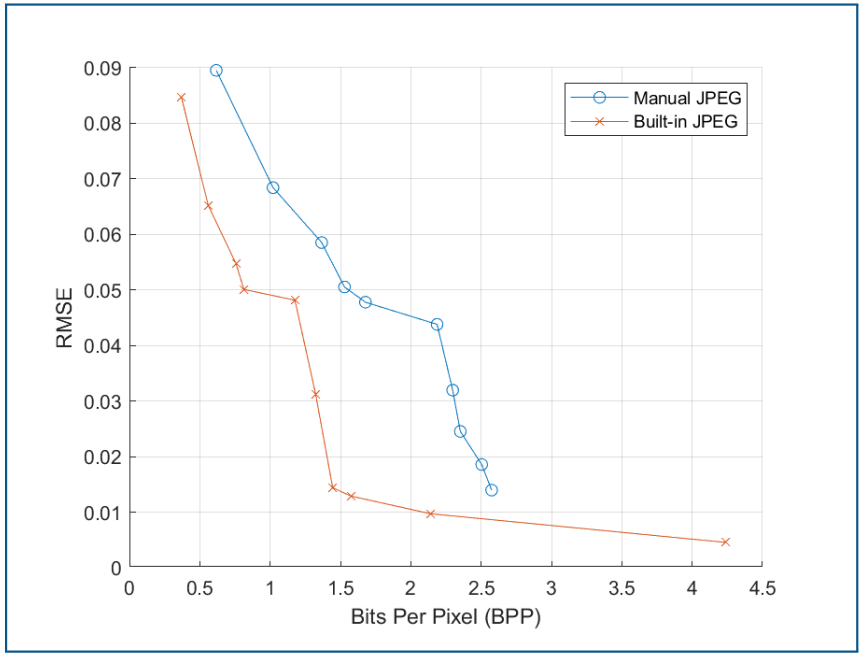


Qf = 10:10:100

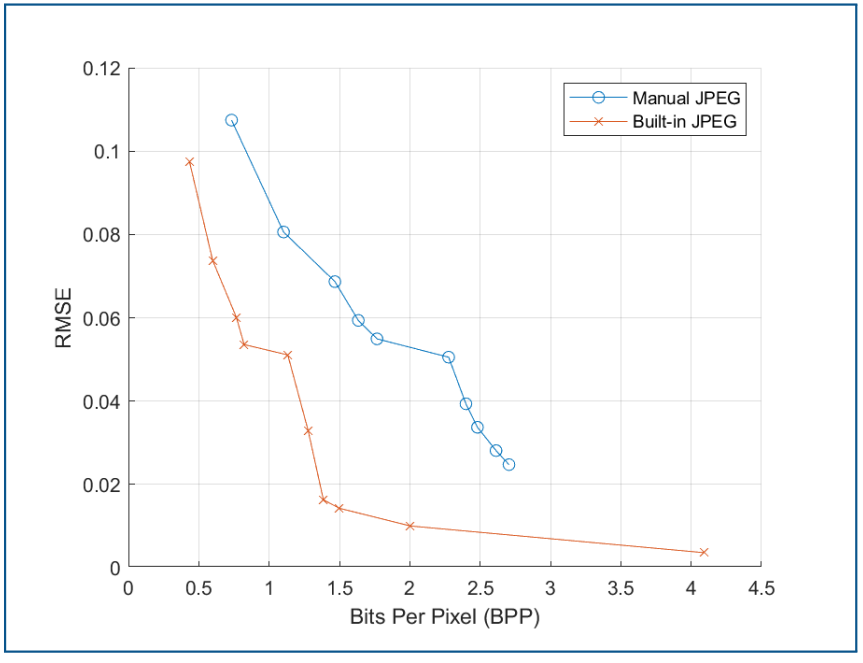
4. Comparison with JPEG (MATLAB)

