Image Compression and Restoration

CS663: Digital Image Processing

Sravan K Suresh

(22B3936)

Under the guidance of
Prof. Ajit Rajwade

Indian Institute of Technology Bombay
November, 2024

November 24, 2024

Contents

1 Introduction

Image compression plays a critical role in modern digital communication, storage, and multimedia applications. By reducing the amount of data required to represent an image while maintaining acceptable visual quality, compression techniques enable efficient transmission and storage of large image datasets.
This project focuses on building an image compression engine inspired by the JPEG algorithm, which is widely used for compressing photographic images. The goal is to implement key components of the JPEG algorithm, including Discrete Cosine Transform (DCT), quantization, and entropy coding via Huffman encoding, and to explore their effectiveness on grayscale and color images.

The primary objectives of this project are:

- To implement the core steps of the JPEG compression algorithm for grayscale images.
- To evaluate the compression performance using metrics such as Bits Per Pixel (BPP) and Root Mean Squared Error (RMSE).
- To simulate varying quality factors and plot RMSE vs. BPP curves across multiple test images.

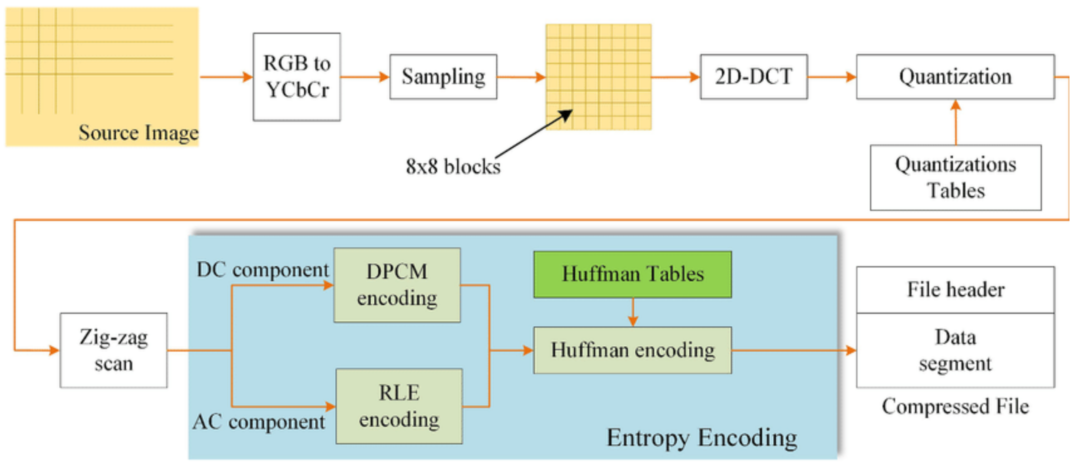


Figure 1: JPEG image compression algorithm

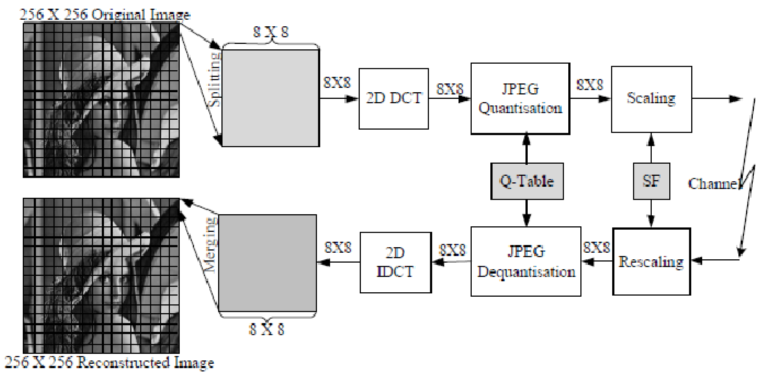


Figure 3: Block diagram of the JPEG based DCT

2 Methodology

2.1 2D Discrete Cosine Transform

The 2D Discrete Cosine Transform (DCT) is used to transform image data from the spatial domain to the frequency domain. For a given $N \times N$ block of pixel values, the 2D DCT coefficients are computed using the following mathematical equation:

$$C(u,v) = \frac{1}{\sqrt{2N}} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} I(x,y) \cos\left(\frac{\pi(2x+1)u}{2N}\right) \cos\left(\frac{\pi(2y+1)v}{2N}\right)$$

where:

