IMAGE COMPRESSION

CS663 Project Report

Authors
Brian Mackwan (Roll Number : 22b0413)
Ekansh Ravi Shankar (Roll Number : 22b1032)
Ved Danait (Roll Number : 22b1818)
Indian Institute of Technology, Bombay

Contents

1 Introduction

We implement a lossy Image Compression scheme using the JPEG algorithm and conduct some analysis of the quality of the implementation relative to CV2's implementation on metrics such as "RMSE vs BPP" and "Compression Rate vs Quality Factor".

We also perform an experiment for 5 different Quantization Matrices : JPEG Standard, Highly Detailed, Low Detail, JPEG2000-like and Uniform and conduct an analysis based on the same metrics. We also see how the compressed image behaves for different kinds of images.

We also implement a compression algorithm using PCA and compare it with our custom implementation of JPEG for our dataset of 19 grayscale images. We see that JPEG outperfoms PCA in most of these cases.

In addition, we also throughly studied, and implemented the paper, Osman Gokhan Sezer, Onur G. Guleryuz and Yucel Altunbasak, "Approximation and Compression With Sparse Orthonormal Transforms", IEEE Transactions on Image Processing, 2015, which gives us an algorithm to obtain transformations for compression by solving optimization problems (called "SOT" from now onwards). We will see how our custom SOT compares with our custom JPEG implementation.

2 Datasets

For all the JPEG analysis we have conducted in this report, we use the 19 grayscale bmp images from the website Hlevkin Test Images which are of appropriate size (dimensions multiple of 8). We could use 20, but there was some issue with Zelda2 so we left it out.

3 JPEG Implementation and Analysis

3.1 Details of the Scheme

This implementation of the JPEG compression algorithm follows the standard steps:

1. Block-based Discrete Cosine Transform (DCT).
2. Quantization using a scaled matrix based on the quality factor.
3. Encoding using Zig-Zag traversal, Run-Length Encoding (RLE), and Huffman coding.
4. Decoding involves reversing the above steps to restore the image.

Quantization Matrix and Scaling

The class `JPEG` initializes a default quantization matrix, scaled by the input quality factor `Q`.
The default quantization matrix is :

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

We scale this using the quality factor Q as :

```
self.scaled_quantization_matrix = np.round(self.quantization_matrix * (50.0 / Q))
```

Discrete Cosine Transform (DCT) and Inverse DCT

Two helper functions handle 2D DCT and inverse DCT on 8x8 image blocks:

```
def _dct2(self, block):
    return dct(dct(block.T, norm='ortho').T, norm='ortho')

def _idct2(self, block):
    return idct(idct(block.T, norm='ortho').T, norm='ortho')
```

Image Chunking and Flattening

Images are divided into 8x8 blocks for processing. After decoding, the blocks are recombined into the original shape. We do this using the functions `_chunkify(self, image)` and `_dechunkify(self, chunks, original_shape)`. Another pair of helper functions we use is `_zigzag(self, matrix)` and `_reverse_zigzag(self, flattened, rows, cols)`. This is used to flatten the quantized patch into a 1 dimensional array in order to apply RLE and Huffman Encoding. The reverse function is used while decoding when we decode in order to restore the shape of the obtained matrix after retrieving the flattened matrix while decoding the compressed file.

Run-Length Encoding (RLE) and Huffman Coding

The RLE algorithm compresses the image's non-zero coefficients, and Huffman coding is used to store them efficiently.

```
def _run_length_encoding(self, matrix):
    rle = []
    zero_count = 0
    for value in matrix:
        if value == 0:
            zero_count += 1
            if zero_count == 16:
                rle.append((15, 0))
                zero_count = 0
        else:
            rle.append((zero_count, value))
            zero_count = 0
    return rle
```

We have used a standard implementation for Huffman encoding which takes the values given after the RLE compression and assigns bitstrings to efficiently encode the values provided with respect to their frequencies.

Encoding Process

1. Quantize the DCT-transformed 8x8 blocks.
2. Apply Zig-Zag traversal.
3. Compress using RLE and encode with Huffman coding.
4. Store results in a binary file.

```
    def compress(self, image, filename):
        image = image.astype('float32') # cast to float
        image = image - 128.0 # rescale the values
        chunks = self._chunkify(image) # break into 8x8 chunks

        flattened_matrices = []
        for chunk in chunks:
            quantized_matrix = self._quantize(self._dct2(chunk))
            flattened_matrix = self._zigzag(quantized_matrix)
            flattened_matrices.append(flattened_matrix)

        self._encode(flattened_matrices, filename)
```

```
    def _encode(self, flattened_matrices, filename):
        with open(filename, 'wb') as f:
            rle_result = []

            for i in range(0,len(flattened_matrices)):
                rle = self._run_length_encoding(flattened_matrices[i])
                rle_result.extend(rle)
                rle_result.append((0,0))

            values = [value for count, value in rle_result]  # Only values for Huffman encoding
            huffman_codes = self._huffman_encoding(values)

            # Store the Quality Factor in the file
            f.write(np.array([self.Q], dtype=np.uint8).tobytes())
            f.flush()


            # Store the Huffman table in the file (value -> huffman code mapping)
            f.write(np.array([len(huffman_codes)], dtype=np.uint16).tobytes())  # Write the number of unique values in Huffman table
            f.flush()


            # Collect all Huffman codes
            all_huffman_codes = []

            for value, code in huffman_codes.items():
                f.write(np.array([value], dtype=np.int16).tobytes())
                f.flush()
                size_in_bits = len(code)
                f.write(np.array([size_in_bits], dtype=np.uint8).tobytes())  # Write the size of the Huffman code
                f.flush()
                all_huffman_codes.append(code)

            all_huffman_codes_str = ''.join(all_huffman_codes)
            total_size_in_bits = len(all_huffman_codes_str)

            f.write(np.array([total_size_in_bits], dtype=np.uint32).tobytes())
            f.flush()

            padding_length = 0
            if len(all_huffman_codes_str) % 8 != 0:
                padding_length = 8 - (len(all_huffman_codes_str) % 8)
            all_huffman_codes_str = all_huffman_codes_str + '0' * padding_length

            bitstream = bitstring.BitStream(bin=all_huffman_codes_str)
            f.write(bitstream.bytes)  # Write the Huffman code as bytes
            f.flush()

            # To store the RLE Result
            # Store the size of the rle_result
            f.write(np.array([len(rle_result)], dtype=np.uint32).tobytes())
            f.flush()

            all_huffman_codes = []
            all_huffman_codes_str = ''

            for count, value in rle_result:
                huffman_code = huffman_codes[value]  # Find Huffman code for this value
                size_in_bits = len(huffman_code)  # The size in bits of the Huffman code

                f.write(np.array([count], dtype=np.uint8).tobytes())
                f.write(np.array([size_in_bits], dtype=np.uint8).tobytes())

                f.flush()
```

```
                all_huffman_codes.append(huffman_code)

            all_huffman_codes_str = ''.join(all_huffman_codes)
            total_size_in_bits = len(all_huffman_codes_str)

            # Calculate padding to ensure byte alignment
            if total_size_in_bits % 8 != 0:
                padding_length = 8 - (total_size_in_bits % 8)
                all_huffman_codes_str += '0' * padding_length  # Add padding to the
                    bitstream
            else:
                padding_length = 0

            # Write the total size in bits to the file (before padding)
            f.write(np.array([total_size_in_bits], dtype=np.uint32).tobytes())
            f.flush()

            # Convert the padded bitstream into a byte array
            bitstream = bitstring.BitStream(bin=all_huffman_codes_str)
            f.write(bitstream.bytes)  # Write the Huffman codes as bytes
            f.flush()
```

Decoding Process

1. Decode Huffman codes and RLE data from the file.
2. Reconstruct blocks using inverse Zig-Zag and dequantization.
3. Apply inverse DCT and combine blocks to restore the image.

```
def decompress(self, filename):
        with open(filename, 'rb') as f:
            # Read the Quality Factor (Q)
            Q = np.fromfile(f, dtype=np.uint8, count=1)[0]

            # Read the Huffman table (number of entries)
            num_huffman_codes = np.fromfile(f, dtype=np.uint16, count=1)[0]

            huffman_codes = {}

            table_entries = []
            # Read each Huffman code (value, size)
            for _ in range(num_huffman_codes):
                value = np.fromfile(f, dtype=np.int16, count=1)[0]
                size_in_bits = np.fromfile(f, dtype=np.uint8, count=1)[0]
                table_entries.append((value,size_in_bits))

            length_huffman_codes = np.fromfile(f, dtype=np.uint32, count=1)[0]
            huffman_codes_bitstream = f.read((length_huffman_codes + 7) // 8)  # Read the
                total size in bytes
            huffman_codes_bitstream = bitstring.BitStream(bytes=huffman_codes_bitstream).bin

            current_pos = 0
            for value, size_in_bits in table_entries:
                huffman_codes[value] = huffman_codes_bitstream[current_pos:current_pos+
                    size_in_bits]
                current_pos+=size_in_bits


            # get the number of rle values
            num_rle_values = np.fromfile(f, dtype=np.uint32, count=1)[0]

            # 3. Decode the RLE-encoded values and sizes
            rle_result = []
```

```
            table_entries = []

            for _ in range(num_rle_values):
                count = np.fromfile(f, dtype=np.uint8, count=1)[0]
                size_in_bits = np.fromfile(f, dtype=np.uint8, count=1)[0]
                table_entries.append((count, size_in_bits))

            # print(table_entries)

            length_huffman_codes = np.fromfile(f, dtype=np.uint32, count=1)[0]
            huffman_codes_bitstream = f.read((length_huffman_codes + 7) // 8)  # Read the total size in bytes
            huffman_codes_bitstream = bitstring.BitStream(bytes=huffman_codes_bitstream).bin

            current_pos = 0
            for count, size_in_bits in table_entries:
                value = self._decode_huffman_code(huffman_codes_bitstream[current_pos:current_pos+size_in_bits], huffman_codes)
                rle_result.append((count, value))
                current_pos+=size_in_bits


            # Reconstruct the original flattened matrices from RLE
            flattened_matrices = []

            current_patch = []
            for count, value in rle_result:
                if count == 0 and value == 0:
                    current_patch = current_patch + [0]*(64 - len(current_patch))
                    flattened_matrices.append(np.array(current_patch))
                    current_patch = []
                else:
                    current_patch = current_patch + [0]*count + [value]


            # convert it back to the original patch dimensions
            for i in range(len(flattened_matrices)):

                flattened_matrices[i] = self._reverse_zigzag(flattened_matrices[i], 8, 8)

                # now de-quantize the matrix and rescale
                flattened_matrices[i] = (flattened_matrices[i] * self.scaled_quantization_matrix)
                flattened_matrices[i] = self._idct2(flattened_matrices[i]) + 128

            # combine the patches together
            restored_image = self._dechunkify(flattened_matrices, (8*int(np.sqrt(len(flattened_matrices))), 8*int(np.sqrt(len(flattened_matrices)))))
            return restored_image
```

Format of the Compressed Image

The structure of a JPEG image file is represented as follows:

- **Quality Factor (`Q`)**: A single byte representing the image compression quality.
- **Length of Huffman Table**: Specifies the number of Huffman codes. Each entry contains:
  - *Value* (`int8`): The symbol associated with the Huffman code.
  - *Size of Code* (`uint8`): Length of the Huffman code in bits.
- **Total Length of Concatenated Huffman Codes**: Specifies the combined size of all Huffman codes in bits (`uint32`).
- **Huffman Codes**: The concatenated Huffman bitstream.

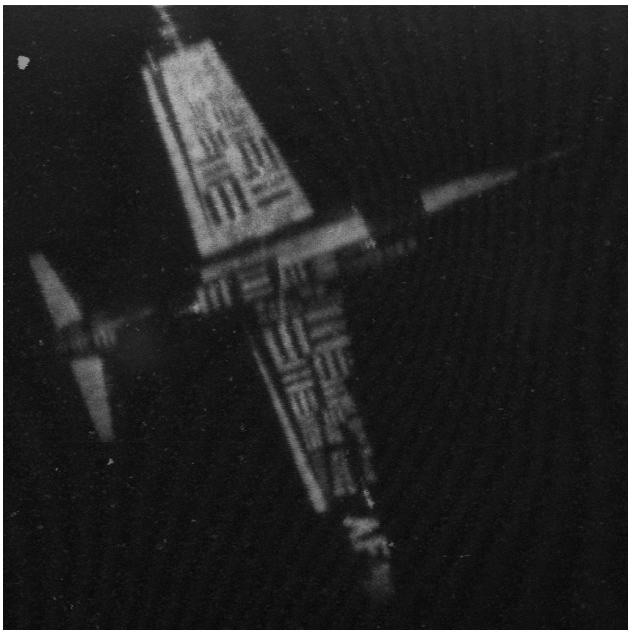
Field	Data Type
Quality Factor (`Q`)	`uint8`
Length of Huffman Table	`uint16`
- Value	`int8`
- Size of Code	`uint8`
Total Length of concatenated Huffman Codes	`uint32`
Huffman Codes	(Variable Length)
Size of Run-Length Encoded (RLE) Result	`uint32`
- Count	`uint8`
- Size of Code	`uint8`
Total Length of concatenated Huffman Codes	`uint32`
Huffman Codes	(Variable Length)

Structure of JPEG Image File

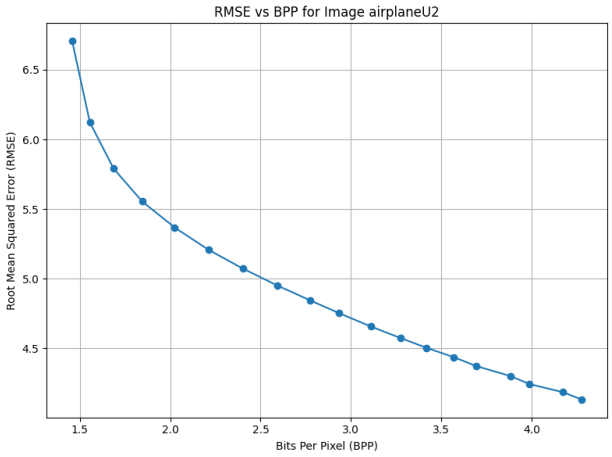
- **Size of Run-Length Encoded (RLE) Result**: Total number of RLE (count, value) pairs (`uint32`).
  - *Count* (`uint8`): Number of consecutive zeros.
  - *Size of Code* (`uint8`): Length of the corresponding Huffman code in bits.

3.2 Results

AirplaneU2

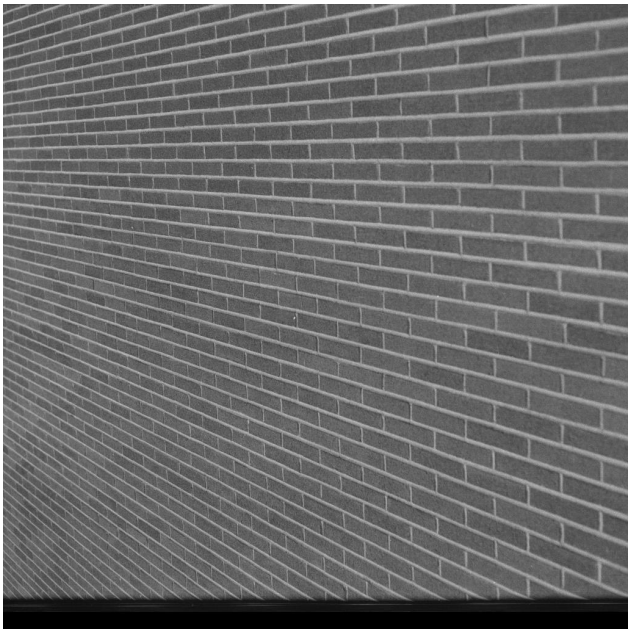


Original Image

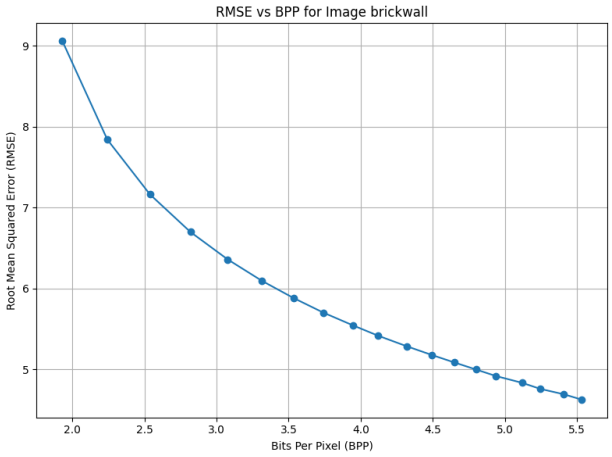


RMSE vs. BPP for varying quality factors (`Q`).

Brickwall



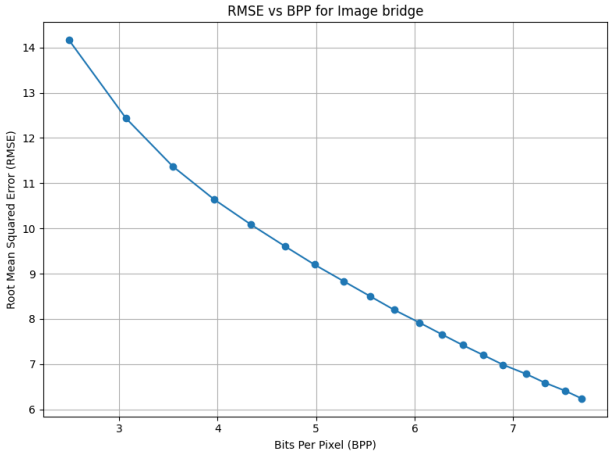
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Bridge

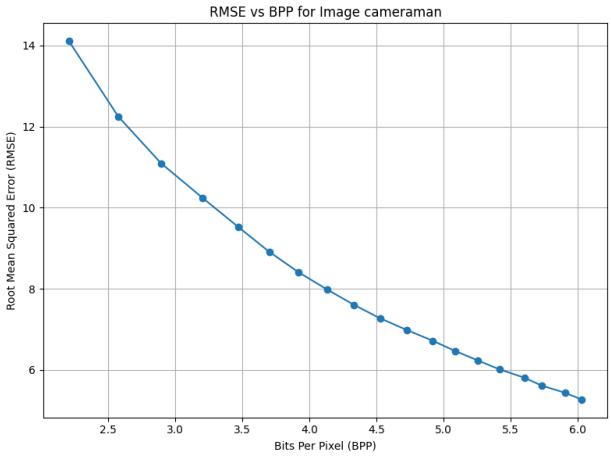
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

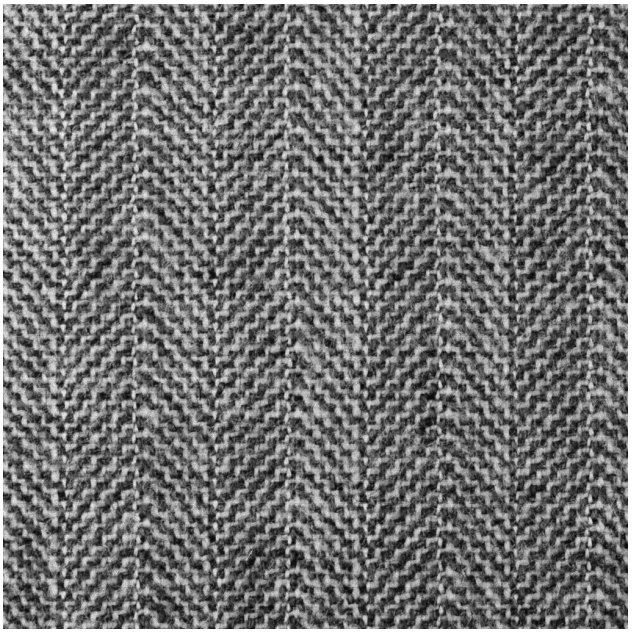
Cameraman

Original Image

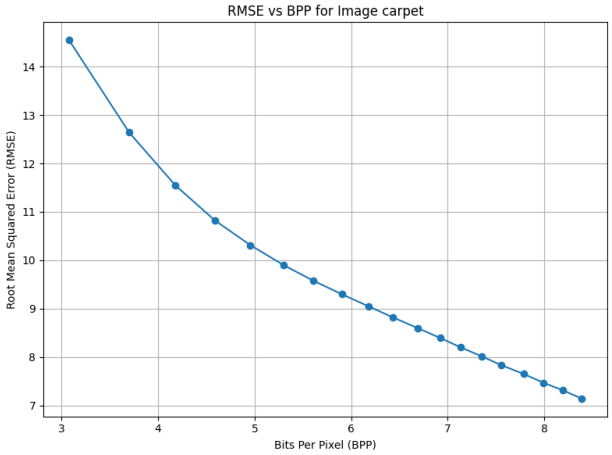


RMSE vs. BPP for varying quality factors (`Q`).