Coloured Images

RMSE v/s BPP Plot | Our Engine with MATLAB JPEG

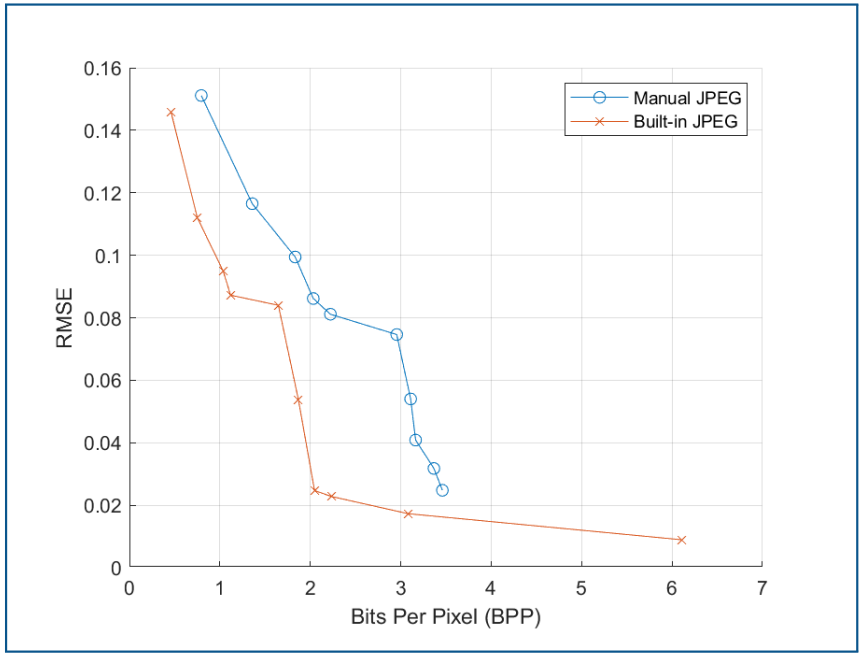


Qf = 10:10:100

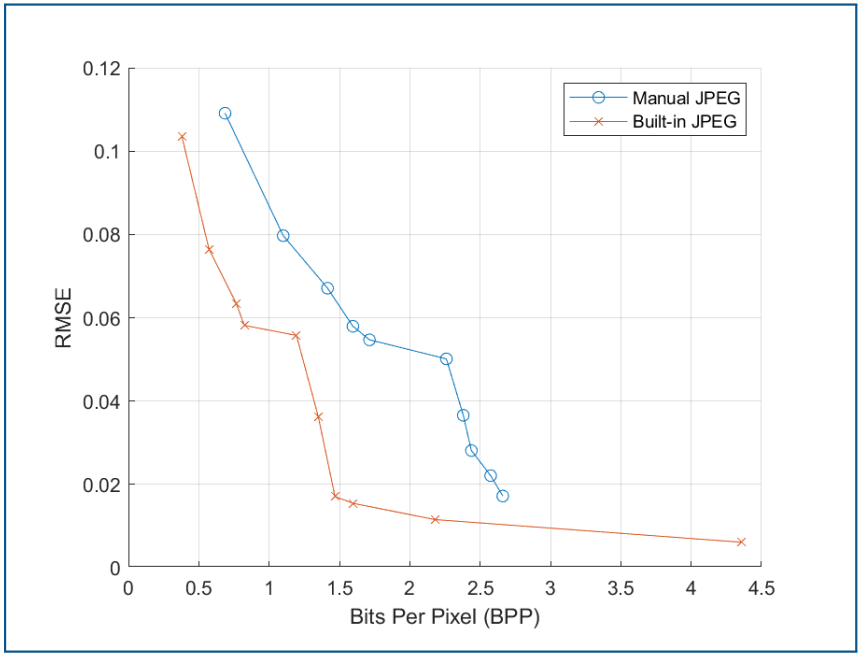


Qf = 10:10:100

RMSE v/s BPP Plot | Our Engine with MATLAB JPEG



Qf = 10:10:100



Qf = 10:10:100

Observations and Analysis

- **General Trend:** Both algorithms exhibit a reduction in RMSE as BPP increases, indicating better reconstruction quality at higher bit rates.
- For high `Quality factor` values (80-100), the *Built-in JPEG* behaves aggressively, causing a sharp increase in BPP, which suggests a change in the compression algorithm.
- Although RMSE varies based on image content, the performance of the `Built-in JPEG` closely aligns with our implementation, indicating its correctness.

5. Implementation of PCA based Compression Engine

PCA Basis

- Extract Patches of dimentsion [pS * pS] from the images in the set
- Calculate the row and column correlation matrices
- Calculate the PCA basis as the kronecker product of eigen vectors of correlation matrices

```
for j = 1:patchSize:H-patchSize+1
    for k = 1:patchSize:W-patchSize+1
        v = im(j:j+patchSize-1, k:k+patchSize-1);
        patchCount = patchCount + 1;
        patches{patchCount} = v;
    end
end
```

```
CR = zeros(patchSize); CC = CR;
for i=1:patchCount
    v = patches{i};
    CC = CC + v*v';
    CR = CR + v'*v;
end
CC = CC/(patchCount-1);
CR = CR/(patchCount-1);

[VR,DR] = eig(CR);
[VC,DC] = eig(CC);
VR = VR(:,patchSize:-1:1);
VC = VC(:,patchSize:-1:1);

PCA_basis = kron(VR,VC);
```

Implementation of PCA based Compression Engine

Image Compression

- Divide the image to be compressed into patches
- Express each patch as linear combination of PCA basis
- Store the coefficients

```
coefficients = zeros(patchSize^2, num_patches);
patch_idx = 1;
for j = 1:patchSize:H_padded
    for k = 1:patchSize:W_padded
        patch = img_padded(j:j+patchSize-1, k:k+patchSize-1);
        patch_vector = patch(:);

        coeff = PCA_basis \ patch_vector;
        coefficients(:, patch_idx) = coeff;

        patch_idx = patch_idx + 1;
    end
end
```