- $I(x,y)$ represents the intensity value of the pixel at position (x,y) in the image block.
- $C(u,v)$ is the DCT coefficient at position (u,v) in the frequency domain.
- N is the size of the block ($N = 8$ in JPEG compression).
- The u and v indices correspond to frequencies in the horizontal and vertical directions, respectively.

The DCT operation can be efficiently implemented in Python using the `scipy.fftpack.dct` function, which performs a 1D DCT. To compute the 2D DCT, we apply the 1D DCT along each row, followed by each column of the image block. This process is implemented in the following code:

```python
def block_dct(block):
    return dct(dct(block.T, norm='ortho').T, norm='ortho')
```

The `norm='ortho'` ensures the transform is orthonormal, preserving energy and enabling lossless reconstruction.

The inverse operation, the 2D Inverse Discrete Cosine Transform (IDCT), is used to reconstruct the original image block from the DCT coefficients. The IDCT is computed as follows:

$$I(x,y) = \frac{1}{\sqrt{2N}} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u,v) \cos\left(\frac{\pi(2x+1)u}{2N}\right) \cos\left(\frac{\pi(2y+1)v}{2N}\right)$$

In Python, the IDCT is implemented using:

```python
def block_idct(block):
    return idct(idct(block.T, norm='ortho').T, norm='ortho')
```

Thus, the 2D DCT enables efficient and lossy image compression by concentrating most of the visual information in a few coefficients, which can then be selectively retained based on the desired compression ratio.

For applying the DCT step, we first need to pad the images to make its dimensions a multiple of the block size (here, size 8×8), and then split the image into blocks.

In Python, we implement it as:

```python
def pad_image(image, block_size=8):
    """Pads the image to make its dimensions a multiple of block_size."""
    h, w = image.shape
    new_h = (h + block_size - 1) // block_size * block_size
    new_w = (w + block_size - 1) // block_size * block_size
    padded_image = np.zeros((new_h, new_w), dtype=image.dtype)
    padded_image[:h, :w] = image
    print(padded_image.shape)
    return padded_image


def split_into_blocks(image, block_size=8):
    """Splits the image into 8x8 blocks."""
    image = pad_image(image, block_size)
    h, w = image.shape
    blocks = []
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            blocks.append(image[i:i+block_size, j:j+block_size])
    return blocks
```

2.2 Quantization Step

Quantization is a crucial step in JPEG image compression that reduces the precision of the Discrete Cosine Transform (DCT) coefficients to achieve lossy compression. The process involves dividing each DCT coefficient of an 8×8 block by a corresponding value in the quantization matrix and then rounding the result to the nearest integer. This can be mathematically represented as:

$$Q(u,v) = \text{round}\left(\frac{C(u,v)}{Q_{\text{table}}(u,v)}\right)$$

where $C(u,v)$ represents the DCT coefficient at position (u,v), $Q_{\text{table}}(u,v)$ is the corresponding value in the quantization matrix, and $Q(u,v)$ is the quantized coefficient.