Carpet



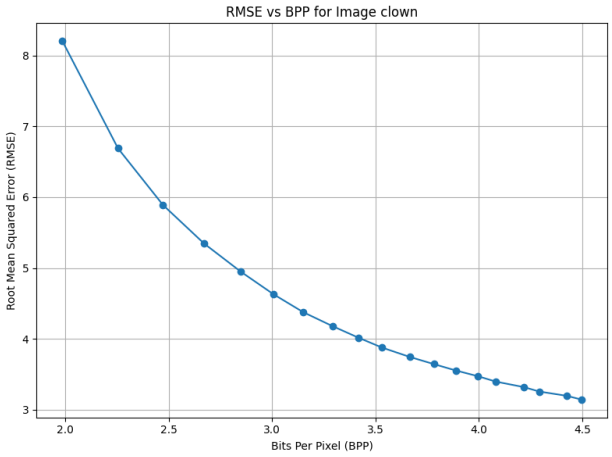
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Clown

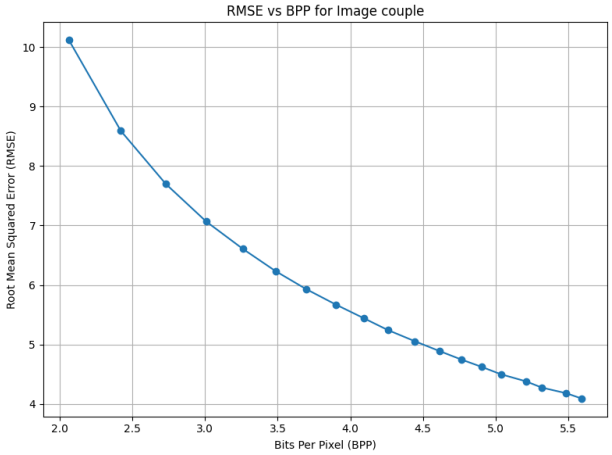
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Couple

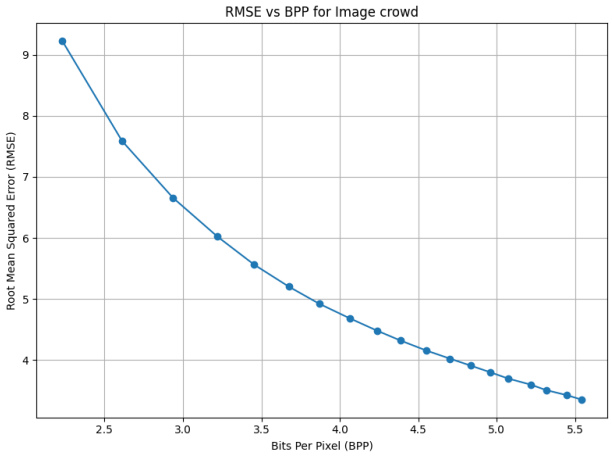
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Crowd

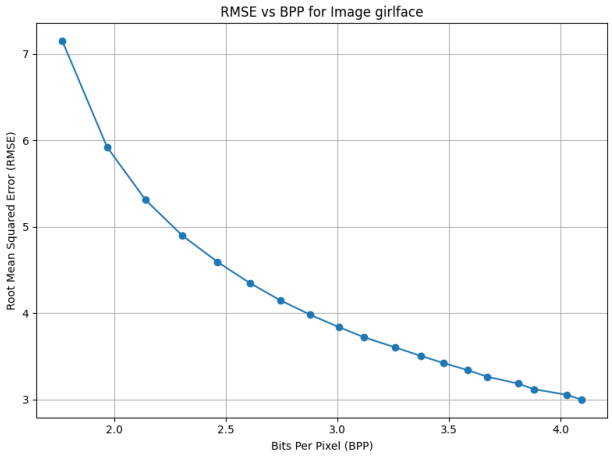
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Girlface

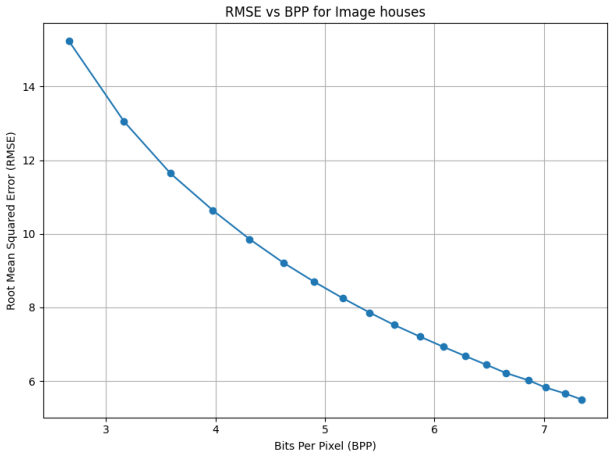
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Houses

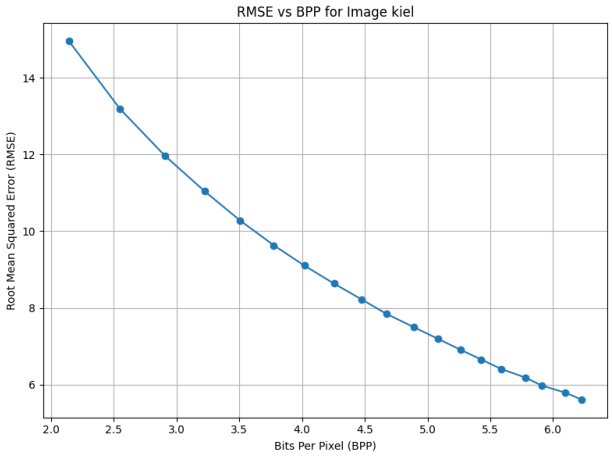
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Kiel

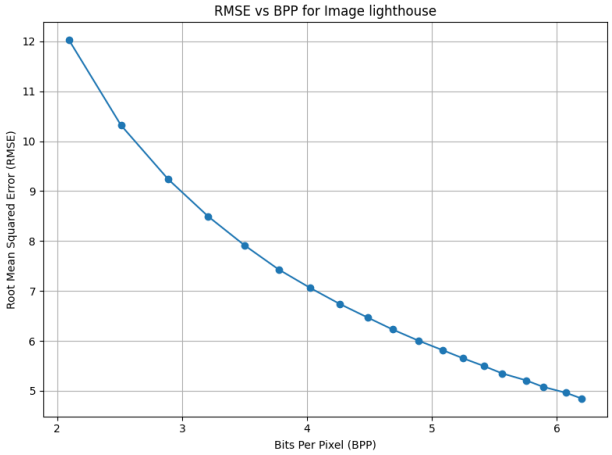
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Lighthouse

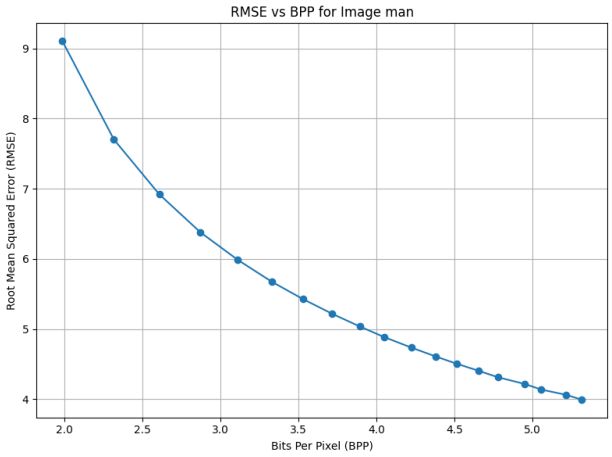
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Man

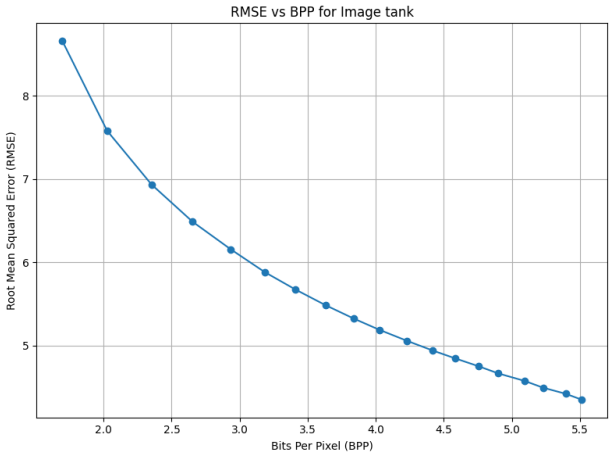
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Tank

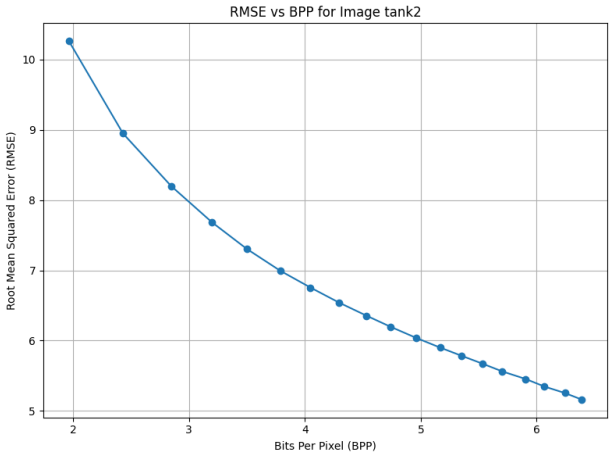
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

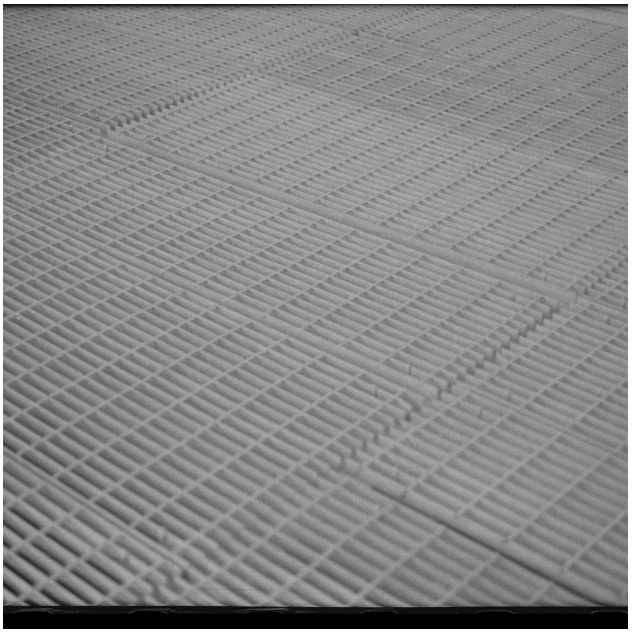
Tank2

Original Image

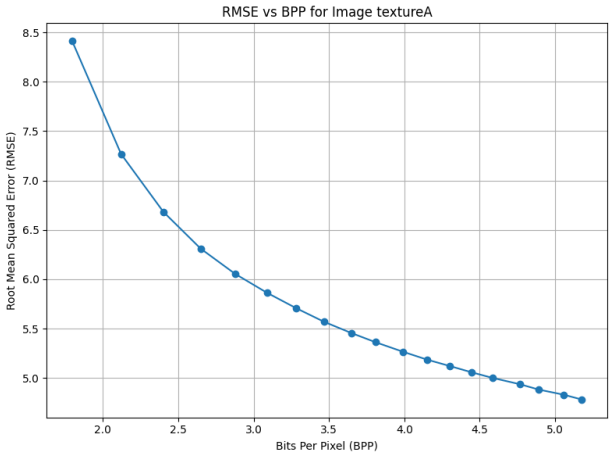


RMSE vs. BPP for varying quality factors (`Q`).

TextureA



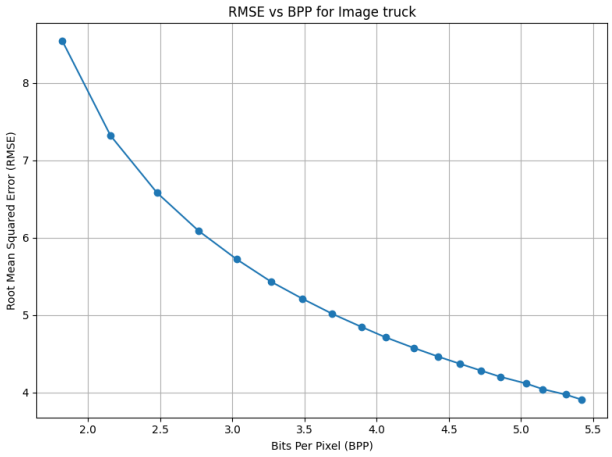
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Truck

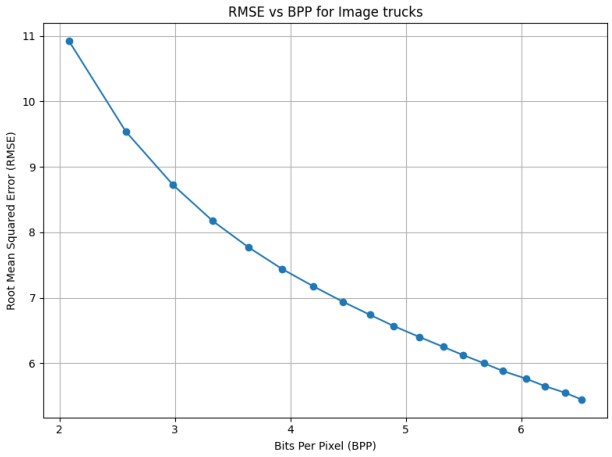
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

Trucks

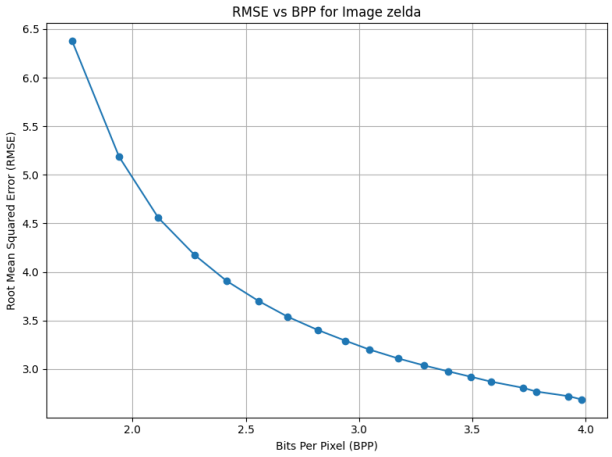
Original Image



RMSE vs. BPP for varying quality factors (`Q`).

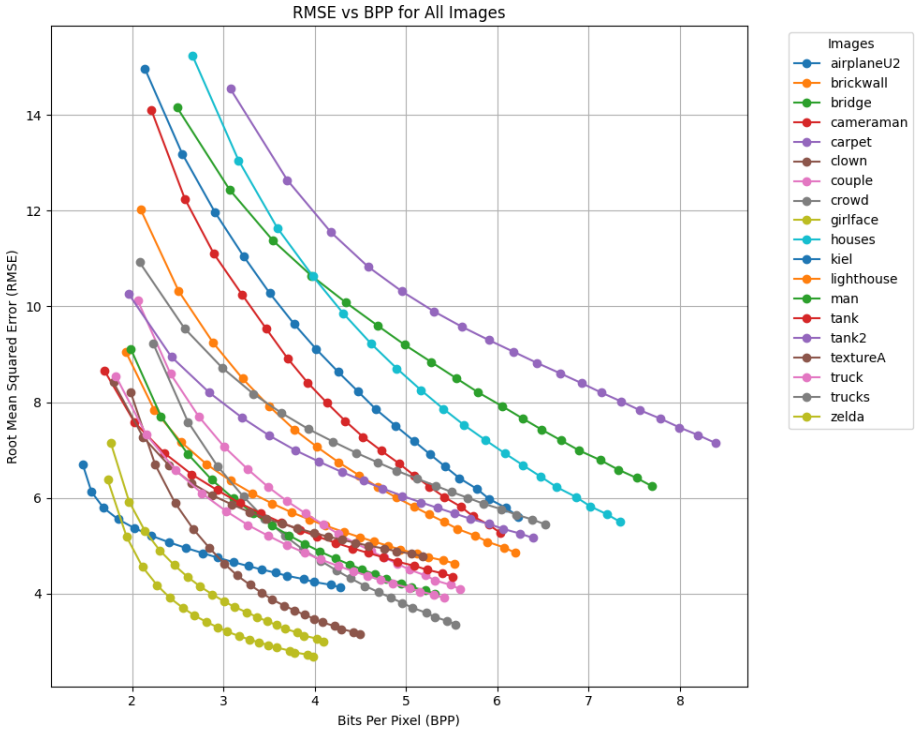
Zelda

Original Image



RMSE vs. BPP for varying quality factors (`Q`).

All Images



RMSE vs. BPP for varying quality factors (`Q`).

The graph illustrates the relationship between **Root Mean Squared Error (RMSE)** and **Bits Per Pixel (BPP)** for various images. A general trend observed across all images is that the RMSE decreases as BPP increases. This aligns with expectations, as higher BPP allows for a higher fidelity representation, reducing the reconstruction error.

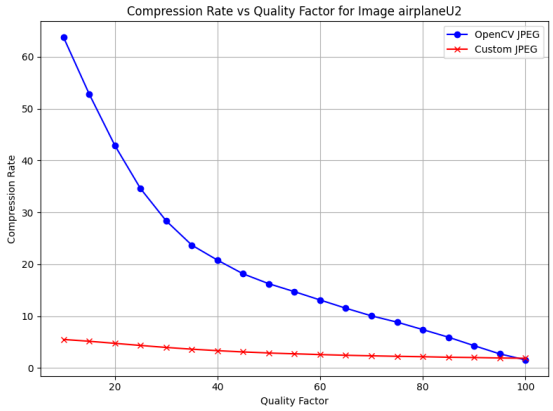
At low BPP values (e.g., BPP < 3), the RMSE decreases sharply for most images, indicating that even a modest increase in BPP leads to significant quality improvements. For example, images like *airplaneU2* and *zelda* show a steep decline in RMSE within this range, reflecting their efficient compression and reconstruction characteristics. In contrast, images with more texture or detail, such as *brickwall* and *textureA*, exhibit higher RMSE values at similar BPP, suggesting that they are more challenging to compress effectively.