Image Reconstruction

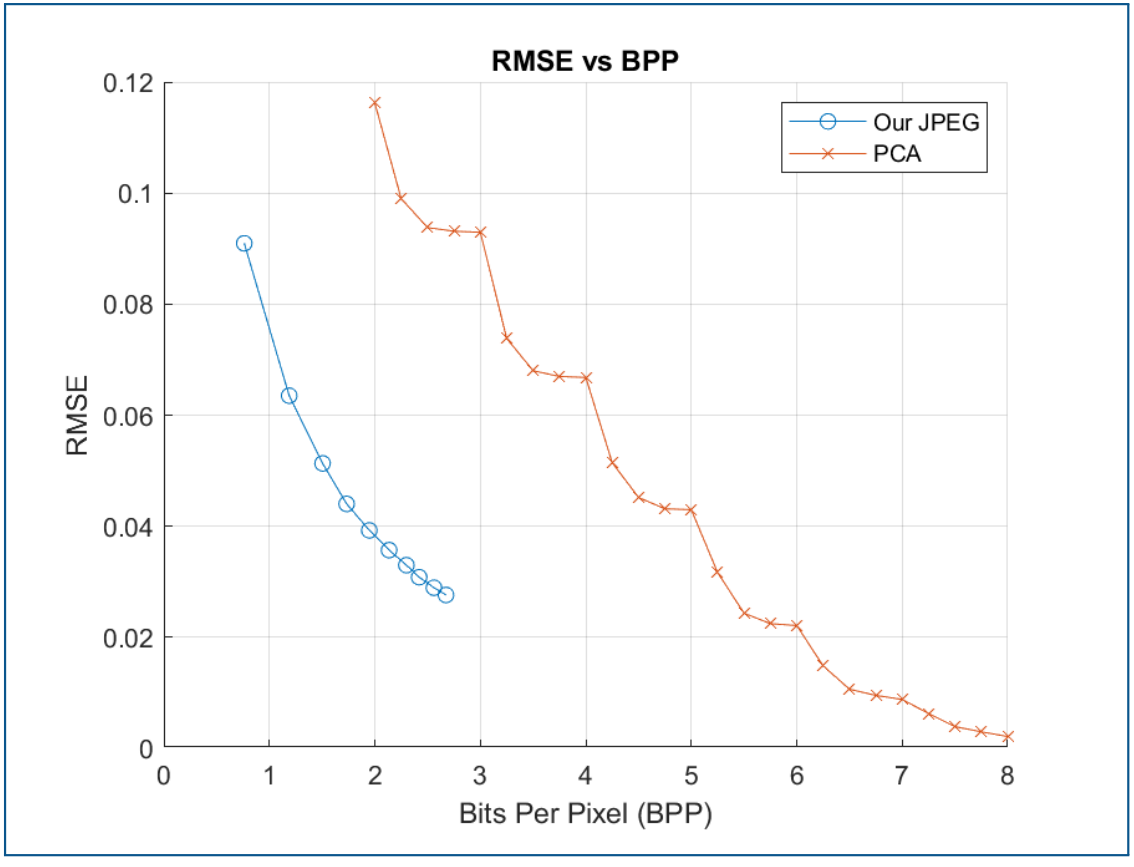
- Depending on the requirement, adjust `numComponents` from 1 to $(pS)^2$
- Reconstruct each patch taking only `numComponents` columns from `PCA Basis`

```
for j = 1:patchSize:H_padded
    for k = 1:patchSize:W_padded
        coeff_top_k = coefficients(:, patch_idx);
        coeff_top_k(numComponents+1:end) = 0;

        patch_vector_reconstructed = PCA_basis * coeff_top_k;
        patch_reconstructed = reshape(patch_vector_reconstructed, [patchSize, patchSize]);

        reconstructed_patches(:, :, patch_idx) = patch_reconstructed;
        patch_idx = patch_idx + 1;
    end
end
```