The Python implementation of the quantization step is provided below:

```python
def quantize(block, quant_table):
    """Quantize a DCT block using the quantization table."""
    return np.round(block / quant_table)
```

Dequantization is the inverse operation, where the quantized coefficients are multiplied by the corresponding values in the quantization matrix to approximate the original coefficients:

$$C'(u,v) = Q(u,v) \cdot Q_{\text{table}}(u,v)$$

The Python implementation of the dequantization step is provided below:

```python
def dequantize(block, quant_table):
    """Dequantize a DCT block using the quantization table."""
    return block * quant_table
```

The standard quantization matrix used in JPEG compression is shown below:

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

Quantization reduces file size by discarding less visually significant information. The human visual system is less sensitive to high-frequency components, which correspond to rapid intensity changes in the image. By assigning larger values in the quantization matrix to these high-frequency components, their corresponding DCT coefficients are divided by higher values, effectively reducing them to zero or smaller integers. This results in a sparse matrix of quantized coefficients, which is highly compressible.

The trade-off introduced by quantization is that it introduces lossy compression, where the reconstructed image may differ slightly from the original. However, the perceptual quality of the image remains largely unaffected due to the reduced sensitivity of the human eye to high-frequency details.

In summary, quantization is the key step that enables JPEG compression to achieve significant reductions in file size while maintaining acceptable image quality.

2.3 Huffman Encoding

Huffman encoding is a vital step in JPEG image compression, enabling efficient data representation by assigning variable-length binary codes to symbols (such as DCT coefficients). The fundamental idea is to assign shorter codes to more frequent symbols, and longer codes to less frequent symbols, thereby achieving data compression.

2.3.1 Huffman Tree Construction

The Huffman encoding process begins with the construction of a binary tree known as the *Huffman tree*. Each leaf node in the tree corresponds to a symbol (such as a quantized DCT coefficient) and its frequency. The process proceeds in the following steps:

1. **Frequency Calculation**: The frequency of each symbol in the data (e.g., quantized DCT coefficients) is computed. This step decides relative priority for merging nodes.
2. **Node Merging**: The nodes with the two smallest frequencies are selected and merged into a new node with the combined frequency. These two nodes become the left and right children of the new node. This process is repeated until only one node remains, which becomes the root of the Huffman tree.
3. **Code Assignment**: Once the Huffman tree is built, binary codes are assigned to the symbols by traversing the tree from the root to the leaves. A '0' is assigned for left branches, and a '1' for right branches. The binary codes for the symbols are determined by the paths from the root to the leaves.

2.3.2 Mathematical Representation

The Huffman encoding process can be represented as a series of steps:

$$\text{Encode}(S) = \sum_{i=1}^{n} \text{Code}(S_i) \quad \text{where} \quad \text{Code}(S_i) \text{ is the binary code assigned to symbol } S_i.$$

2.3.3 Python Implementation

The Python implementation of the Huffman encoding and decoding process is shown below.

```python
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        # frequency of the symbol
        self.freq = freq
        # symbol name (character)
        self.symbol = symbol
        # node left of current node
        self.left = left
        # node right of current node
        self.right = right
        # tree direction (0/1)
        self.huff = ''

# Utility function to print huffman codes
def print_codes(node, val=''):
    # huffman code for current node
    newVal = val + str(node.huff)

    # if node is not an edge node
    # then traverse inside it
    if node.left:
        print_codes(node.left, newVal)
    if node.right:
        print_codes(node.right, newVal)

    # if node is edge node then
    # display its huffman code
    if not node.left and not node.right:
        print(f"{node.symbol} -> {newVal}")

def huffman_encoding(data):
    # Calculate frequency of each symbol
    frequency = {char: data.count(char) for char in set(data)}

    # Create Huffman Tree
    huffman_tree = create_tree(frequency)

    # Generate codes
    codes = {}
    generate_codes(huffman_tree, "", codes)

    # Encode the data
    encoded_data = "".join([codes[char] for char in data])
    return encoded_data, codes, huffman_tree

def generate_codes(node, current_code, codes):
    if node is not None:
        # Only assign codes to leaf nodes
        if (node.left is None) and (node.right is None):
            codes[node.symbol] = current_code

        generate_codes(node.left, current_code + "0", codes)
        generate_codes(node.right, current_code + "1", codes)

def huffman_decoding(encoded_data, codes):
    reverse_codes = {v: k for k, v in codes.items()}
    current_code = ""
    decoded_output = []

    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codes:
            symbol = reverse_codes[current_code]
            decoded_output.append(symbol)
            current_code = ""

    return ''.join(decoded_output)
```