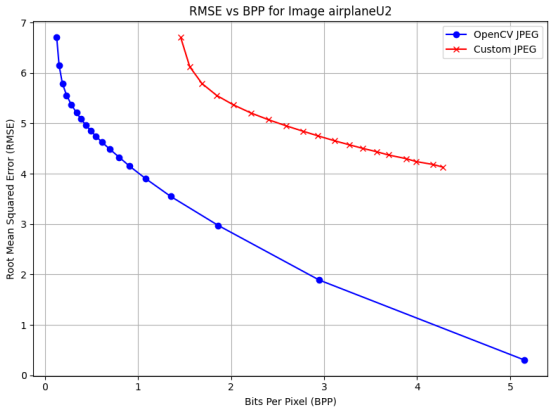
Beyond a certain threshold (e.g., BPP > 6), the curves exhibit asymptotic behavior, with RMSE improvements becoming negligible. This phenomenon demonstrates diminishing returns in quality with increased BPP, as the representation approaches near-lossless levels. For example, images like *brickwall* and *crowd* display relatively flat RMSE curves beyond BPP $= 6$, indicating that further increases in BPP would result in minimal perceptual improvement.

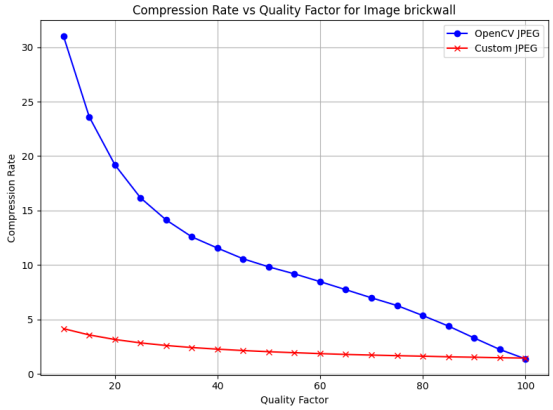
The variability between the curves highlights the impact of image characteristics on compression performance. Simpler images, such as *airplaneU2* and *zelda*, achieve lower RMSE values at the same BPP compared to more complex images like *brickwall* and *textureA*. Additionally, some images exhibit overlapping curves in the low BPP range (e.g., *clown*, *truck*, and *couple*), suggesting that certain features are handled similarly by the compression algorithm under stringent bit-budget constraints.

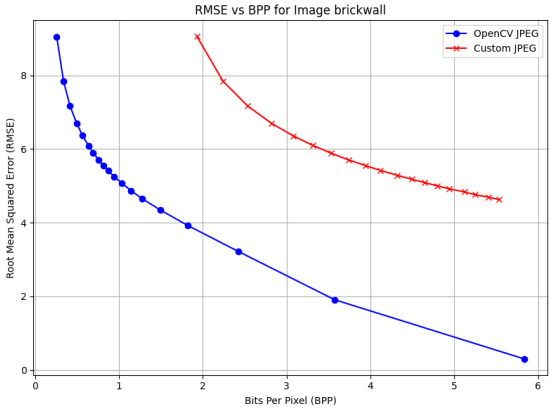
In summary, the analysis highlights the tradeoff between compression efficiency and reconstruction quality. For simpler images, lower BPP values suffice to maintain high quality, whereas textured images require higher BPP to achieve comparable RMSE. The diminishing returns beyond certain BPP values emphasize the importance of selecting an optimal bit allocation for practical applications.

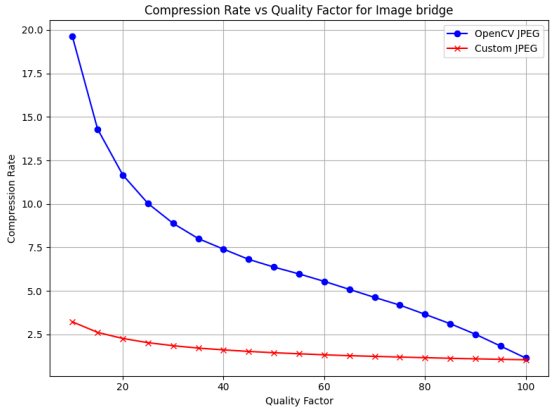
4 Comparison with CV2 Implementation

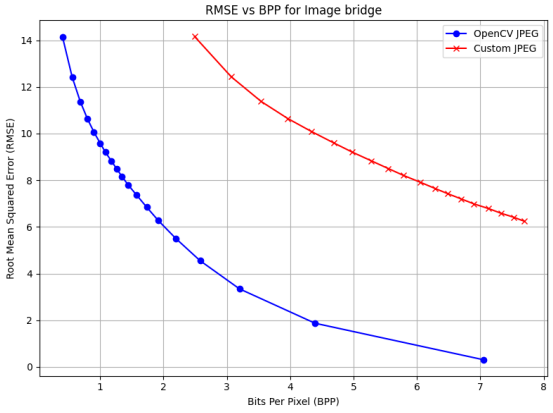


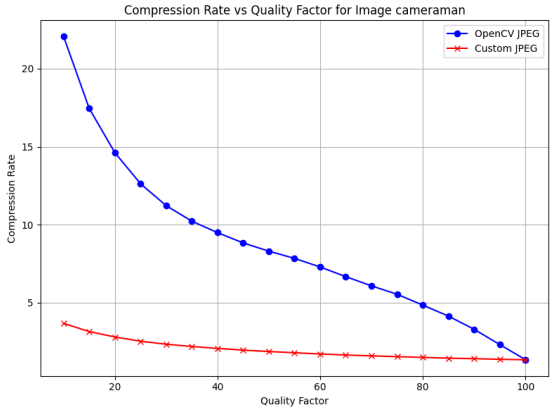


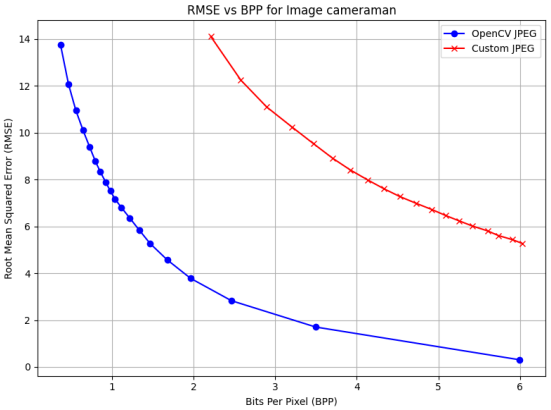


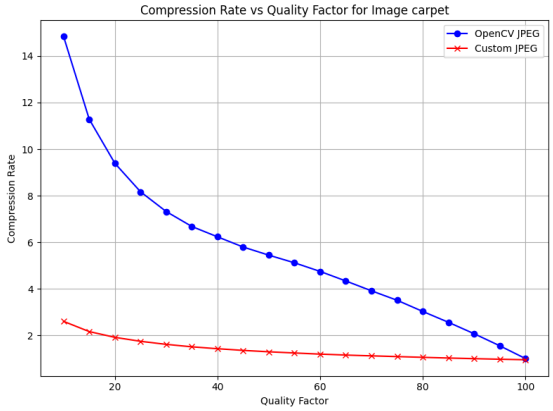


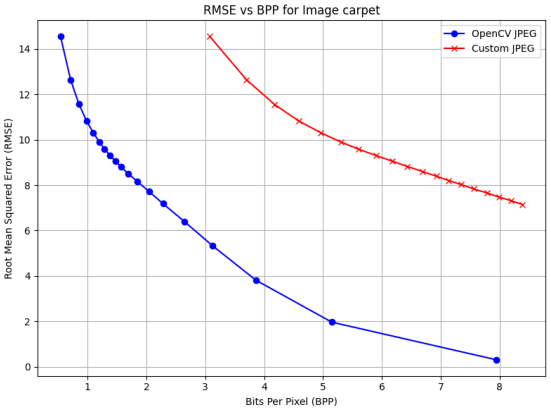


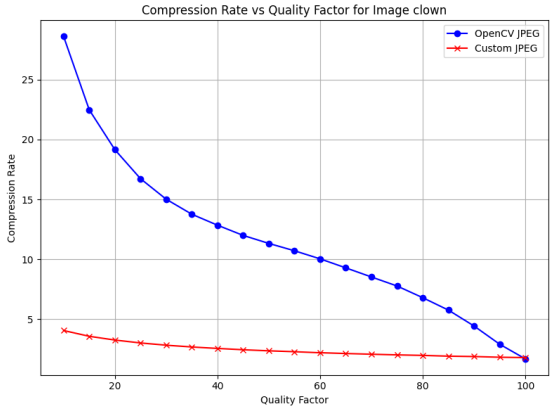


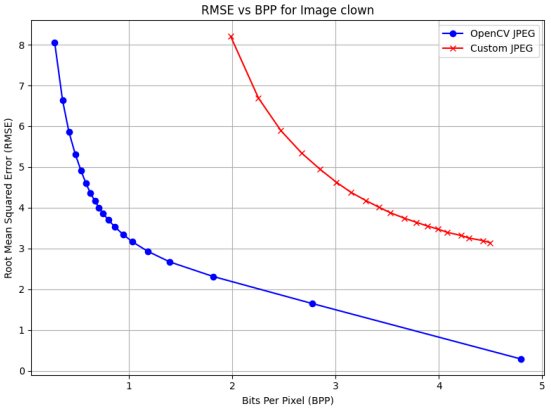

Compression Rate vs Quality Factor for Image cameraman
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

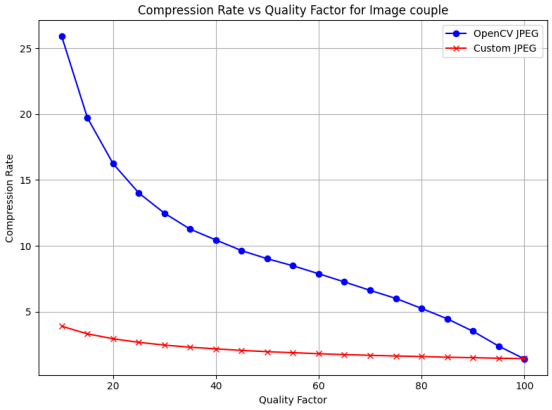

RMSE vs BPP for Image cameraman
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

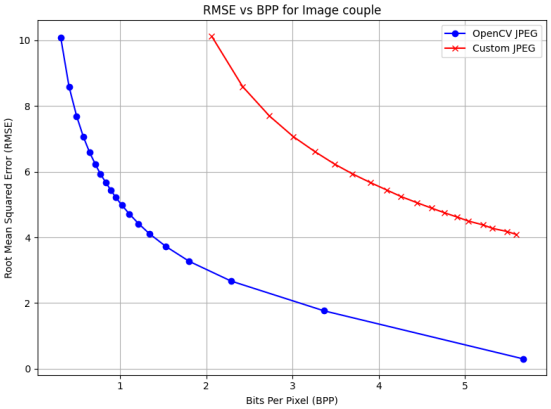

Compression Rate vs Quality Factor for Image carpet
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

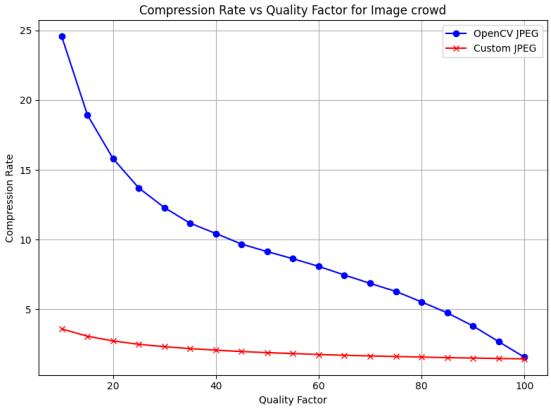

RMSE vs BPP for Image carpet
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

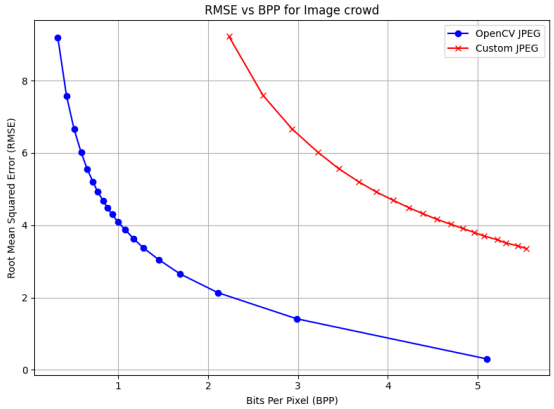

Compression Rate vs Quality Factor for Image clown
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

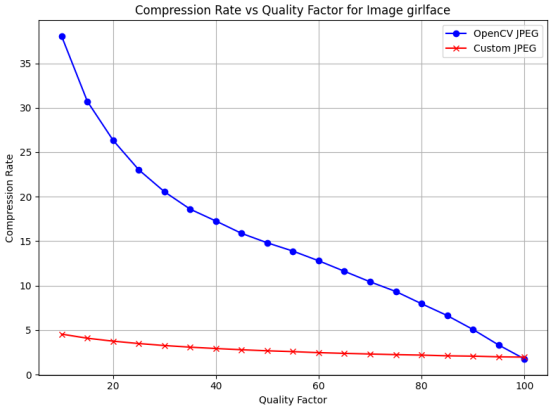

RMSE vs BPP for Image clown
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

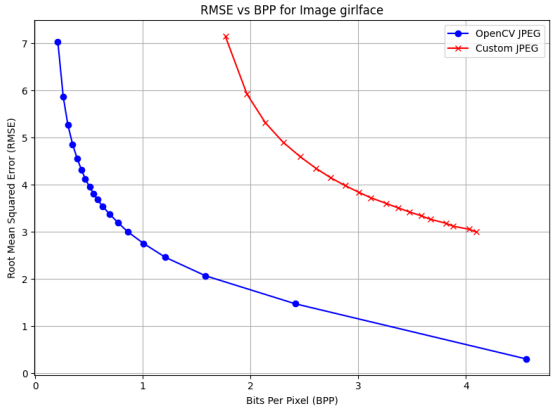

Compression Rate vs Quality Factor for Image couple
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

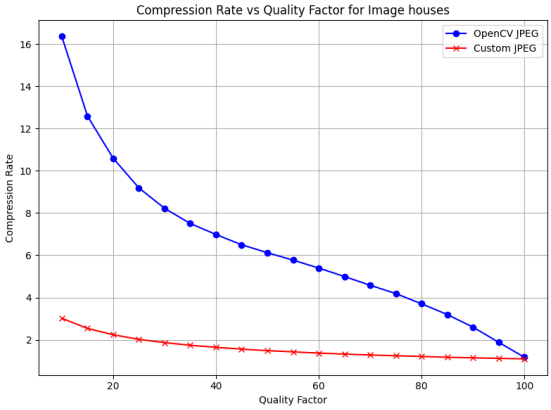

RMSE vs BPP for Image couple
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

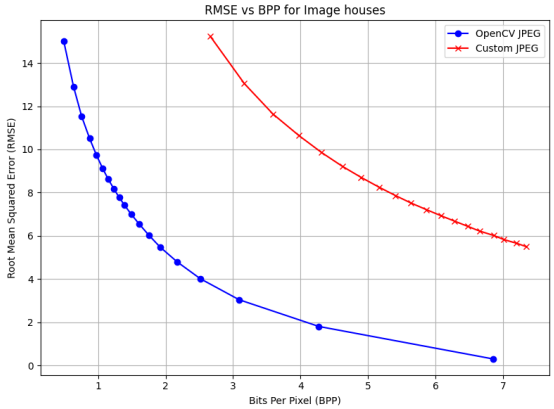

Compression Rate vs Quality Factor for Image crowd
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

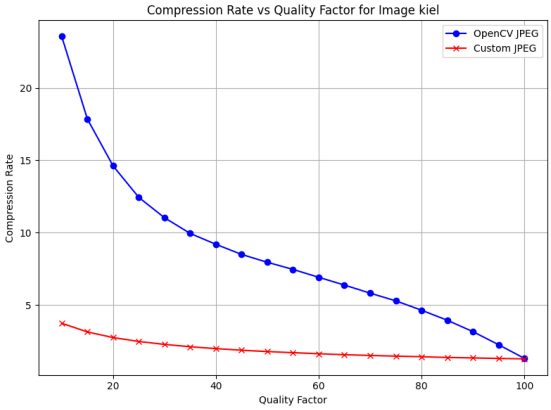

RMSE vs BPP for Image crowd
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

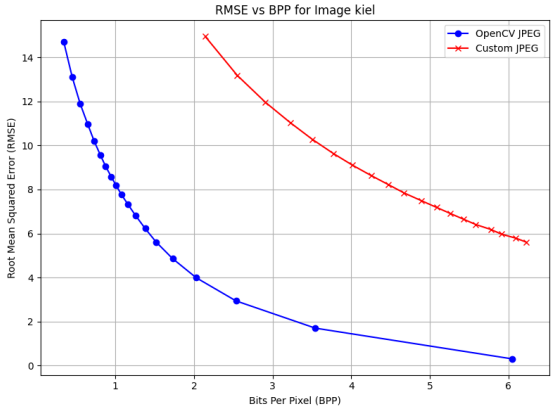

Compression Rate vs Quality Factor for Image girlface
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

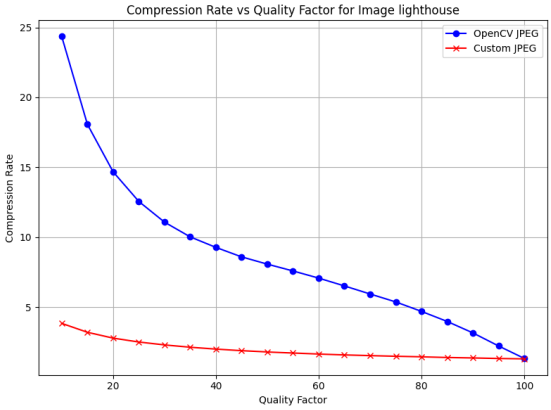

RMSE vs BPP for Image girlface
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

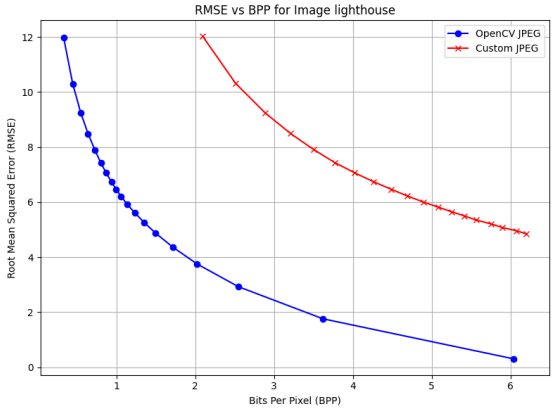

Compression Rate vs Quality Factor for Image houses
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