6. Comparison with PCA based Compression Engine

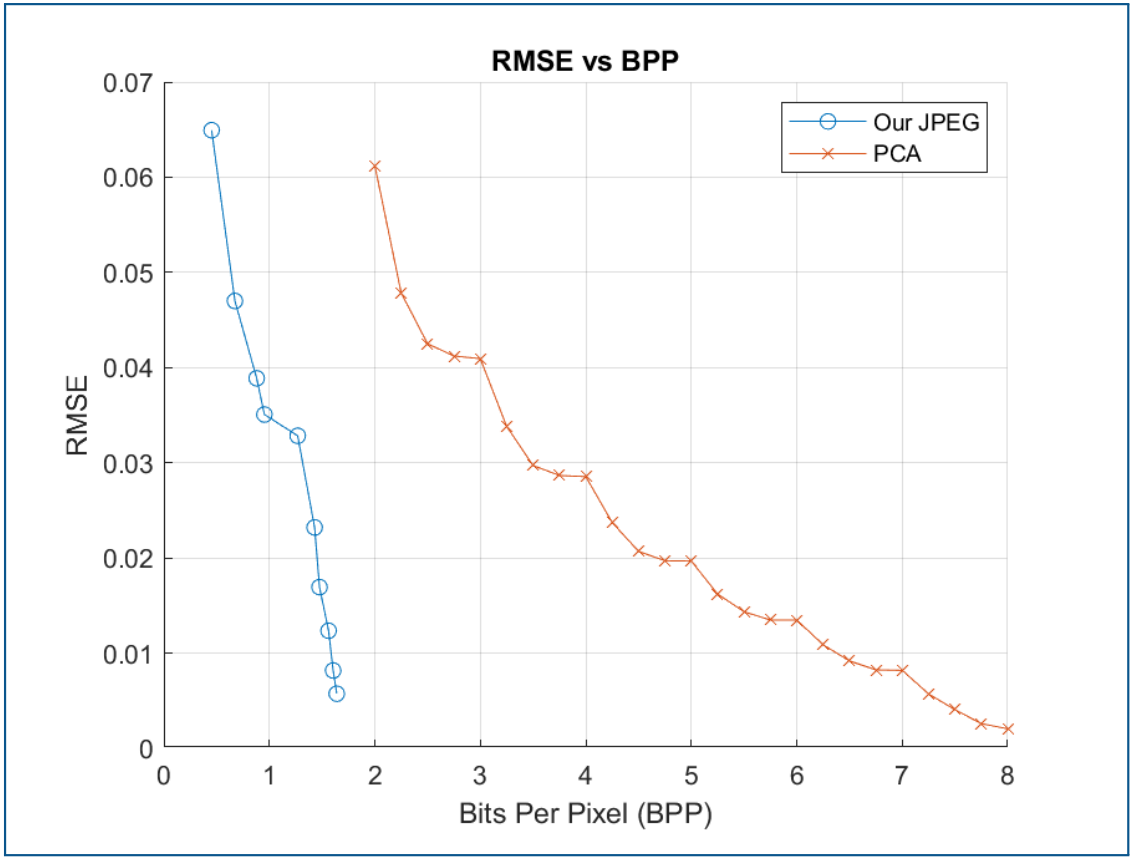
Single Images

Comparison with PCA | Single Image



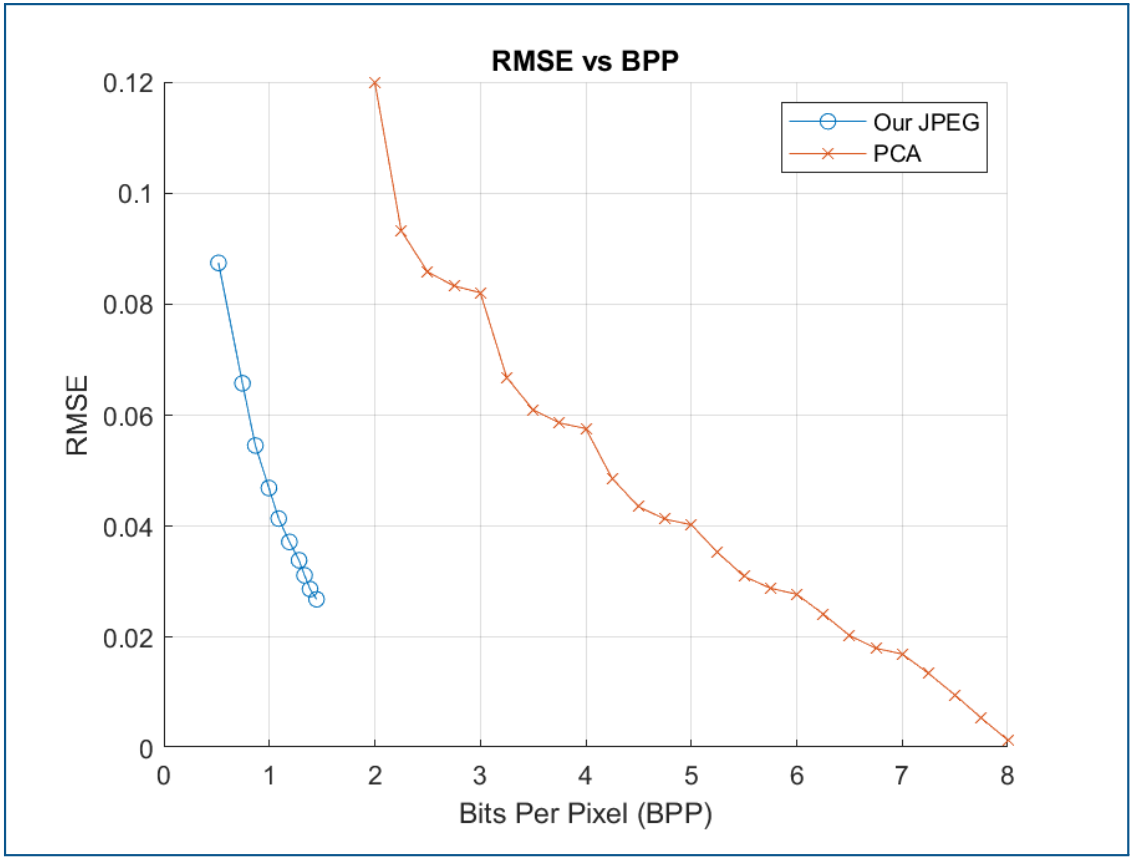
Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Single Image



Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Single Image



Qf = 10:10:100 & numComponents = 16:2:64

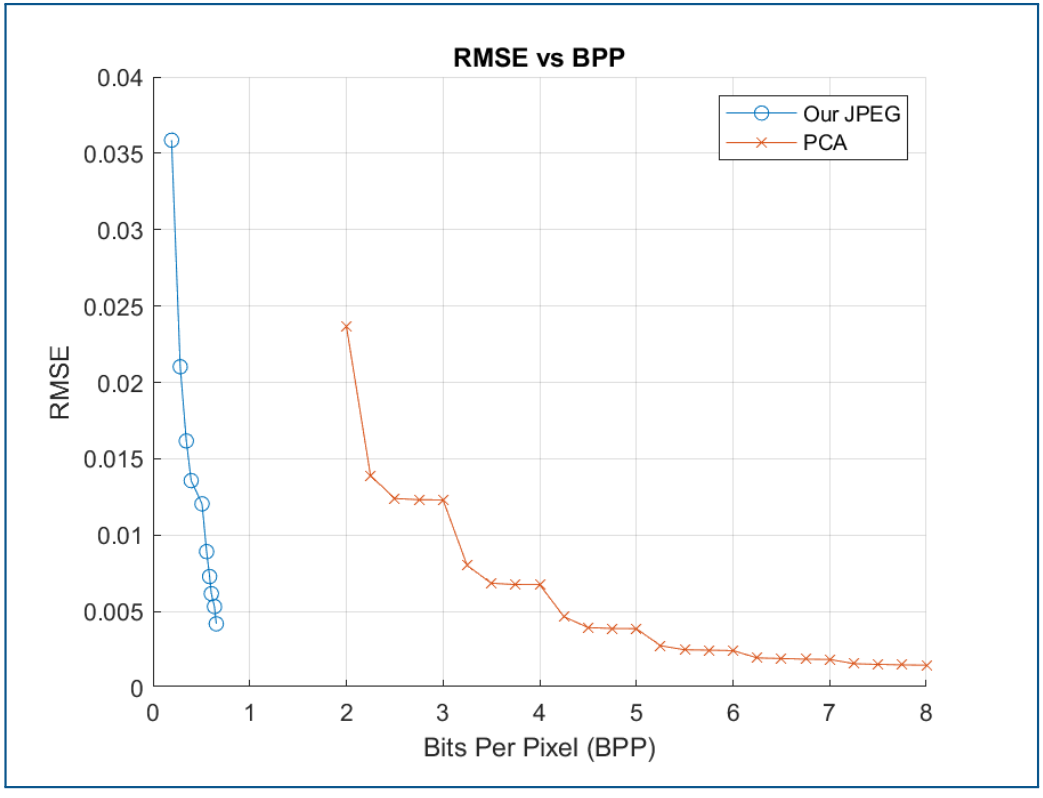
Single Image Compression

- For the first and third images, there are numerous repetitive patterns throughout, resulting in extracted patches that are highly similar to one another.
- At moderate BPP values (4-6), the PCA compression engine achieves very low RMSE values, making it particularly effective for images with repetitive patterns if high quality is required.
- For the second image, which depicts an airplane, there are fewer repetitive patterns. This leads to less efficient compression and a relatively suboptimal BPP.
- For all 3 images, JPEG engine is able to achieve [70 - 80]% compression with low RMSE.