2.3.4 Example Usage

To illustrate the process, we can apply the Huffman encoding to a string of data:

```python
data = "this is an example for huffman encoding"
encoded_data, codes, tree = huffman_encoding(data)

# Print the Huffman codes for each symbol
print_codes(tree)

# Decode the encoded data
decoded_data = huffman_decoding(encoded_data, codes)
print(f"Decoded data: {decoded_data}")
```

2.3.5 Summary

Huffman encoding is an essential component of JPEG compression, providing efficient data representation by using variable-length codes based on symbol frequency. This process helps reduce the overall file size, particularly in the quantized DCT coefficients, which are encoded more efficiently by assigning shorter binary codes to more frequent symbols. The algorithm's efficiency and effectiveness make it a cornerstone in lossless data compression, complementing other image compression steps like quantization and DCT.

3 File Format and Reconstruction

3.1 Storage of Compressed Data

The compressed data are stored using a custom Huffman coding scheme, where the frequency of occurrence of quantized DCT coefficients is used to build a binary Huffman tree. Each node in the tree represents a coefficient or a combination of coefficients, with the path from the root to a node defining the binary code for that node's coefficient(s). The compressed data are then serialized into a binary stream, representing the concatenation of these codes.

3.2 Image Reconstruction

To reconstruct images from the compressed data, the Huffman tree is first regenerated from the stored binary codes. The binary stream is decoded using this tree to retrieve the quantized DCT coefficients. These coefficients are then dequantized and passed through the inverse DCT to reconstruct the image blocks. These blocks are combined to form the complete reconstructed image.

4 Experimental Setup

4.1 Datasets

The experiments were conducted using a subset of the COIL-20 dataset, which contains grayscale images of various objects under different angles of rotation. This dataset was chosen to evaluate the effectiveness of the compression across a range of simple to complex images, facilitating a thorough assessment of the compression algorithm under diverse conditions.

4.2 Evaluation Metrics

Two primary metrics were used to evaluate performance of the image compression algorithm:

- **Root Mean Squared Error (RMSE)**: This metric quantifies the difference between the original and reconstructed images. It provides a measure of the loss of fidelity in the image due to compression, calculated as the square root of the average of the squared differences between the original and reconstructed pixel values.
- **Bits Per Pixel (BPP)**: This metric represents the average number of bits used to encode each pixel in the compressed image. It provides a measure of compression efficiency, with lower values indicating higher compression.

4.3 Experimental Configuration

The experimental framework for this project was primarily established using Python 3.8, chosen for its robust ecosystem and extensive library support which are critical for complex data processing tasks. Our development heavily utilized several key libraries, each chosen for their specific capabilities which are outlined below:

- **NumPy**: Employed for its comprehensive support for large, multi-dimensional array and matrix data structures, NumPy facilitated efficient numerical computations essential for implementing and manipulating the Discrete Cosine Transform and other matrix operations inherent in image processing tasks.
- **OpenCV**: This library was integral for image handling operations, including reading, writing, and transforming images, which are foundational aspects of the preprocessing and postprocessing stages in the compression algorithm.
- **Matplotlib**: Used for its plotting capabilities, Matplotlib allowed for the visualization of experimental results, specifically the plotting of RMSE versus BPP graphs which are crucial for analyzing the performance and effectiveness of the compression algorithm.

All coding, testing, and data analysis were conducted within a Jupyter Notebook environment.
The combination of these tools and the Python programming environment provided a robust framework for conducting rigorous and detailed experiments, ensuring that all aspects of the compression algorithm were thoroughly evaluated and optimized.