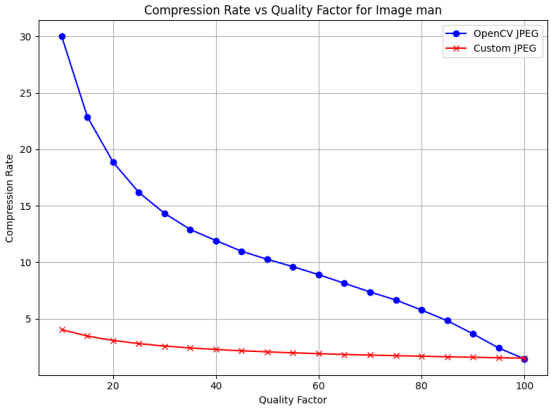

RMSE vs BPP for Image houses
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

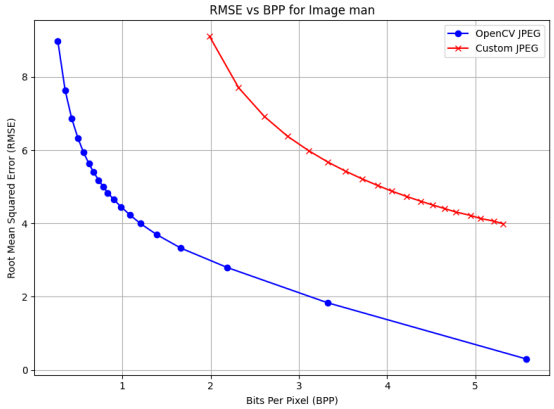

Compression Rate vs Quality Factor for Image kiel
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

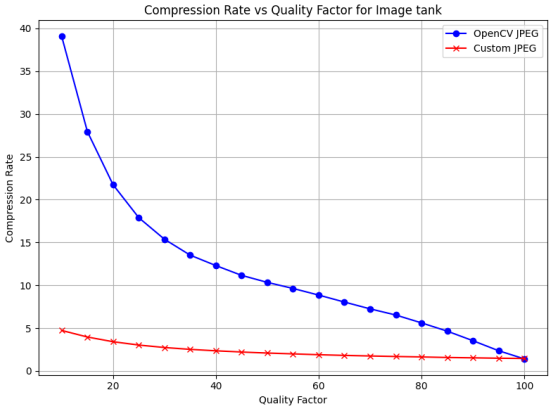

RMSE vs BPP for Image kiel
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

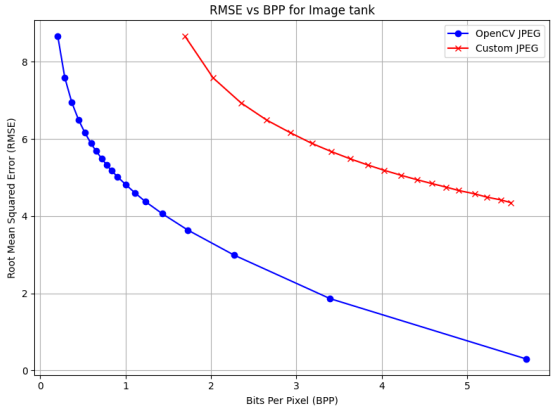

Compression Rate vs Quality Factor for Image lighthouse
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

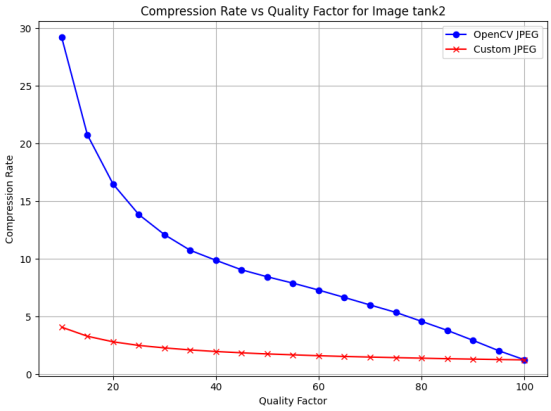

RMSE vs BPP for Image lighthouse
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

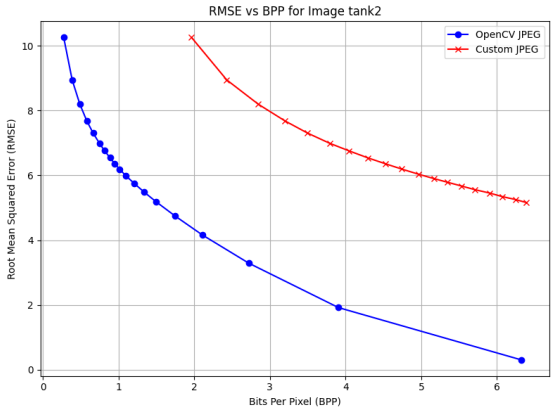

Compression Rate vs Quality Factor for Image man
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

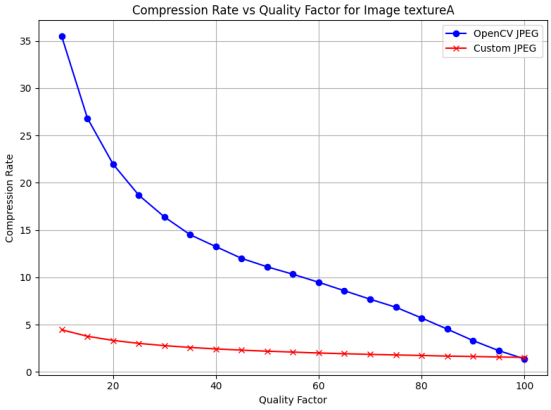

RMSE vs BPP for Image man
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

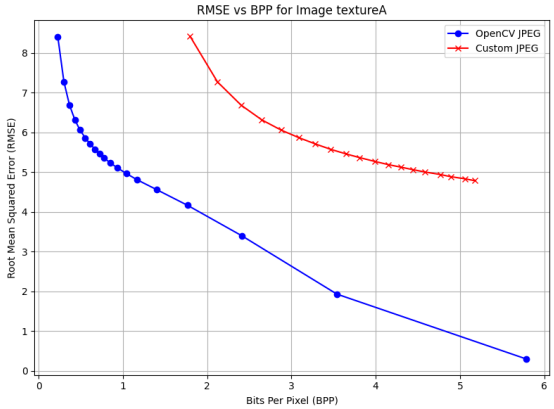

Compression Rate vs Quality Factor for Image tank
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

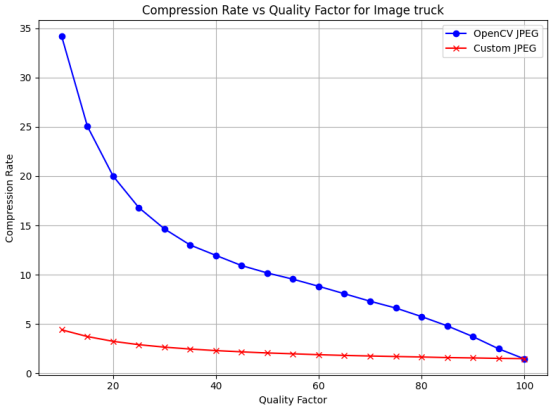

RMSE vs BPP for Image tank
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

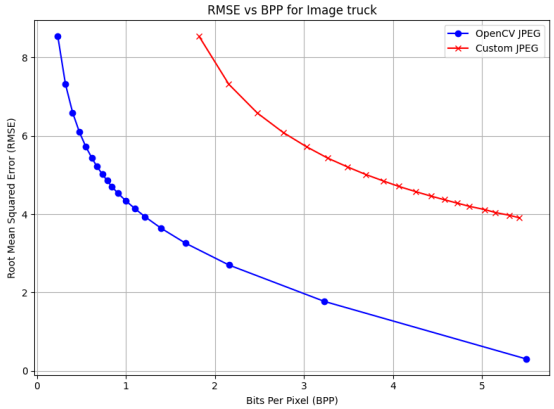

Compression Rate vs Quality Factor for Image tank2
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

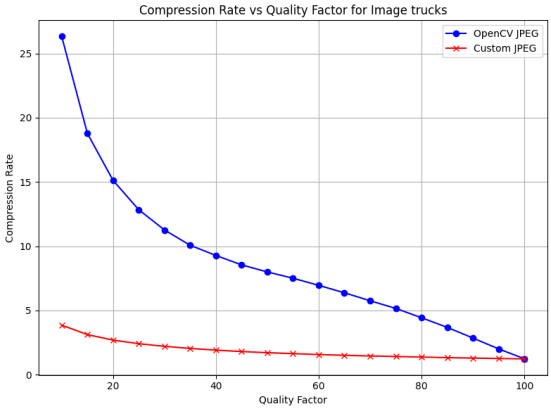

RMSE vs BPP for Image tank2
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

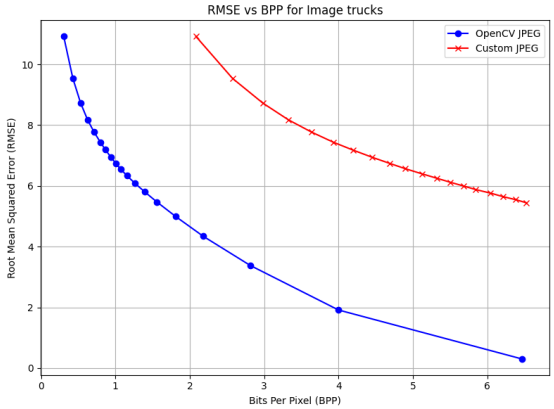

Compression Rate vs Quality Factor for Image textureA
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor

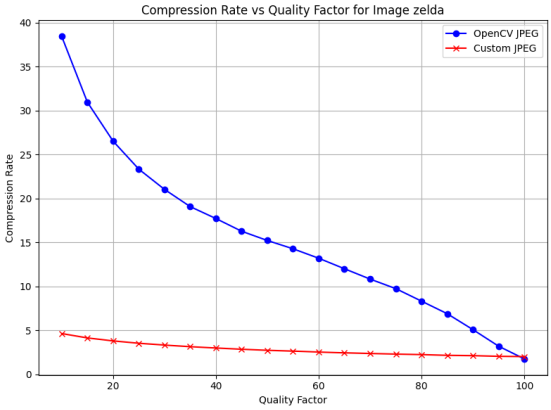

RMSE vs BPP for Image textureA
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

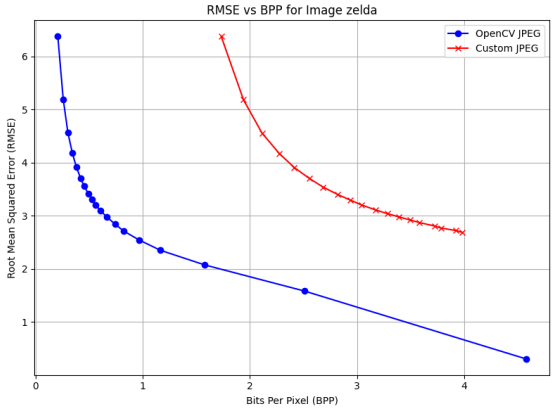

Compression Rate vs Quality Factor for Image truck
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor


RMSE vs BPP for Image truck
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)


Compression Rate vs Quality Factor for Image trucks
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor


RMSE vs BPP for Image trucks
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)


Compression Rate vs Quality Factor for Image zelda
OpenCV JPEG
Custom JPEG
Compression Rate
Quality Factor


RMSE vs BPP for Image zelda
OpenCV JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

5 Experiments with Various Quantization Matrices

We run our custom JPEG algorithm using 3 different Quantization matrices and plot the "Compression Rate vs Quality Factor" and "RMSE vs BPP" curves for the same. We also display the image at Q=50 and try and observe the variations. The matrices are as follows:

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

$$\text{HIGHLY DETAILED} = \begin{bmatrix} 4 & 3 & 3 & 4 & 6 & 10 & 13 & 15 \\ 3 & 3 & 4 & 5 & 7 & 16 & 16 & 14 \\ 4 & 4 & 5 & 7 & 10 & 15 & 18 & 15 \\ 4 & 5 & 6 & 8 & 14 & 24 & 23 & 19 \\ 6 & 7 & 11 & 17 & 20 & 32 & 30 & 23 \\ 10 & 16 & 25 & 29 & 37 & 48 & 53 & 41 \\ 13 & 16 & 20 & 23 & 28 & 56 & 55 & 46 \\ 15 & 14 & 15 & 19 & 23 & 41 & 43 & 41 \end{bmatrix}$$

$$\text{LOW DETAIL} = \begin{bmatrix} 50 & 40 & 35 & 50 & 60 & 80 & 90 & 100 \\ 40 & 40 & 50 & 60 & 70 & 100 & 110 & 90 \\ 45 & 50 & 60 & 75 & 85 & 100 & 120 & 100 \\ 50 & 60 & 70 & 90 & 110 & 150 & 140 & 120 \\ 60 & 70 & 90 & 110 & 130 & 180 & 170 & 140 \\ 80 & 100 & 130 & 140 & 160 & 200 & 210 & 160 \\ 90 & 110 & 130 & 150 & 170 & 220 & 240 & 190 \\ 100 & 120 & 140 & 160 & 180 & 220 & 240 & 210 \end{bmatrix}$$

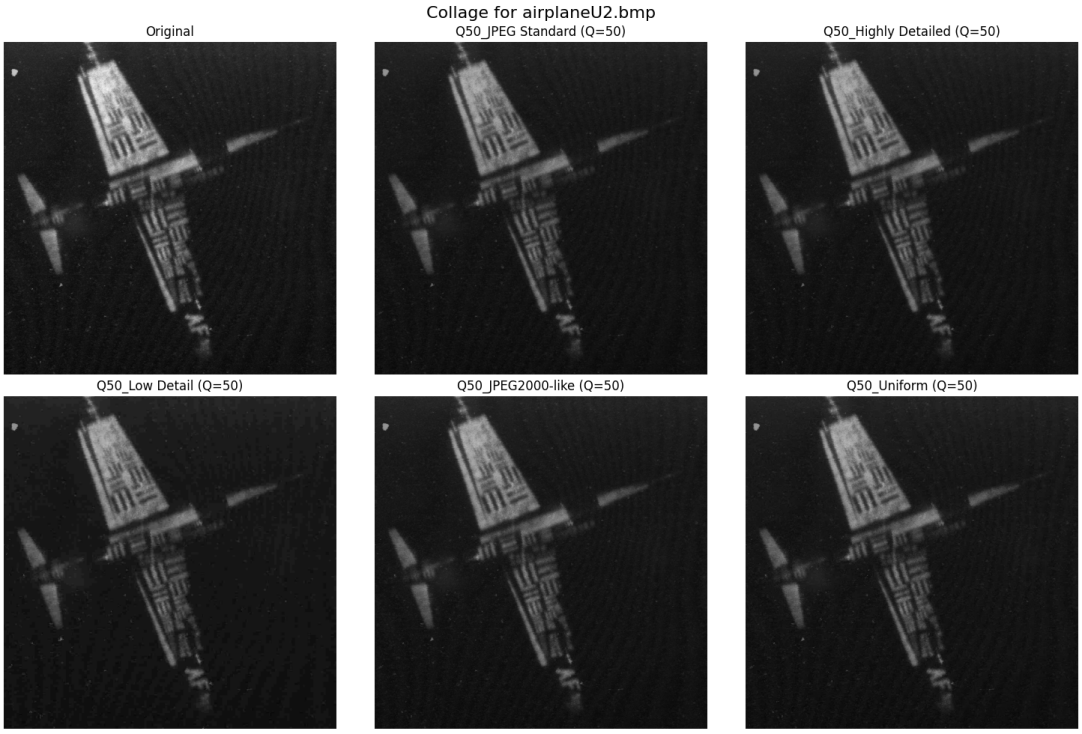
$$\text{JPEG2000 LIKE} = \begin{bmatrix} 9 & 5 & 4 & 6 & 8 & 12 & 16 & 18 \\ 5 & 5 & 6 & 8 & 10 & 18 & 19 & 16 \\ 6 & 6 & 8 & 10 & 14 & 21 & 24 & 20 \\ 6 & 8 & 10 & 13 & 17 & 27 & 26 & 21 \\ 8 & 10 & 14 & 17 & 21 & 31 & 29 & 23 \\ 12 & 18 & 21 & 24 & 30 & 39 & 42 & 32 \\ 16 & 19 & 24 & 26 & 33 & 43 & 44 & 36 \\ 18 & 16 & 20 & 21 & 26 & 36 & 39 & 35 \end{bmatrix}$$

$$\text{UNIFORM} = \begin{bmatrix} 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \\ 20 & 20 & 20 & 20 & 20 & 20 & 20 & 20 \end{bmatrix}$$

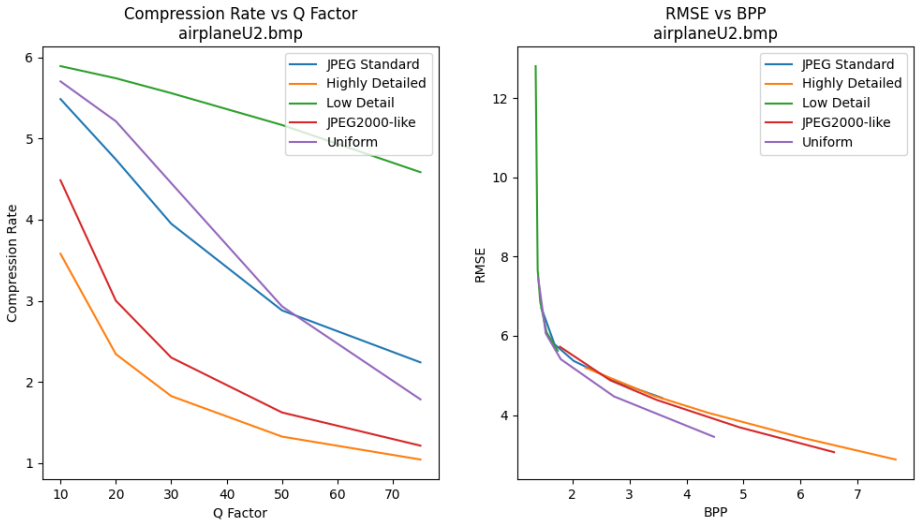
We present the results of image compression experiments for a subset of our images. Each page contains:

1. A collage of six images: the original image and Q=50 compressed images for various quantization matrices.
2. Two graphs comparing compression metrics mentioned above

AirplaneU2

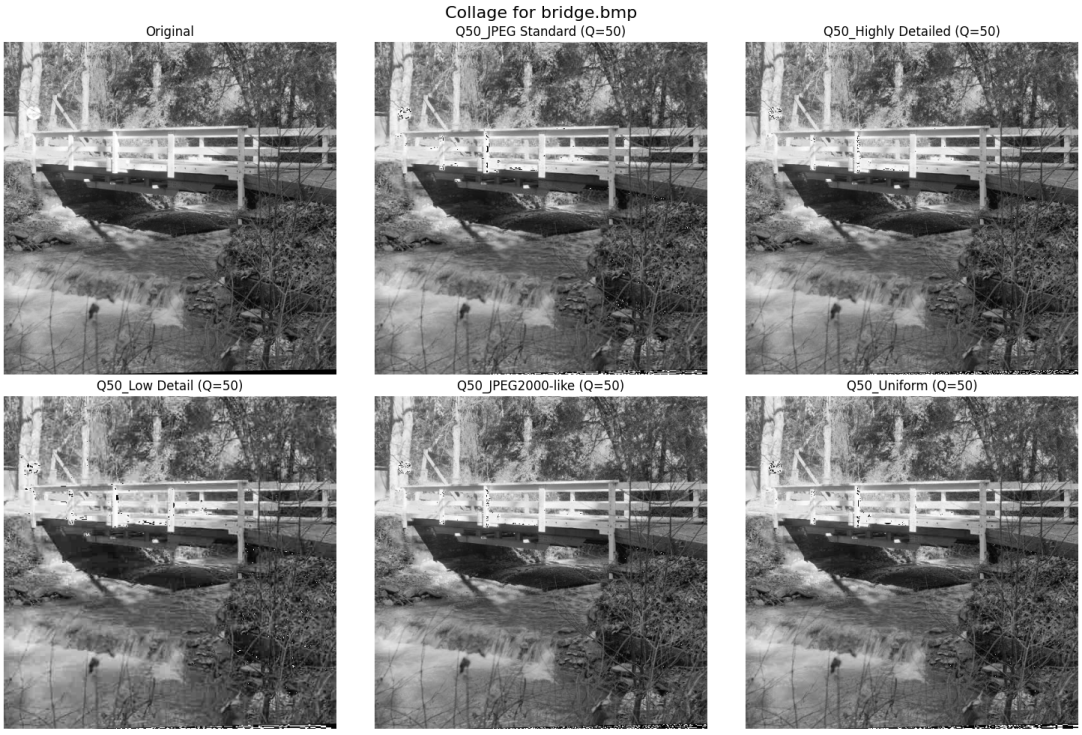


Original image and Q=50 compressed images.