6. Comparison with PCA based Compression Engine

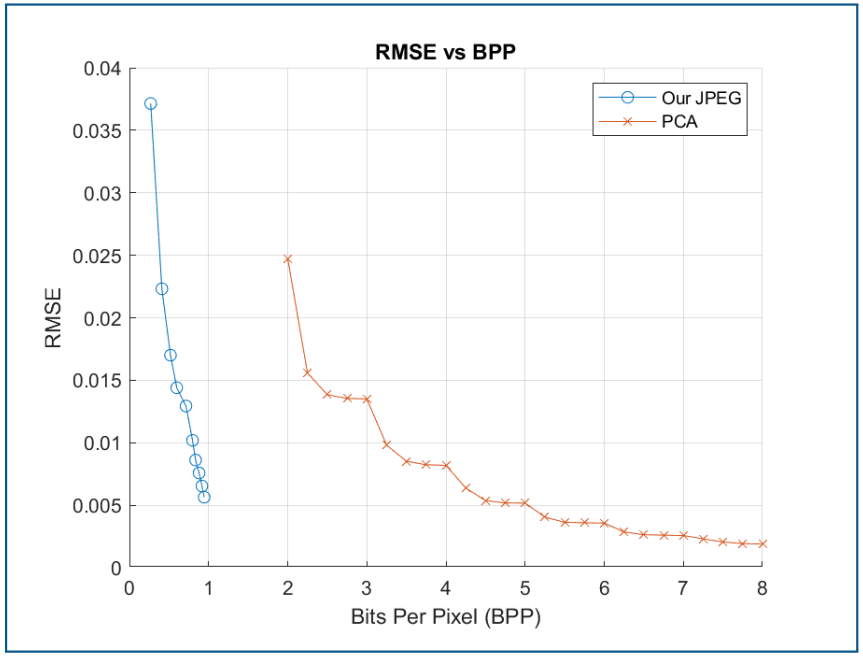
Class of Images

Comparison with PCA | Image Class - Flowers

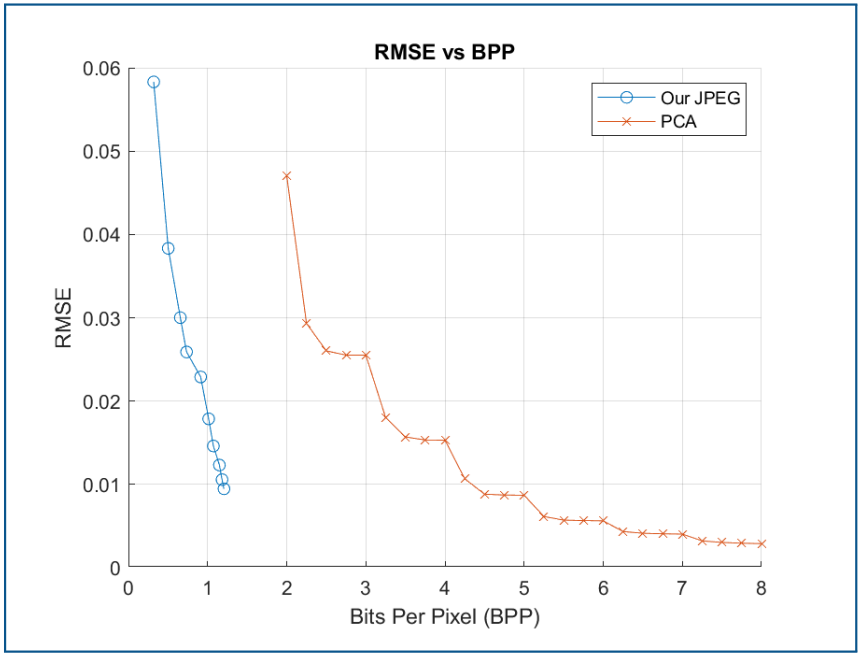


Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Image Class - Flowers

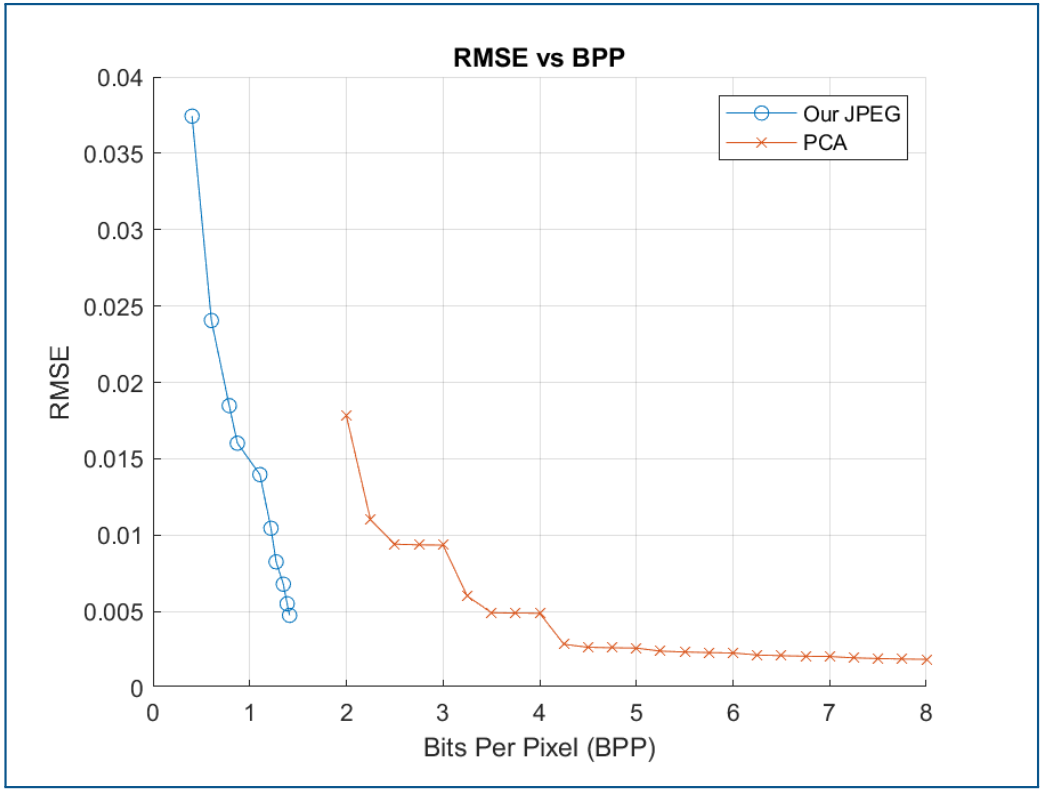


Qf = 10:10:100 & numComponents = 16:2:64



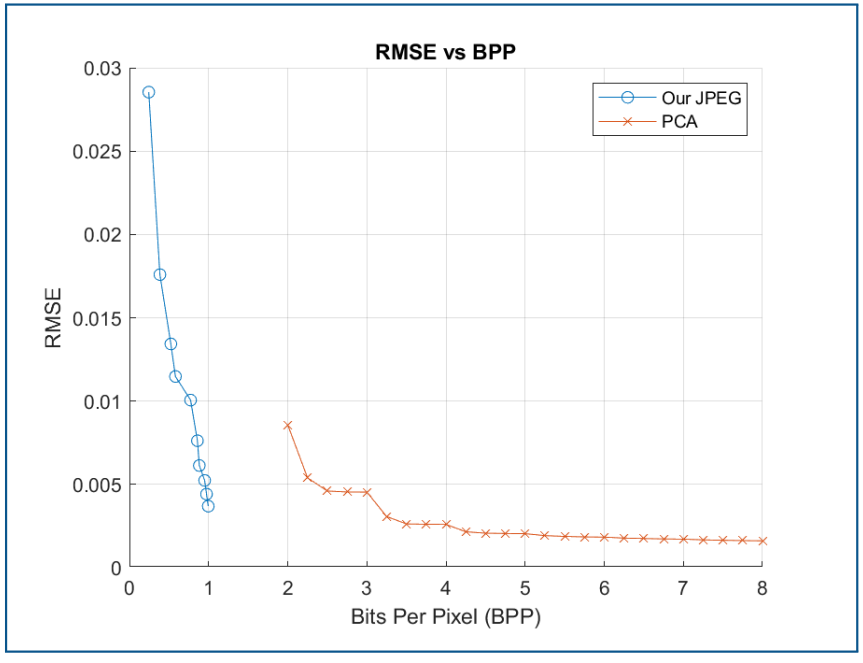
Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Image Class - Clouds