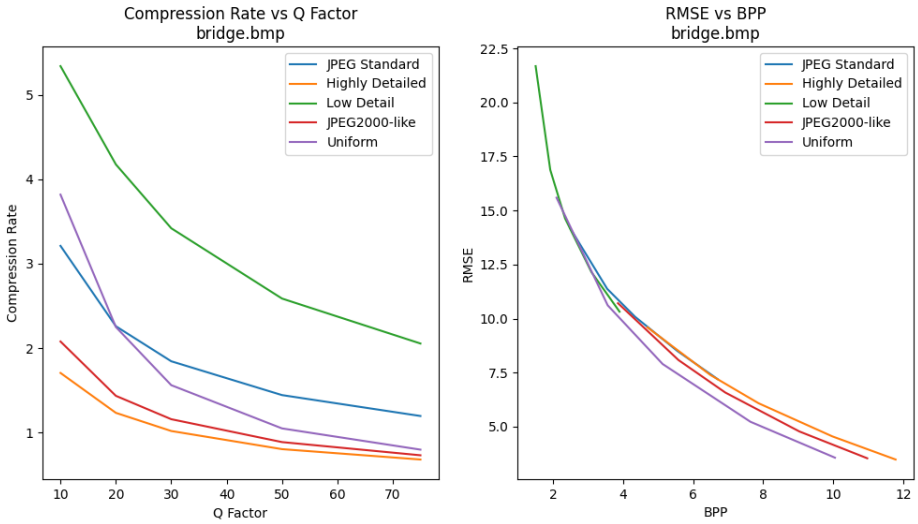


Graphs for metrics comparison.

Bridge

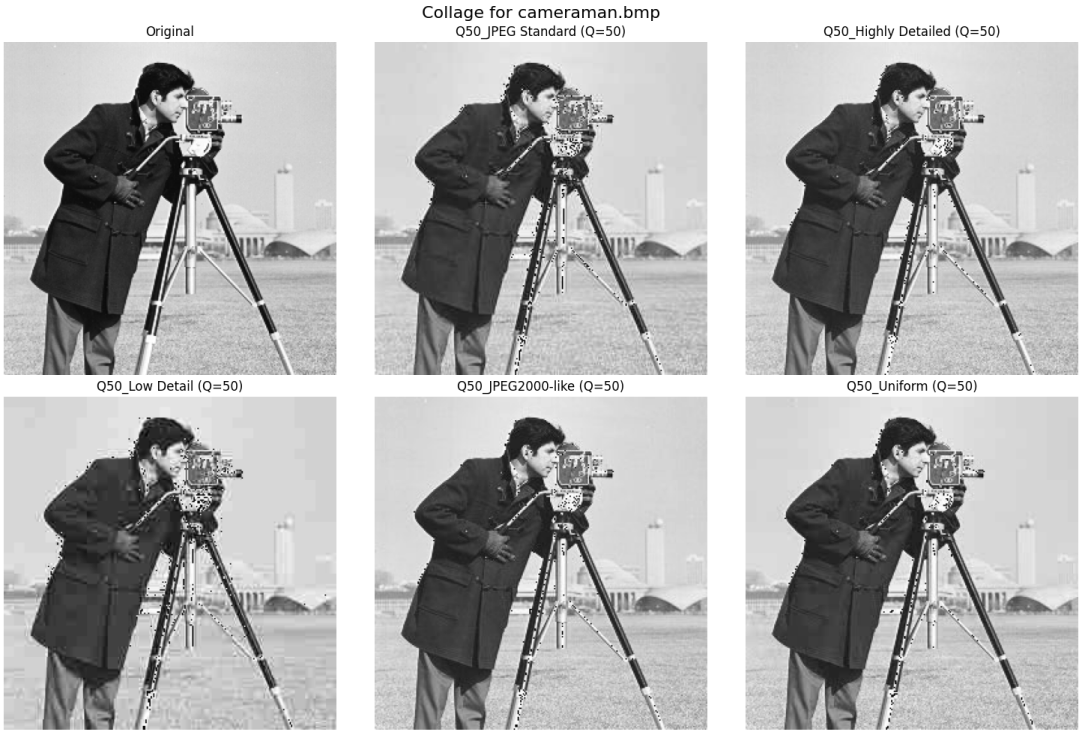


Original image and Q=50 compressed images.

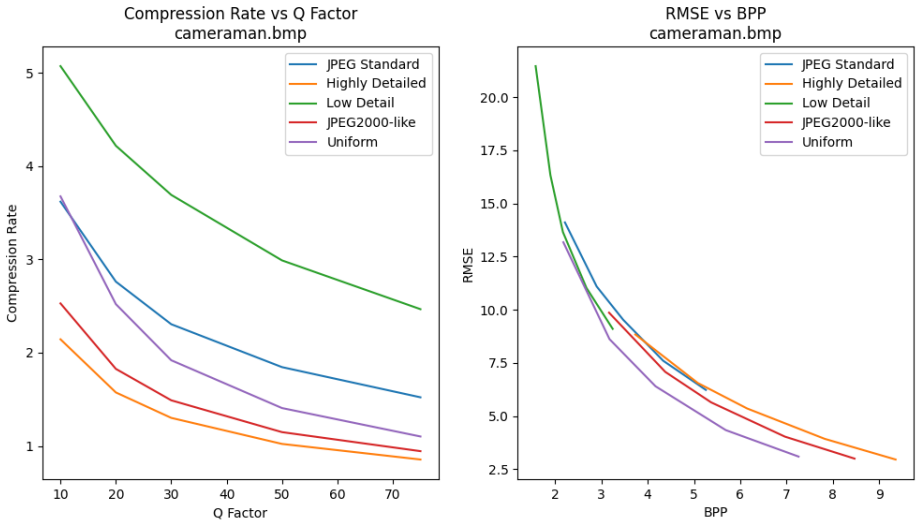


Graphs for metrics comparison.

Cameraman



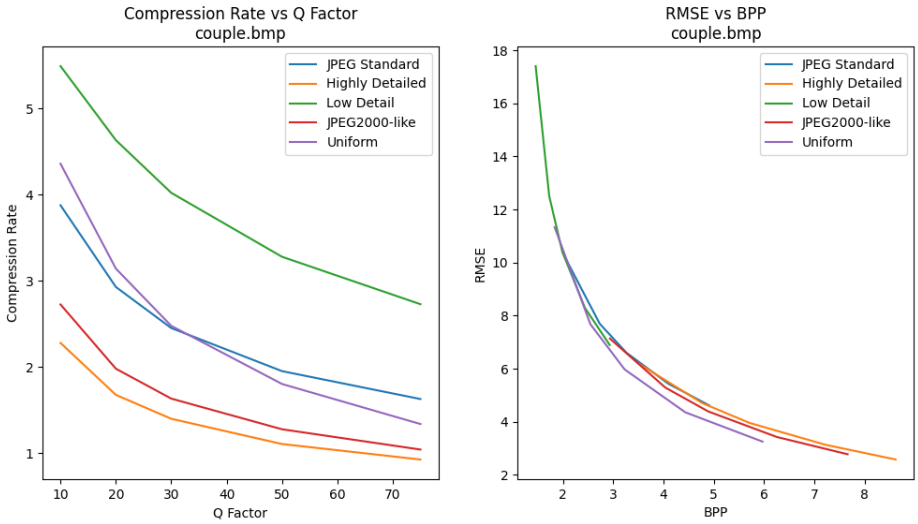
Original image and Q=50 compressed images.



Graphs for metrics comparison.

Couple

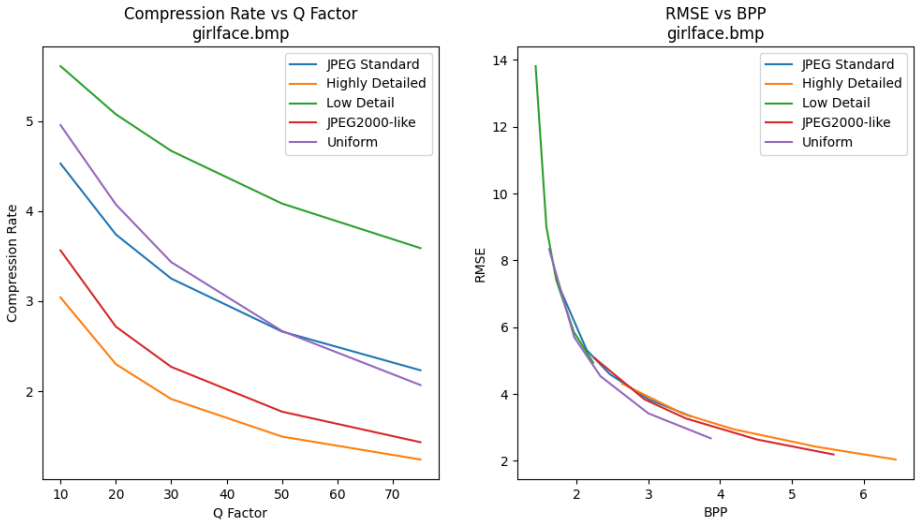
Original image and Q=50 compressed images.



Graphs for metrics comparison.

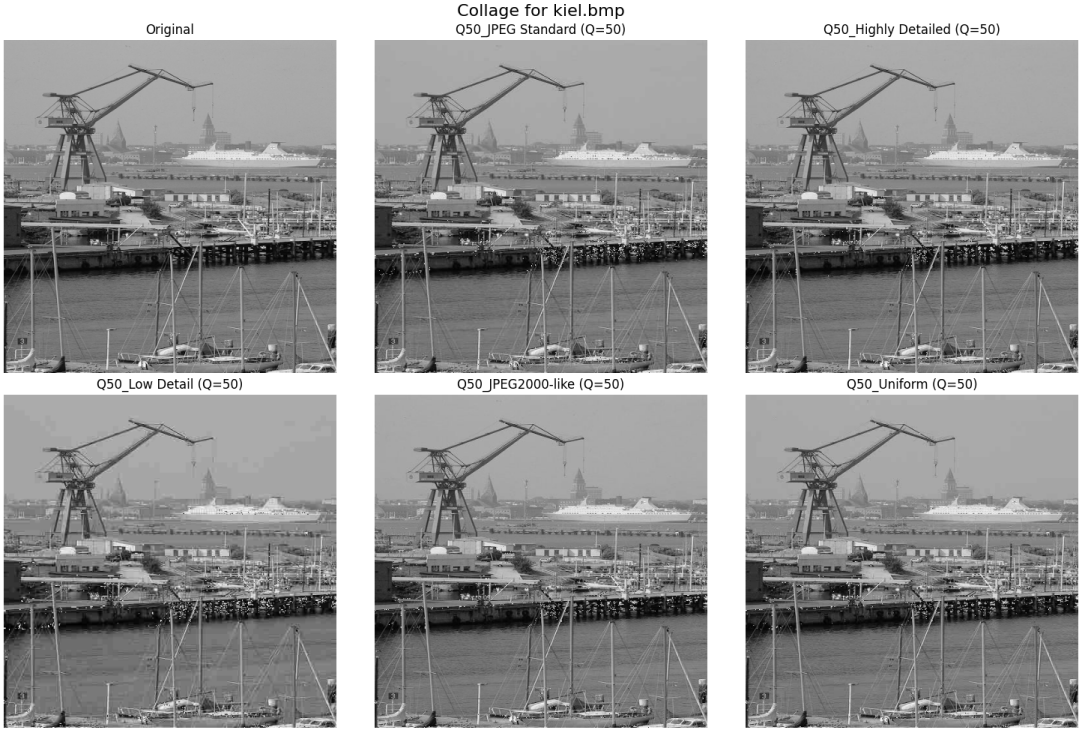
Girlface

Original image and Q=50 compressed images.

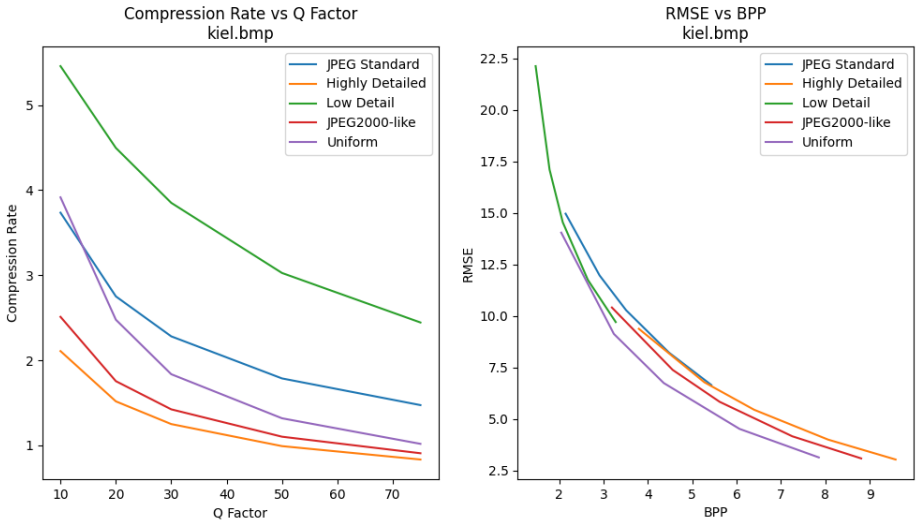


Graphs for metrics comparison.

Kiel



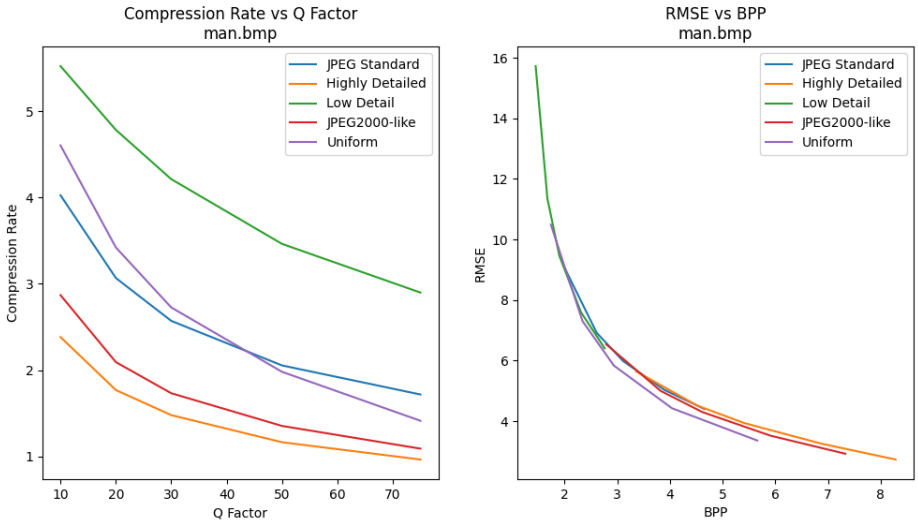
Original image and Q=50 compressed images.



Graphs for metrics comparison.

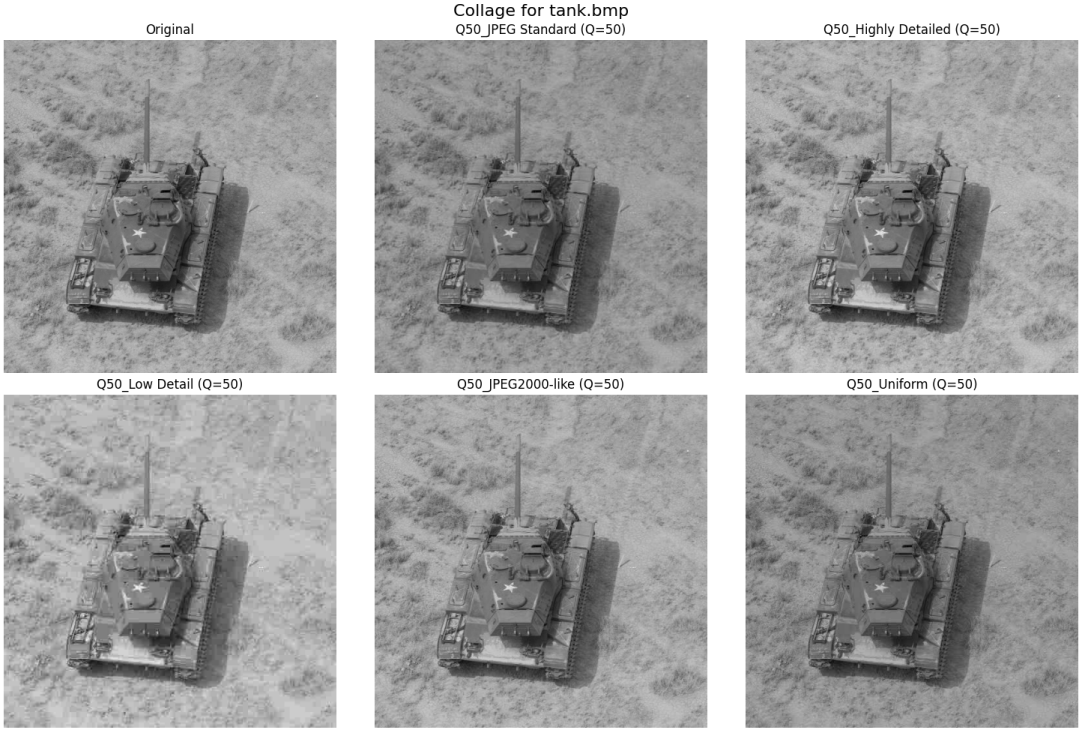
Man

Original image and Q=50 compressed images.

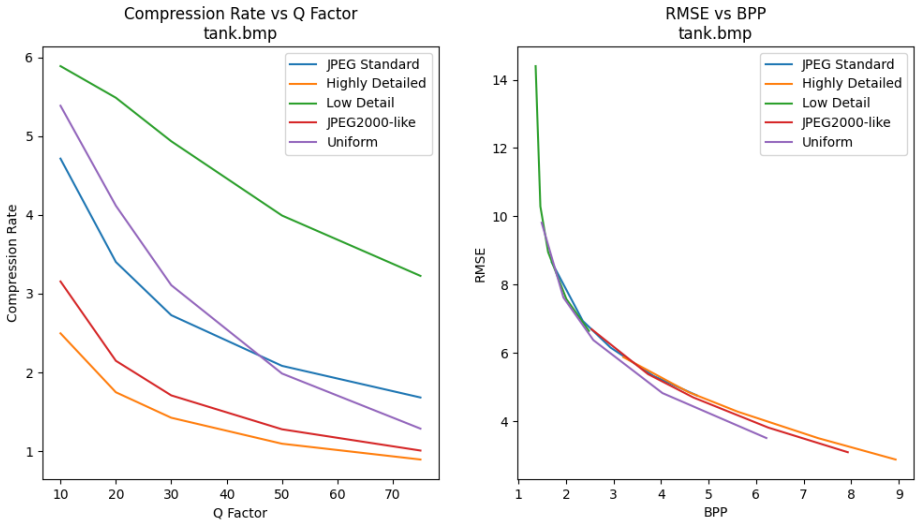


Graphs for metrics comparison.

Tank



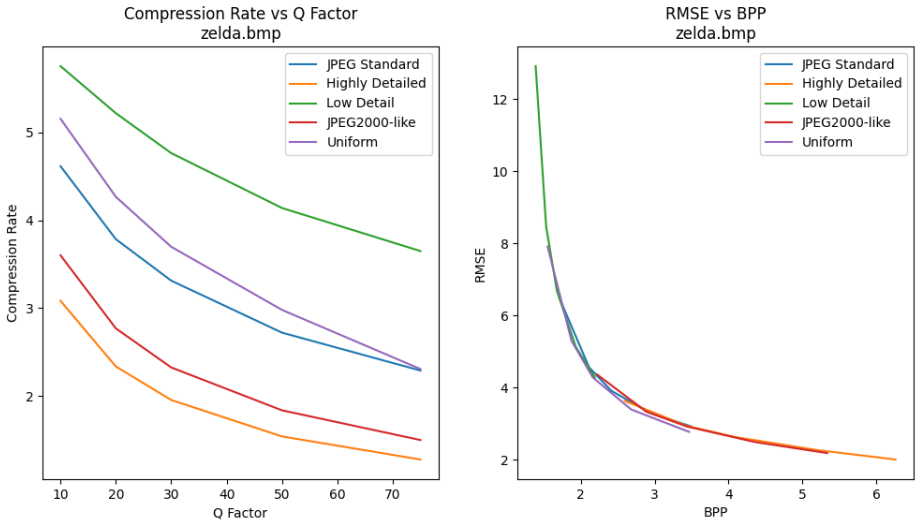
Original image and Q=50 compressed images.



Graphs for metrics comparison.

Zelda

Original image and Q=50 compressed images.



Graphs for metrics comparison.

Analysis

The plots illustrate how the choice of quantization matrix impacts compression performance and quality, highlighting the differences across matrices in achieving a balance between compression rate and reconstruction accuracy.

Compression Rate vs. Q Factor

The left plot shows the compression rate as a function of the Q factor for various quantization matrices. Across all matrices, the compression rate decreases with increasing Q factor, but the extent of this decrease varies:

- The *JPEG Standard* quantization matrix performs the best, achieving a balance between high compression rates and adaptability across Q factors. This reflects its general-purpose design and its ability to allocate bits efficiently across image regions.
- The *Low Detail* matrix exhibits higher compression rates at lower Q factors, as it aggressively removes less critical information. However, its compression advantage diminishes at higher Q factors, where the matrices converge.
- The *Highly Detailed* matrix achieves the lowest compression rates across all Q factors, as it prioritizes retaining intricate details over aggressive compression.
- The *Uniform* and *JPEG2000-like* matrices display intermediate behavior, maintaining compression rates between the extremes. They perform closer to the *JPEG Standard* matrix but exhibit slightly less efficiency.

At low Q factors (10–30), there is greater divergence in compression rates, reflecting the matrices' distinct quantization strategies. At higher Q factors (above 50), the compression rates converge, as the emphasis shifts to preserving image details uniformly across all strategies.

RMSE vs. BPP

The right plot shows how RMSE varies with BPP. As BPP increases, the RMSE decreases for all matrices, but the *JPEG Standard* matrix consistently outperforms the others:

- The *JPEG Standard* matrix achieves the lowest RMSE at nearly all BPP values, demonstrating its superior balance between quality and compression efficiency. This reflects its optimized bit allocation to minimize reconstruction error across image regions.
- The *Highly Detailed* matrix also maintains a low RMSE, especially at low BPP, as it allocates bits to preserve fine details. However, its performance slightly lags behind the *JPEG Standard*.
- The *Low Detail* matrix starts with a significantly higher RMSE at low BPP values due to its aggressive compression. Its RMSE reduces more sharply as BPP increases, but it fails to catch up with the *JPEG Standard* or *Highly Detailed* matrices at higher BPP values.
- The *Uniform* and *JPEG2000-like* matrices perform moderately well, lying between the *Low Detail* and *Highly Detailed* strategies, but do not match the *JPEG Standard*'s quality.

At higher BPP values (above 4), differences in RMSE across matrices diminish, as sufficient bits are allocated to reconstruct most of the image, irrespective of the matrix used.

Impact of Quantization Matrices

The variation across quantization matrices underscores their distinct design goals:

- The *JPEG Standard* matrix optimizes for a general-purpose tradeoff, achieving both high compression rates and low RMSE across the Q factor and BPP ranges.
- The *Low Detail* matrix prioritizes compression efficiency by discarding high-frequency information but sacrifices quality at low BPP values.
- The *Highly Detailed* matrix minimizes RMSE, particularly at low BPP, but at the cost of reduced compression rates.
- Intermediate matrices like *Uniform* and *JPEG2000-like* balance these objectives but do not surpass the *JPEG Standard* in either metric.

6 PCA Implementation and Comparison with JPEG

6.1 Details of Implementation

The `PCA` class implements Principal Component Analysis (PCA) for compressing and decompressing image data. The class performs the following key operations:

1. `fit` method: Trains the PCA model on input data by computing the mean, covariance matrix, and eigenvectors.
2. `compress` method: Compresses the data by projecting it onto the top principal components.
3. `decompress` method: Reconstructs the original data by applying the inverse of the transformation.

Initialization (`__init__`)

The initialization method accepts the number of components (k) to retain in the compression. It sets this value in the class and prepares the PCA model.

```
def __init__(self, n_components):
    self.n_components = n_components
```

Explanation: The constructor initializes the PCA model with a user-defined number of components. This value will be used during the dimensionality reduction (compression) process.

Fit Method (`fit`)

The `fit` method computes the principal components of the input data, which involves centering the data, calculating the covariance matrix, and extracting eigenvectors and eigenvalues.

```
def fit(self, data):
    # Center the data by subtracting the mean
    num_images, num_pixels = data.shape
    self.mean = np.mean(data, axis=0)
    centered_data = data - self.mean

    # Calculate the covariance matrix
    cov_matrix = np.cov(centered_data, rowvar=False)

    # Compute the eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

    # Sort eigenvalues and eigenvectors in descending order
    sorted_indices = np.argsort(eigenvalues)[::-1]
    self.eigenvalues = eigenvalues[sorted_indices]
    self.eigenvectors = eigenvectors[:, sorted_indices]

    # Select the top n_components eigenvectors
    self.components = self.eigenvectors[:, :self.n_components]
```

Explanation: - The data is centered by subtracting the mean of each feature (pixel). - The covariance matrix of the centered data is calculated to capture the variance and correlation between features. - Eigenvalues and eigenvectors are computed using `np.linalg.eigh`, which is specifically suited for symmetric matrices (like covariance matrices). - The eigenvectors are sorted in descending order based on their corresponding eigenvalues, which represent the variance explained by each principal component. - The top $n_{\text{components}}$ eigenvectors are selected as the principal components.

Mathematically, the process involves:

$$\text{Centering: } \mathbf{X}_{\text{centered}} = \mathbf{X} - \mu$$

where μ is the mean of the data matrix $\mathbf{X}$.

The covariance matrix Σ is calculated as:

$$\Sigma = \frac{1}{n}\mathbf{X}_{\text{centered}}^{\top}\mathbf{X}_{\text{centered}}$$

The eigenvalue decomposition of the covariance matrix yields the eigenvectors $\mathbf{v}$ and eigenvalues λ:

$$\Sigma\mathbf{v}_i = \lambda_i\mathbf{v}_i$$

where λ_i is the eigenvalue corresponding to the eigenvector $\mathbf{v}_i$.

Compress Method (`compress`)

The `compress` method reduces the dimensionality of the data by projecting it onto the principal components. It also saves the compressed data, components, and mean to a binary file for future use.

```
def compress(self, data, output_file="compression_output.npy"):
    # Center the data (subtract the mean)
    centered_data = data - self.mean

    # Compress the data by projecting it onto the components
    compressed_data = np.dot(centered_data, self.components)

    compressed_data = compressed_data.astype("int16")
    self.components = self.components.astype("float16")
    self.mean = self.mean.astype("uint8")

    # Store the data into a binary file using np.save
```

```
    with open(output_file, 'wb') as f:
        np.save(f, compressed_data)  # Save compressed data
        np.save(f, self.components)  # Save components
        np.save(f, self.mean)        # Save mean
```

Explanation: - The data is centered by subtracting the mean, similar to the `fit` method. - The data is projected onto the principal components by performing a dot product with the eigenvectors:

$$\mathbf{X}_{\text{compressed}} = \mathbf{X}_{\text{centered}} \cdot \mathbf{V}$$

where $\mathbf{V}$ is the matrix of the top $n_{\text{components}}$ eigenvectors. - The compressed data is then cast into a lower precision format to save memory space, specifically converting to `int16`, `float16`, and `uint8` for the compressed data, components, and mean, respectively. - The `np.save` function is used to save the compressed data, components, and mean to a binary file.

Decompress Method (`decompress`)

The `decompress` method reconstructs the original data from the compressed data by applying the inverse transformation (dot product with the transpose of the components and adding the mean).

```
def decompress(self, file_name="compression_output.npy"):
    # Open the file in read mode and load the data
    with open(file_name, 'rb') as f:
        compressed_data = np.load(f)  # Load compressed data
        components = np.load(f)       # Load components
        mean = np.load(f)             # Load mean

    # Reconstruct the data
    decompressed_data = np.dot(compressed_data, components.T) + mean
    return decompressed_data
```

Explanation: - The compressed data, components, and mean are loaded from the binary file. - To reconstruct the data, the compressed data is multiplied by the transpose of the components matrix:

$$\mathbf{X}_{\text{reconstructed}} = \mathbf{X}_{\text{compressed}} \cdot \mathbf{V}^T + \mu$$

where $\mathbf{V}^T$ is the transpose of the matrix of principal components, and μ is the mean.

6.2 Comparison for Original Dataset

The graphs compare the performance of PCA and a custom implementation of the JPEG compression algorithm in terms of RMSE versus BPP. The x-axis represents the BPP, which indicates the amount of information retained after compression. The y-axis denotes the RMSE, a measure of reconstruction error. A lower RMSE signifies higher image fidelity after compression. The two curves, plotted for PCA and the custom JPEG algorithm, reveal distinct characteristics of the two approaches to compression.

Performance of PCA

The blue curve represents the PCA-based compression. Initially, at very low BPP values, PCA shows very high RMSE, indicating poor reconstruction quality. However, as BPP increases, the RMSE drops significantly. PCA demonstrates a smooth curve, reflecting its ability to distribute the retained information across principal components effectively. The PCA curve converges to a relatively low RMSE at higher BPP, but it requires significantly higher BPP compared to JPEG to achieve comparable reconstruction quality.

Performance of Custom JPEG

The red curve represents the custom JPEG implementation. JPEG compression achieves a consistently lower RMSE than PCA at equivalent BPP values. This result highlights the efficiency of JPEG in leveraging the Discrete Cosine Transform (DCT) and quantization tables to optimize compression while retaining perceptual quality. The Custom JPEG curve flattens earlier than PCA, indicating that it achieves near-optimal quality at lower BPP values.

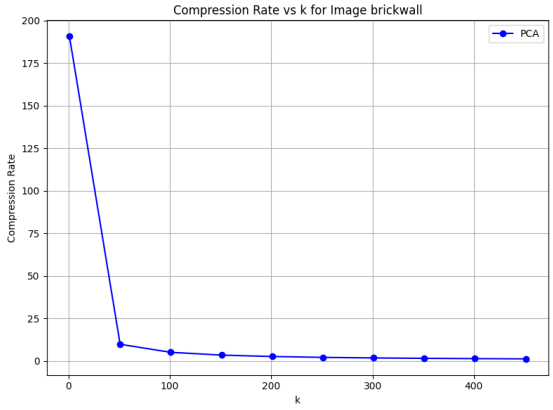
Comparison Between PCA and JPEG

For a given BPP, the custom JPEG consistently outperforms PCA in terms of RMSE. This is especially evident at lower BPP values, where PCA struggles with higher reconstruction errors. The superior performance of JPEG can be attributed to its design, which incorporates human visual system characteristics, prioritizing perceptually important information.

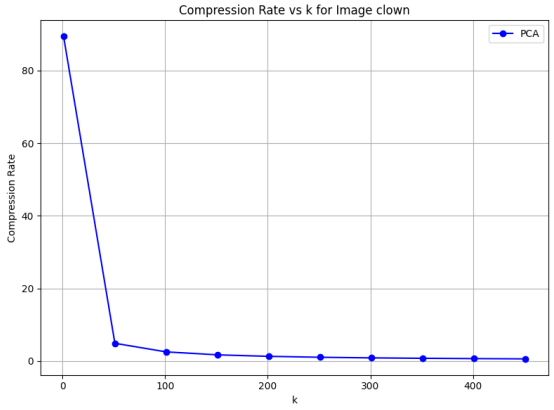
Trade-off Analysis

The trade-off between compression and reconstruction quality is more favorable for JPEG at lower BPP. PCA might be preferable in scenarios where high dimensionality reduction is required, but it is less effective for perceptual image quality preservation compared to JPEG.

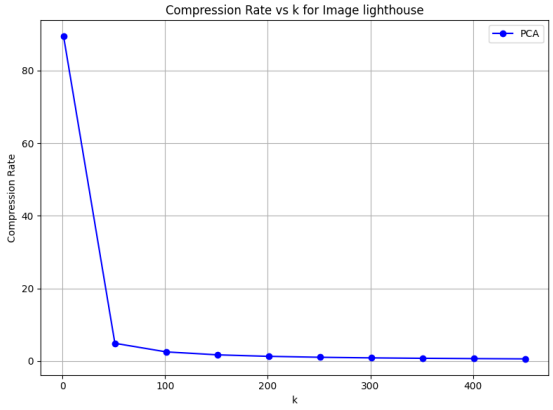
Compression Rate vs. k for a few images



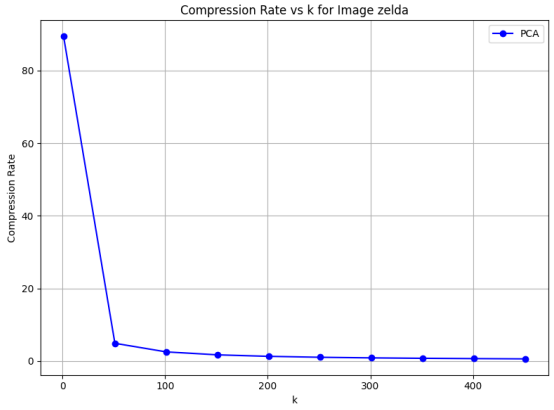
(a) Image 1: Compression Rate vs. k



(b) Image 2: Compression Rate vs. k

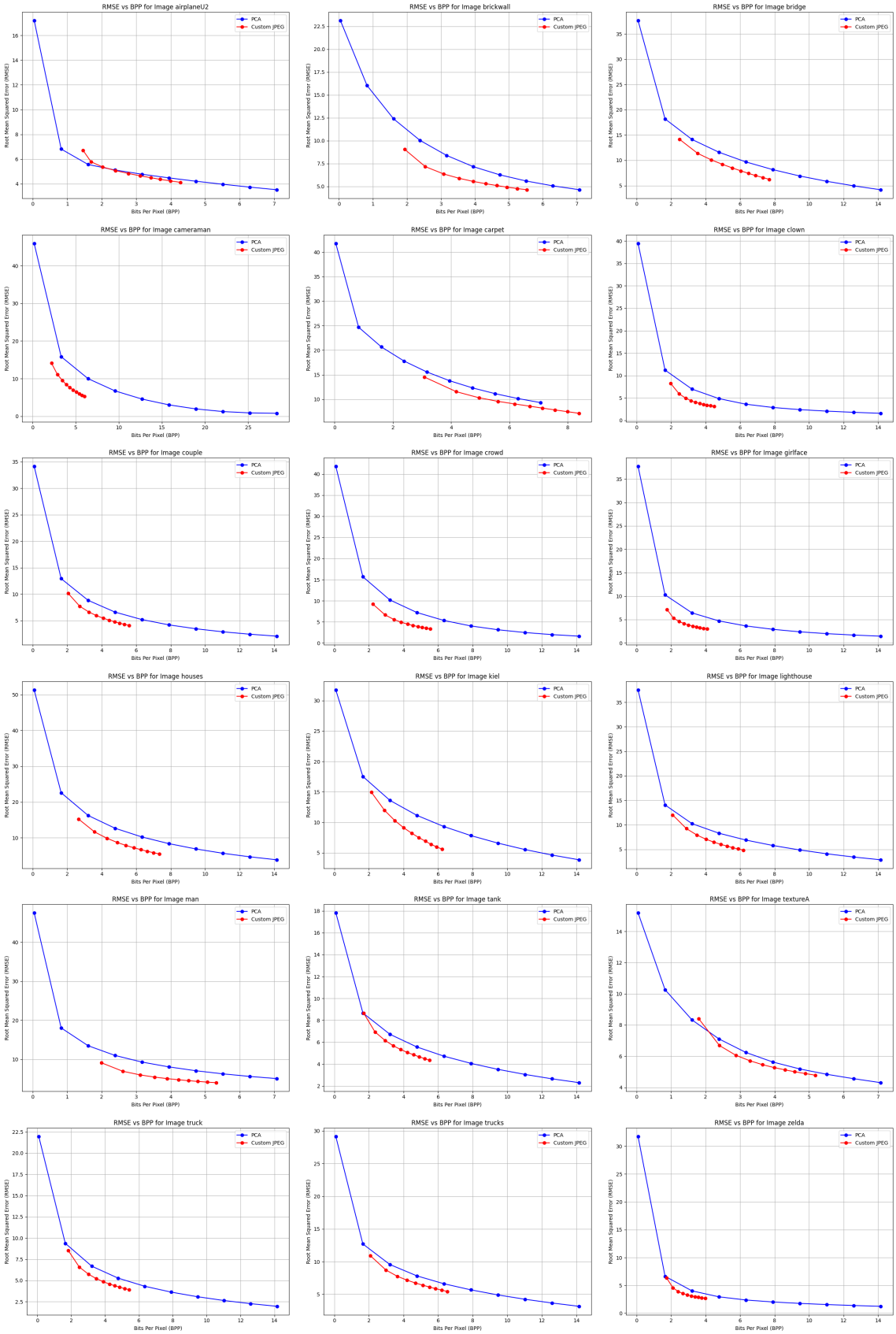


(c) Image 3: Compression Rate vs. k



(d) Image 4: Compression Rate vs. k

Figure 20: Compression Rate vs. k for a few images. Each graph shows how the compression rate evolves as k increases. The compression rate decreases sharply at first and stabilizes as k increases, indicating diminishing returns in compression efficiency with higher values of k.