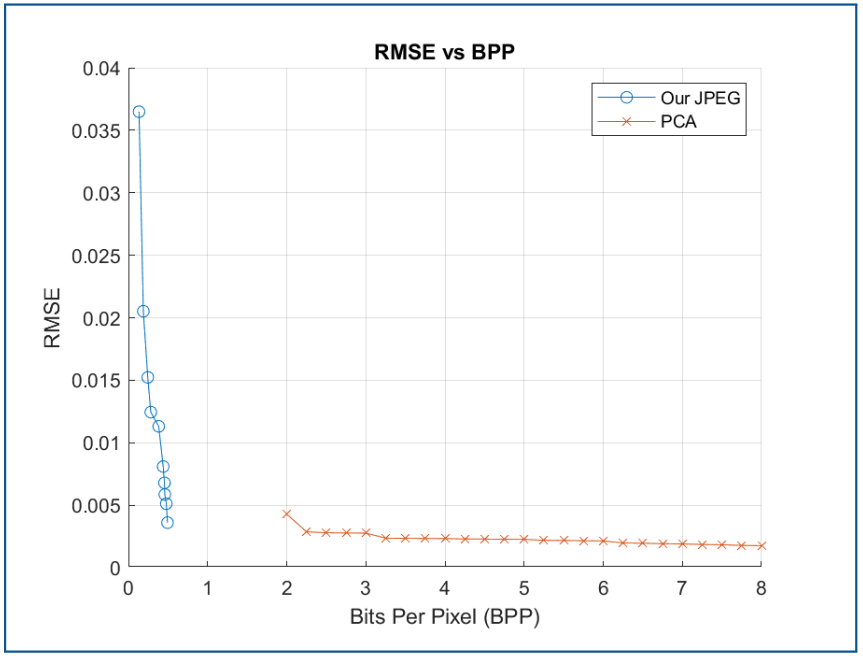


Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Image Class - Clouds

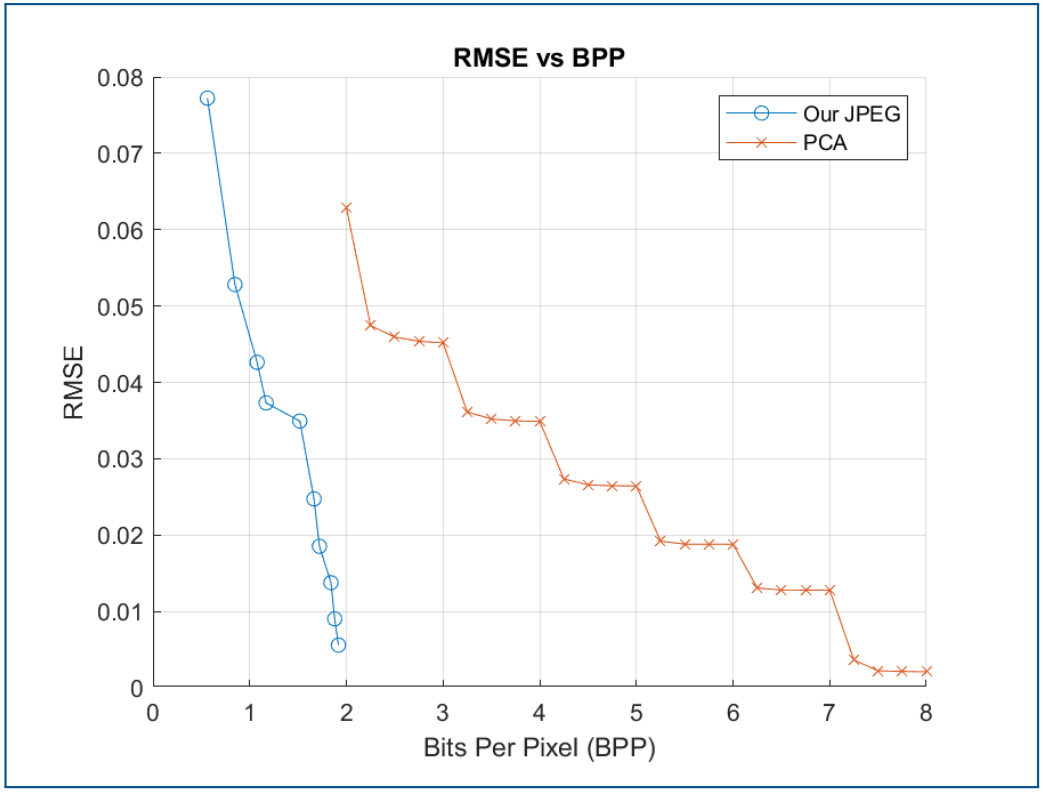


Qf = 10:10:100 & numComponents = 16:2:64



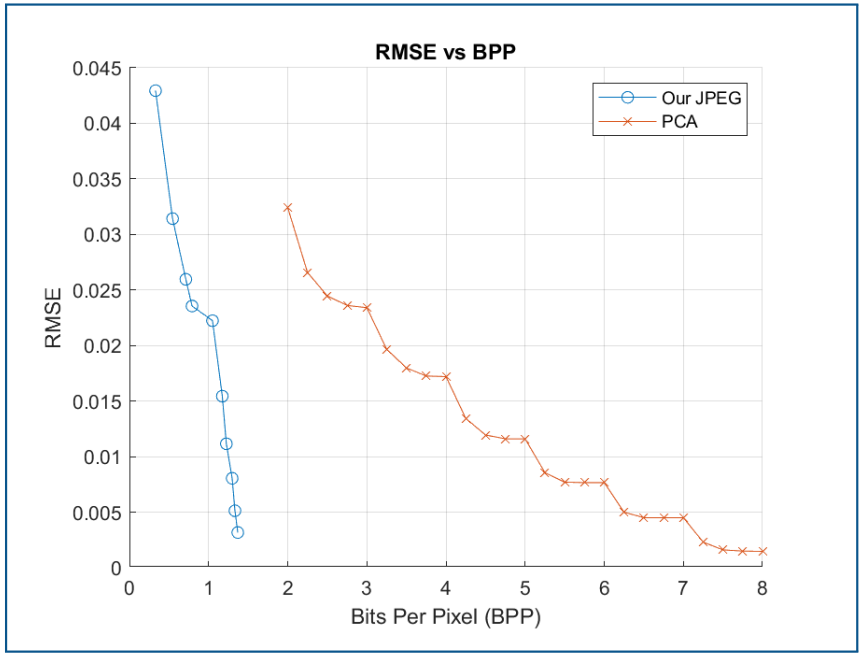
Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Image Class - Buildings

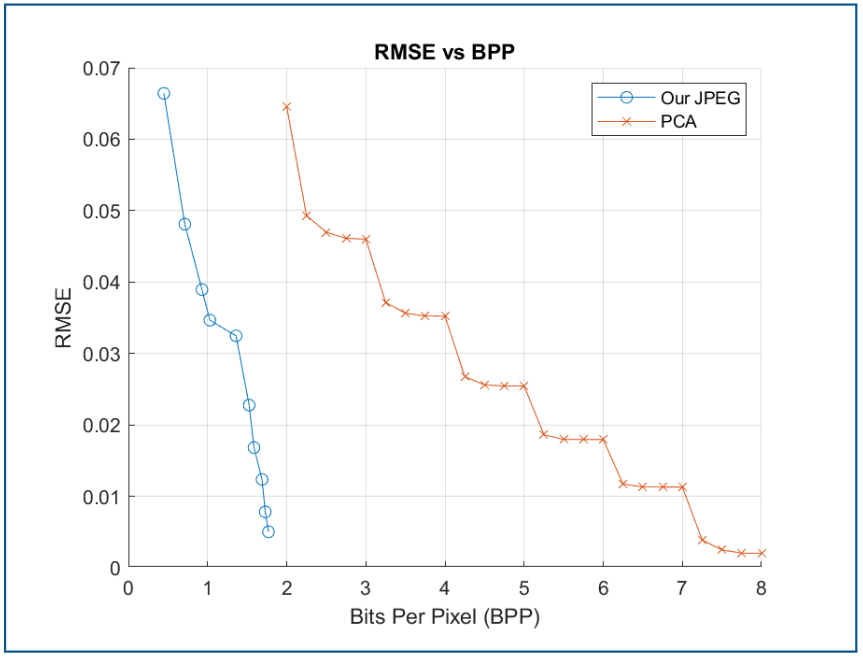


Qf = 10:10:100 & numComponents = 16:2:64

Comparison with PCA | Image Class - Buildings



Qf = 10:10:100 & numComponents = 16:2:64



Qf = 10:10:100 & numComponents = 16:2:64

PCA basis Constructed from a Set of Images

- For the *Flower* and *Cloud* classes, the high similarity among images within each class allows PCA to achieve very low RMSE values at moderate BPP values.
- For the *Building* class, the images are significantly different from one another, resulting in suboptimal PCA performance in terms of both RMSE and BPP.
- This effect is also observed with the `JPEG` engine, where the BPP for the *Flower* and *Cloud* classes is almost half that of the *Building* class.
- Note the **periodic drops** in the **RMSE** value of `PCA` each time 8 more components are considered (graph is plotted with jumps of 2), indicating that every 8 elements are similar.
- When PCA basis is reshaped to form an image, each row is similar.

7. Conclusion

Conclusion

Good Aspects

- Irrespective of the image content, the model consistently achieves 75% compression while maintaining an RMSE of less than 10%.
- The model exhibits a predictable and smooth performance curve, allowing for consistent quality improvements as the bit rate increases. This ensures it performs well across a variety of images.

Bad Aspects

- Our implementation does not adapt to the specific content of the image being compressed and simply follows a fixed algorithm. For images with overall high intensity, it could achieve better compression by scaling the quantization matrix accordingly.
- At high quality factors, our model struggles to achieve very low RMSE. To preserve quality in such cases, the model could bypass quantization and perform only Huffman encoding.

Dataset

- The dataset used for experimentation in this report is **Microsoft Resarch Cambridge Object Recognition Database**
- It includes diverse image categories, ensuring robust performance analysis across various scenarios.
- The dataset is widely used in research for benchmarking image processing techniques, including compression and recognition.
- For more information, you can access the dataset at *Microsoft Image Understanding Dataset*.