RMSE vs BPP for Image airplaneU2
RMSE vs BPP for Image brickwall
RMSE vs BPP for Image bridge
RMSE vs BPP for Image cameraman
RMSE vs BPP for Image carpet
RMSE vs BPP for Image clown
RMSE vs BPP for Image couple
RMSE vs BPP for Image crowd
RMSE vs BPP for Image girlface
RMSE vs BPP for Image houses
RMSE vs BPP for Image kiel
RMSE vs BPP for Image lighthouse
RMSE vs BPP for Image man
RMSE vs BPP for Image tank
RMSE vs BPP for Image textureA
RMSE vs BPP for Image truck
RMSE vs BPP for Image trucks
RMSE vs BPP for Image zelda
PCA
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

7 Sparse Orthonormal Transforms and Comparison with Custom JPEG

7.1 Details of Implementation

The `JPEG_SOT` class is essentially the same as the `JPEG` class as defined before, with just a few changes, which will be discussed. The algorithms were trained on 8x8 blocks of two images, astronaut and gravel from the skimage data module.

Algorithm 1

To implement Algorithm 1, which is essentially convergence of a particular transformation matrix given blocks, we had to find the optimal coefficients for the blocks given a transformation matrix, and the optimal transformation matrix given these coefficients, and continue this cycle until the matrix converges. This is done as follows.

```
def get_opt_coeff_given_transform(G_i, blocks, lam):
    '''
    G_i is NxN, blocks should be CxN (flattened from sqrtN x sqrtN)
    return c_ij (should be CxN)
    '''
    coeffs = blocks @ G_i
    coeffs[np.abs(coeffs) < np.sqrt(lam)] = 0

    return coeffs

def get_opt_transform_given_coeff(c_i, x):
    '''
    x should be a numpy array CxN, where C is total no of blocks, c_i should be a numpy
        array CxN, corresponding to x
    '''
    Y = c_i.T @ x
    U, _, V = np.linalg.svd(Y, full_matrices=False)

    G_i = V @ U.T
    return G_i

def algorithm_1(transform, lam, blocks):
    '''
    transform NxN, coeffs CxN, blocks CxN
    '''
    prev_transform = np.zeros(transform.shape)
    coeffs = None
    while(np.mean((transform-prev_transform)**2) > 0.05):
        prev_transform = transform
        coeffs = get_opt_coeff_given_transform(G_i=transform, blocks=blocks, lam=lam)
        transform = get_opt_transform_given_coeff(c_i=coeffs, x=blocks)
    return transform
```

The formulae required and the proof of optimality for the equations above have been covered in the paper.

Algorithm 2

Algorithm 2 uses Annealing of the lambda parameter, and Algorithm 1 as a sub-process, to perform a clustering sort of process for each block, and transformation matrix. Essentially, we find eight optimal transformations using algorithm 1, perform re-labelling of blocks according to the lowest cost, and then lower the value of lambda and repeat the process, until the labellings converge completely. The code has been added below.

```
def algorithm_2(transforms,blocks, labels, lam_max=200, lam_min=20, lam_diff=20):
    lam = lam_max
    old_labels = np.zeros_like(labels)
    cnt = 0
```

```
    while(not np.array_equal(old_labels, labels)):
        cnt += 1
        if cnt > 100:
            break
        old_labels = labels
        while lam > lam_min:
            for i in range(8):
                reqd_blocks = blocks[np.where(labels == i)[0]]
                transforms[i] = algorithm_1(transform=transforms[i], lam=lam, blocks=
                    reqd_blocks)
            for i in range(blocks.shape[0]):
                costs = np.zeros(8)
                for k in range(8):
                    coeff = get_opt_coeff_given_transform(transforms[k], blocks[i], lam)
                    costs[k] = np.sum((blocks[i]-coeff@transforms[k])**2)+lam*(np.
                        count_nonzero(coeff))
                new_label = np.argmin(costs)
                labels[i] = new_label

            lam -= lam_diff
        lam = lam_max
        print(cnt)
        print(np.sum(old_labels != labels))
    return transforms
```

We found that using lower values of lambda during training and higher values of lambda during compression is beneficial. Here, the loop which calls Algorithm 1 finds the optimal transformations, and the loop after that performs cost minimisation and re-labelling (essentially performing clustering, with transformation matrices as the centers).

Changes to JPEG Class

Now that we have our transformation matrices after training, only trivial changes need to be made to our original `JPEG` class. We need to find and store the optimal transformation matrix corresponding to each block, and perform the transformation and inverse transformation using that matrix. This is done by changing the `_dct2` and `_idct2` functions from the original class. The changes have been shown below. Our `JPEG_SOT` class uses a default lambda of 10000.

```
def t(self, block):
    flat_block = block.flatten()
    flat_block = np.reshape(flat_block, (1,64))
    costs = np.zeros(8)
    for i in range(8):
        coeff = get_opt_coeff_given_transform(self.transforms[i], flat_block, self.lam)
        costs[i] = np.sum((flat_block-coeff@self.transforms[i])**2)+self.lam*(np.
            count_nonzero(coeff))
    best_transform = self.transforms[np.argmin(costs)]
    return np.reshape(get_opt_coeff_given_transform(best_transform, flat_block, self.lam),
        (8,8)), best_transform

def it(self, block, transform):
    return np.reshape((transform @ block.flatten()), (8,8))
```

Essentially, we are creating 64x64 transformation matrices, and hence we use the blocks as 64x1 matrices. We find the transformation matrix with the lowest cost, and return the best transformation matrix for that block, along with the optimal coefficients for it. During the inverse step, the optimal matrix simply needs to be multiplied with the coefficients to get back the image. The storage of matrices was done by adding the following lines. Note that the lines below are simply changes to the original class and not complete functions.

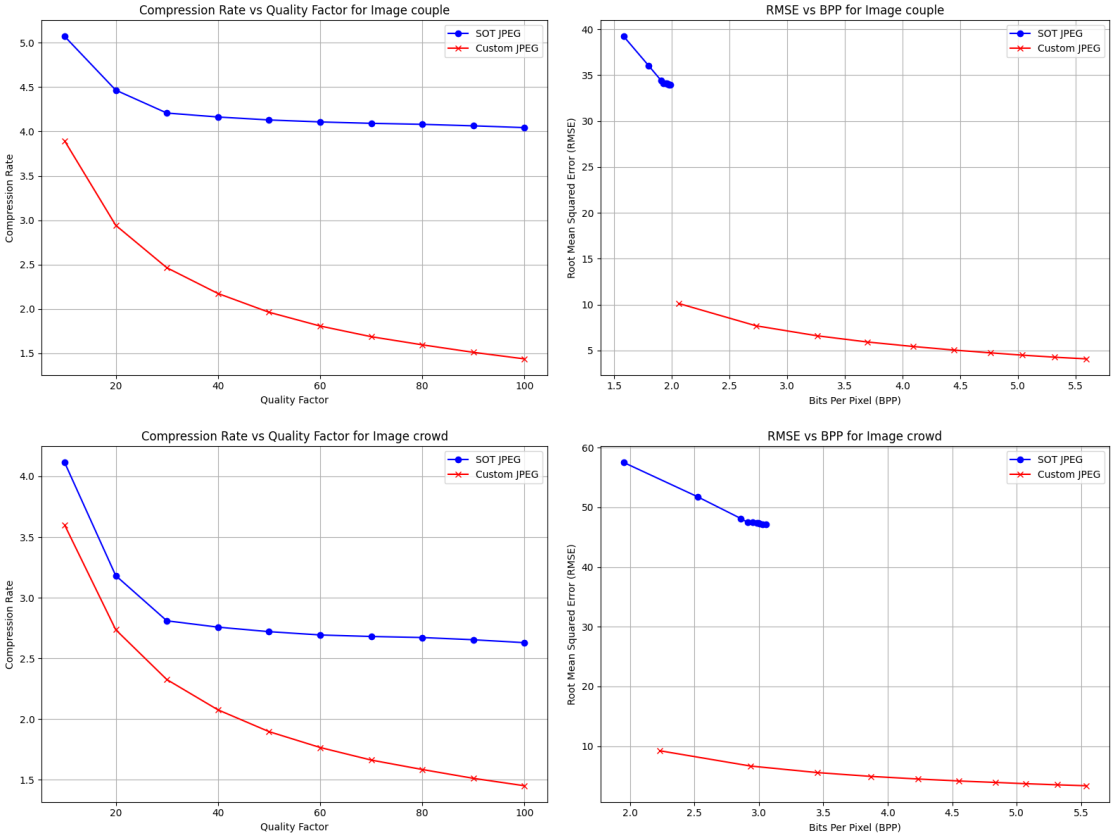
```
def __init__(self, Q = 50, lam=10000):
    self.transforms = transforms
    self.block_to_transform = {}
    self.lam = lam

def compress(self, image, filename):
```

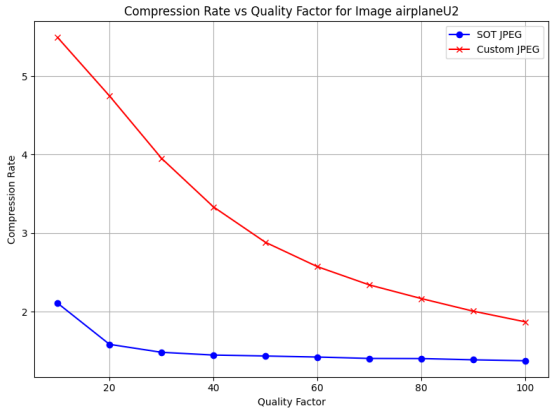
```
    for i, chunk in enumerate(chunks):
        val, transform = self.t(chunk)
        quantized_matrix = self._quantize(val)
        flattened_matrix = self._zigzag(quantized_matrix)
        flattened_matrices.append(flattened_matrix)
        self.block_to_transform[i] = transform


def decompress(self, filename):
    for i in range(len(flattened_matrices)):

        flattened_matrices[i] = self._reverse_zigzag(flattened_matrices[i], 8, 8)

        # now de-quantize the matrix and rescale
        flattened_matrices[i] = (flattened_matrices[i] * self.scaled_quantization_matrix)
        flattened_matrices[i] = self.it(flattened_matrices[i], self.block_to_transform[i]) +
            128
```

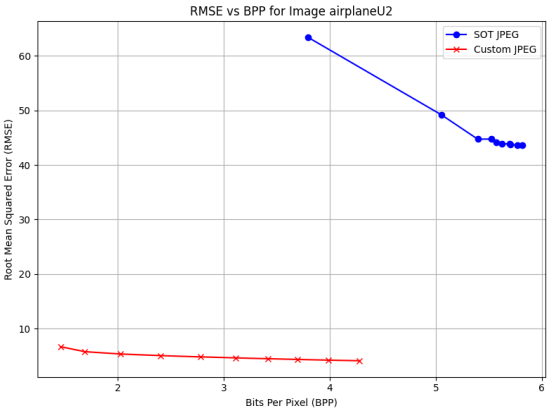
7.2 Comparison with Custom JPEG

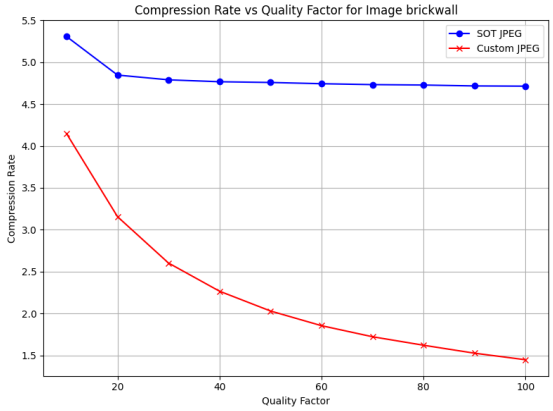
Since we use SOT on top of our custom `JPEG` class, we would want to compare the performance of trained transformations vs direct DCT applications. This comparison would give a direct answer as to whether there are improvements.

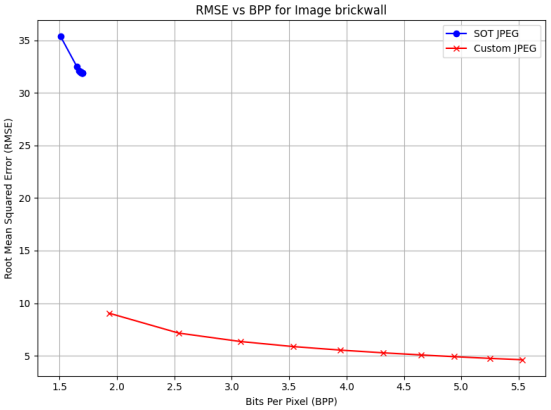
As we can see from the graphs below, on most occasions, SOT outperforms DCT in terms of compression rate, which means that it would give noticeable improvements over standard JPEG, especially with further hyperparameter tuning and more training time. However, the BPP for SOT does not vary much at all, and the RMSE is quite high as well. The results run on the 19 images are shown below.

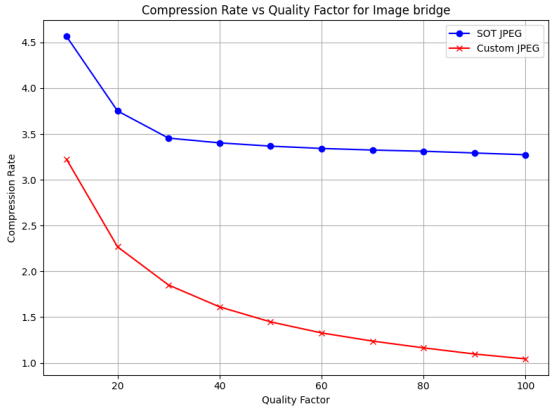


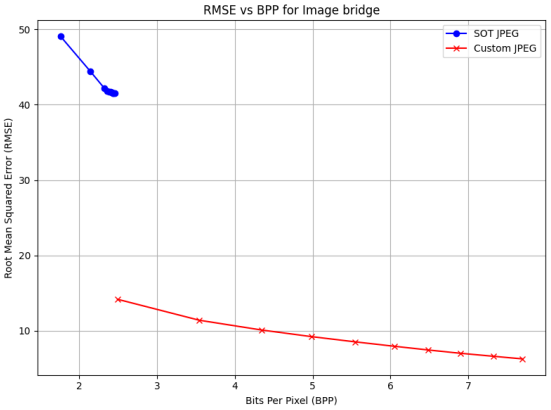

Compression Rate vs Quality Factor for Image airplaneU2
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

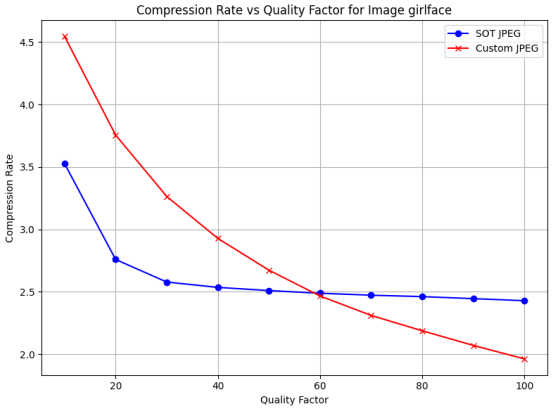

RMSE vs BPP for Image airplaneU2
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

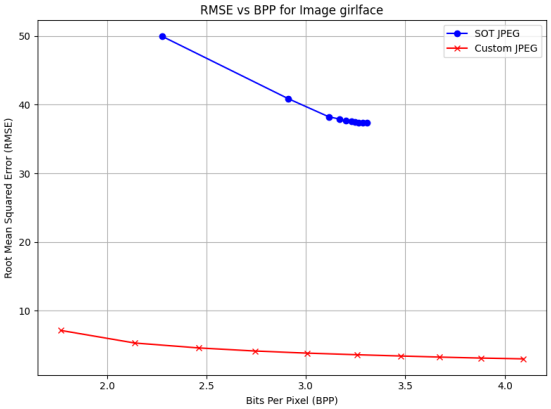

Compression Rate vs Quality Factor for Image brickwall
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

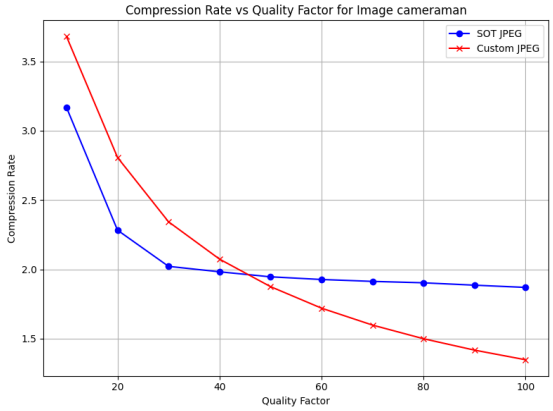

RMSE vs BPP for Image brickwall
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

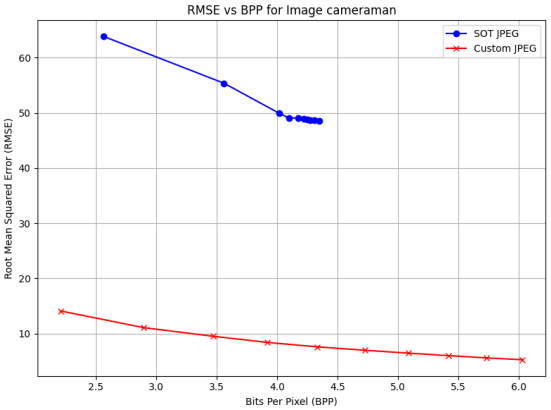

Compression Rate vs Quality Factor for Image bridge
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

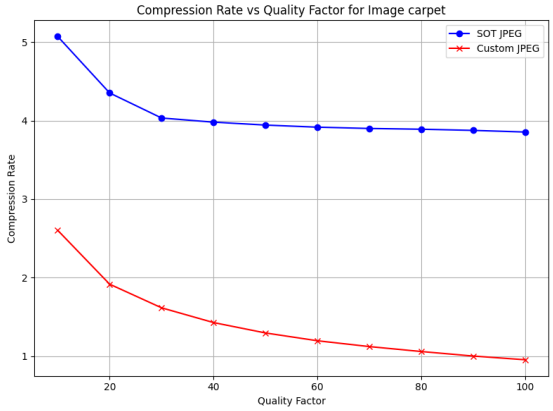

RMSE vs BPP for Image bridge
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

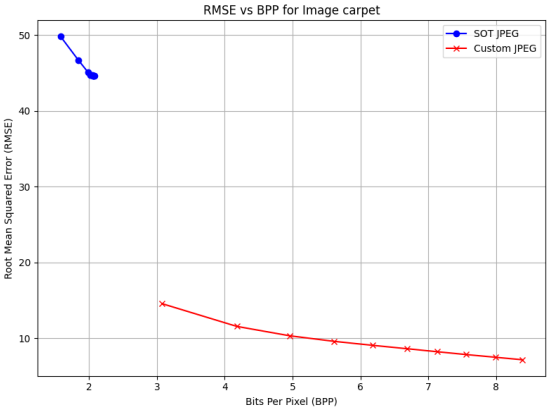

Compression Rate vs Quality Factor for Image girlface
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

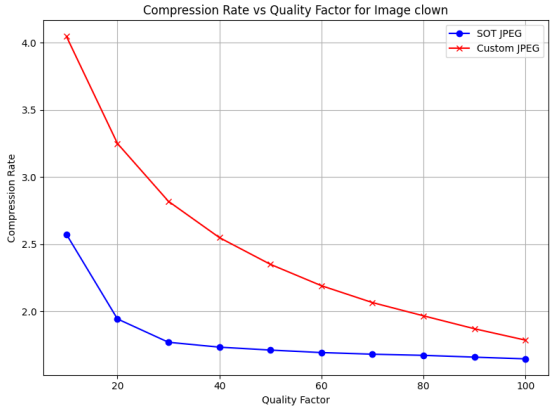

RMSE vs BPP for Image girlface
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

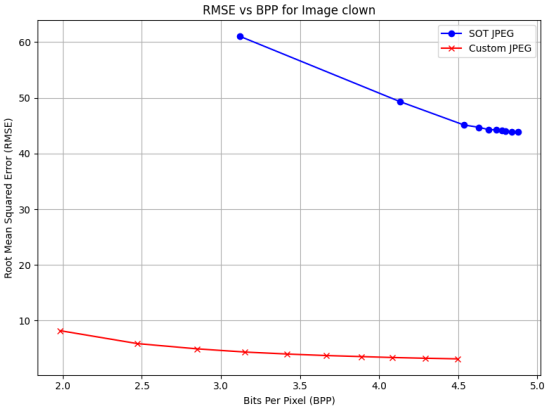

Compression Rate vs Quality Factor for Image cameraman
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

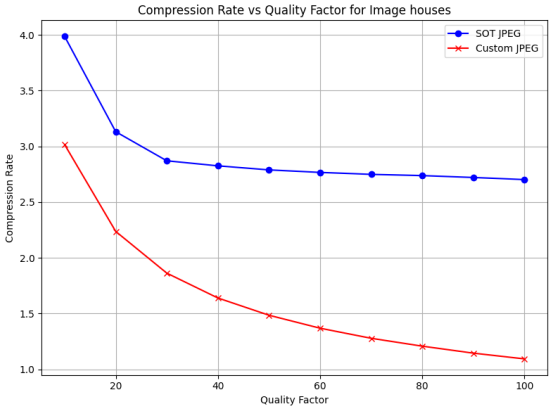

RMSE vs BPP for Image cameraman
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

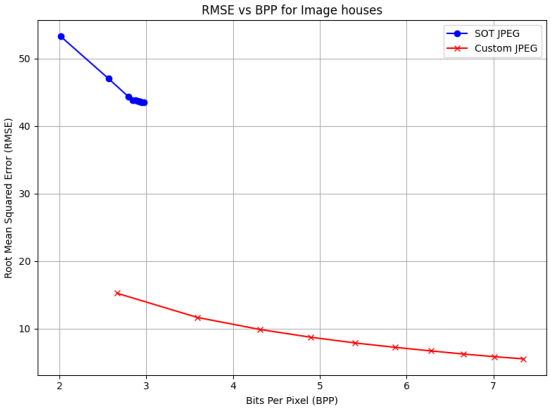

Compression Rate vs Quality Factor for Image carpet
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

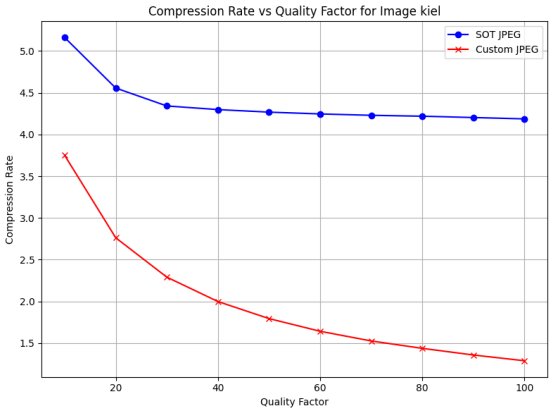

RMSE vs BPP for Image carpet
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

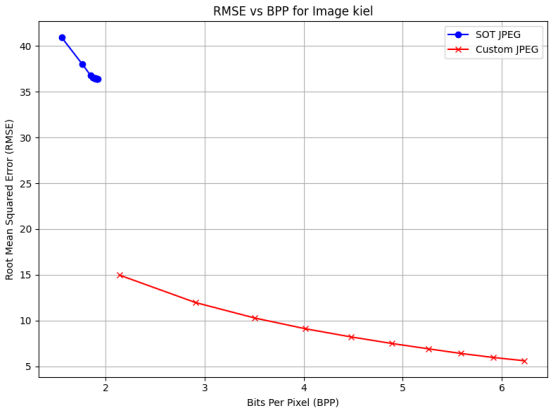

Compression Rate vs Quality Factor for Image clown
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

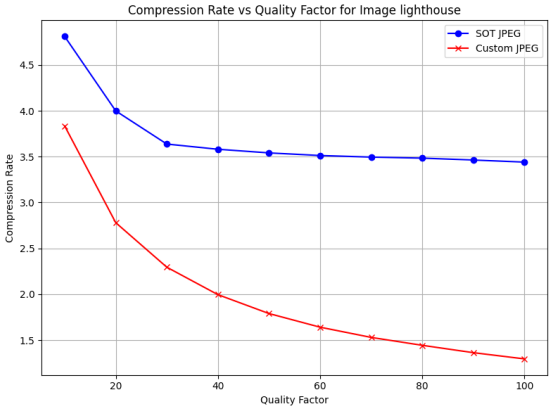

RMSE vs BPP for Image clown
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

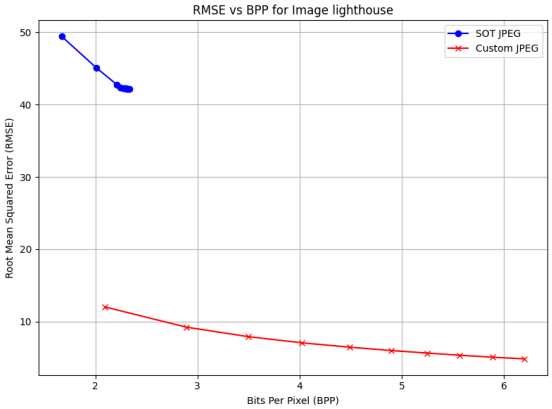

Compression Rate vs Quality Factor for Image houses
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

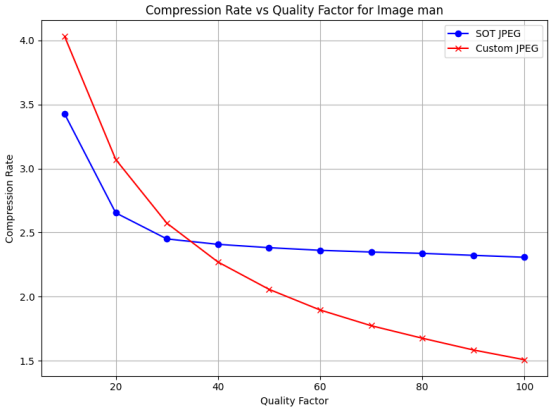

RMSE vs BPP for Image houses
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

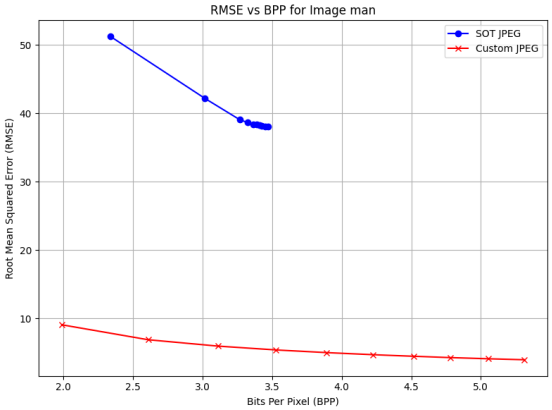

Compression Rate vs Quality Factor for Image kiel
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

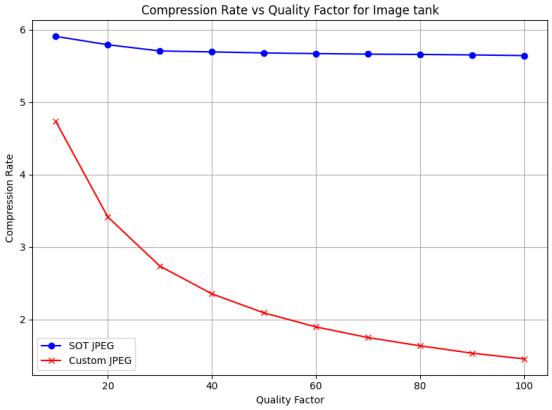

RMSE vs BPP for Image kiel
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

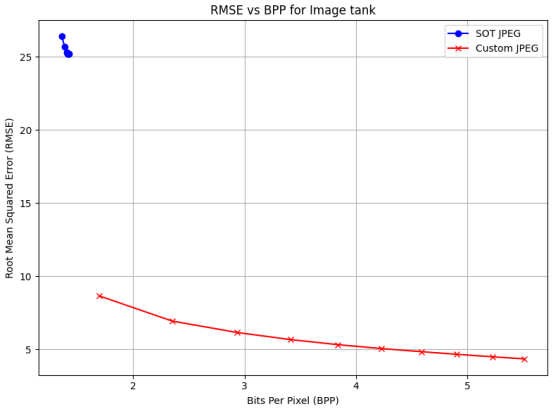

Compression Rate vs Quality Factor for Image lighthouse
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

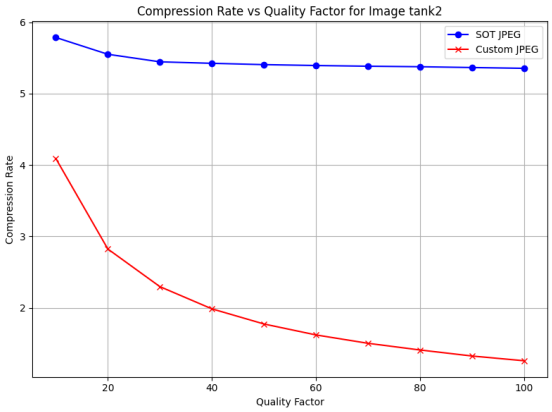

RMSE vs BPP for Image lighthouse
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

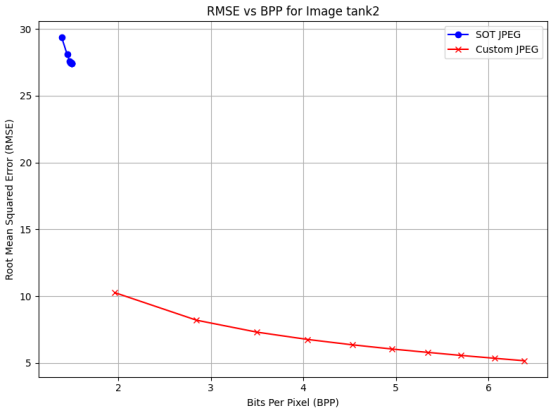

Compression Rate vs Quality Factor for Image man
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

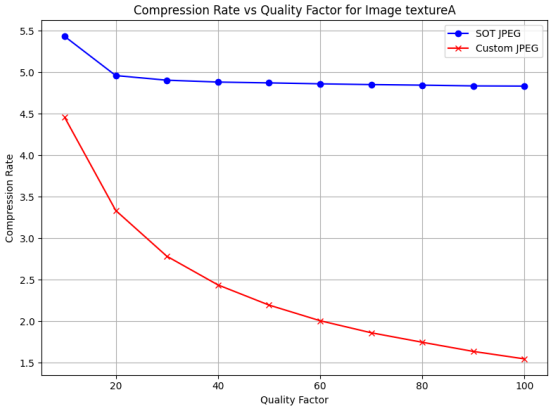

RMSE vs BPP for Image man
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

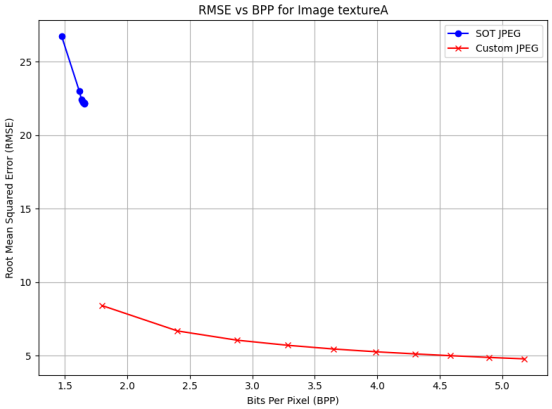

Compression Rate vs Quality Factor for Image tank
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

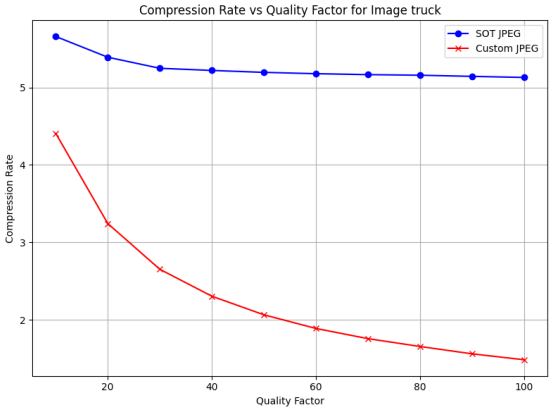

RMSE vs BPP for Image tank
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

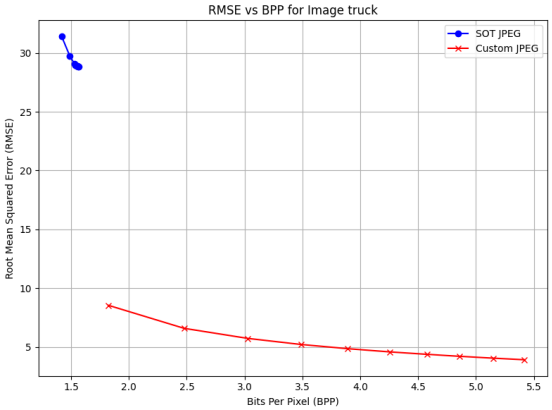

Compression Rate vs Quality Factor for Image tank2
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

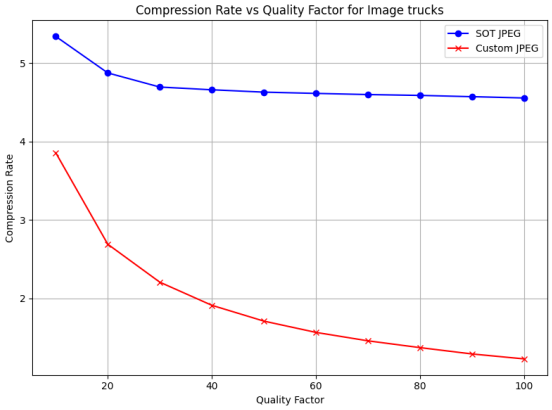

RMSE vs BPP for Image tank2
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)

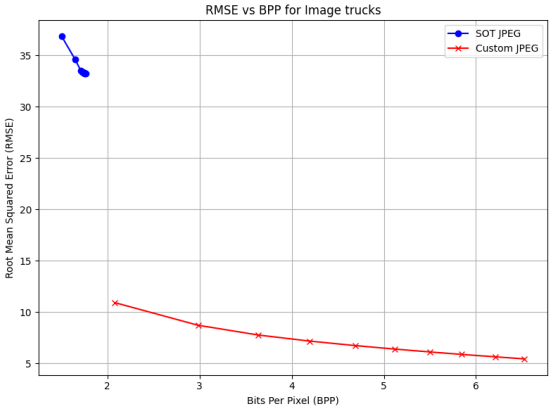

Compression Rate vs Quality Factor for Image textureA
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor

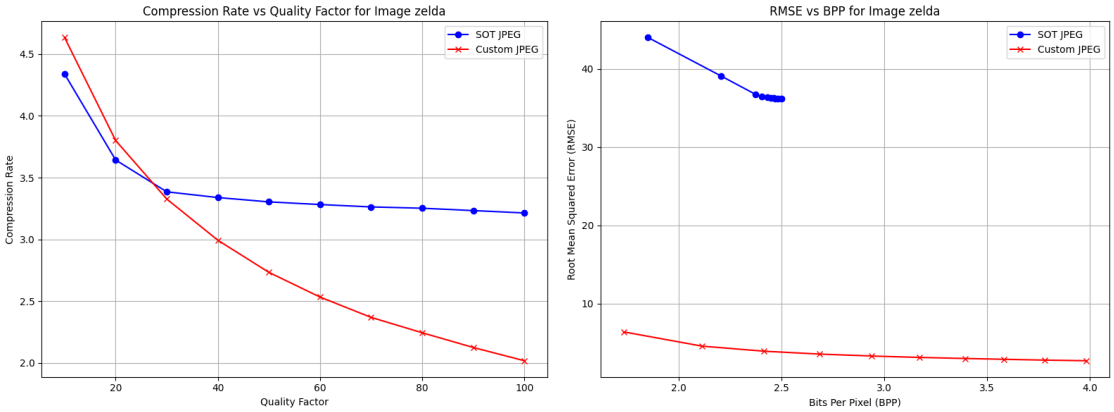

RMSE vs BPP for Image textureA
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)


Compression Rate vs Quality Factor for Image truck
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor


RMSE vs BPP for Image truck
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)


Compression Rate vs Quality Factor for Image trucks
SOT JPEG
Custom JPEG
Compression Rate
Quality Factor


RMSE vs BPP for Image trucks
SOT JPEG
Custom JPEG
Root Mean Squared Error (RMSE)
Bits Per Pixel (BPP)



8 Conclusions and Wrapping Up

Our compression scheme doesn't compare too well to the standard cv2 implementation. This could be for a number of reasons. Most notably, the fact that we have a custom compression and encoding scheme which is most likely filled with inefficiencies. We have also left out the AC/DC coefficients in our implementation which just adds complexity to only marginally improve the compression rate.

On the bright side, we have RMSE values somewhat comparable to the standard cv2 implementation for varying Q. Our implementation is also robust across various different kinds of images, achieving reasonable compression even on "challenging" images where even cv2 fails to get good compression.

Our JPEG implementation also outperforms a naive implementation of PCA Compression in most natural images, which is a good sign.

In addition, we saw positive results with the SOT implementation, which could be added to a well performing JPEG implementation and hence, improve it.

9 Contributions

Everyone contributed equally towards all parts of the project, however the main implementation work was split equally, which has been listed below.

- Brian Mackwan
    1. PCA Implementation
    2. Extensive Analysis and Experiments in all implementations
- Ekansh Ravi Shankar
    1. Sparse Orthogonal Transformation Implementation
    2. Extensive Analysis and Experiments in all implementations
- Ved Danait
    1. Basic JPEG Implementation
    2. Extensive Analysis and Experiments in all implementations