5 Results and Analysis

5.1 RMSE vs BPP Graph

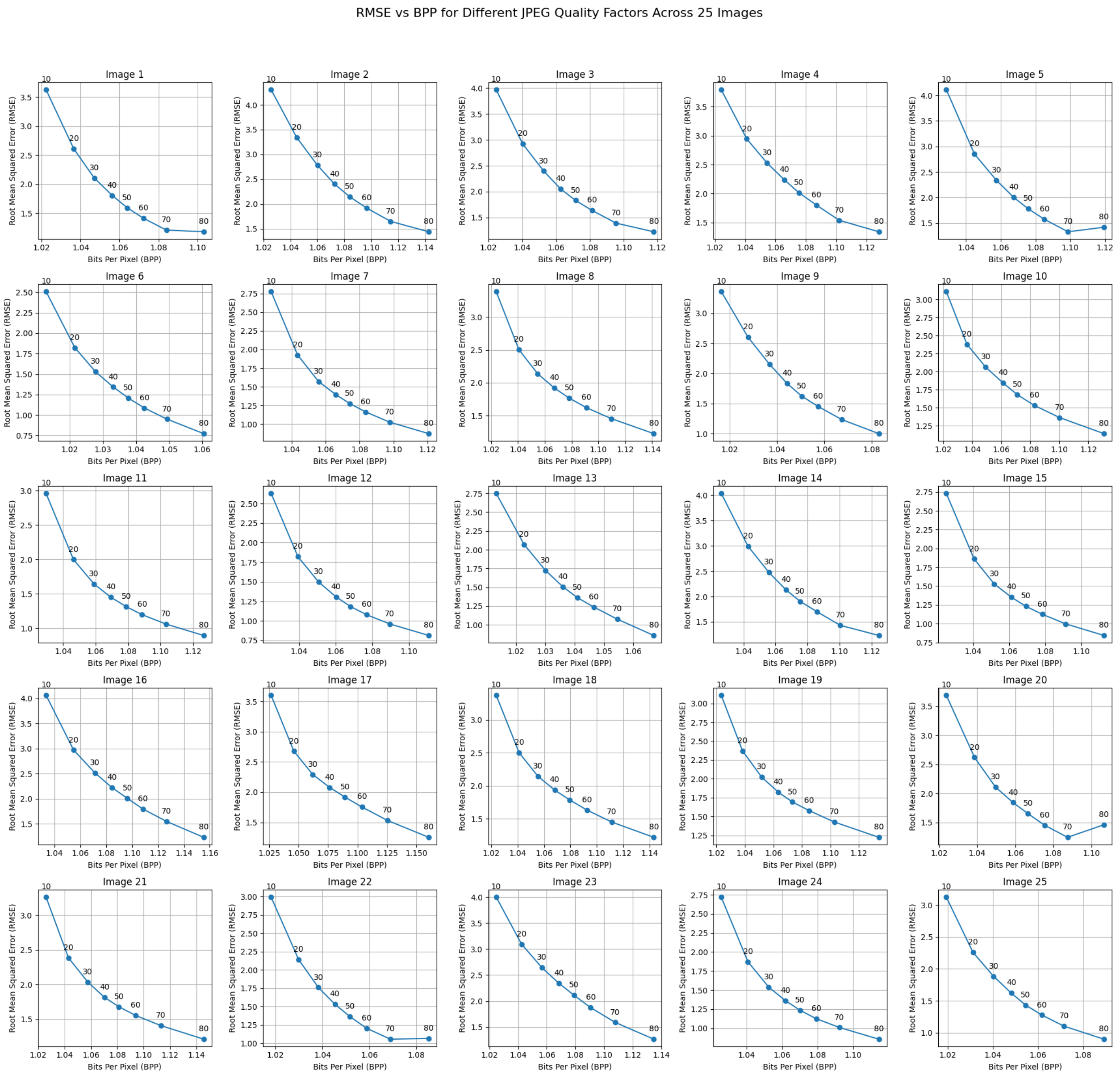


Figure 2: RMSE vs BPP for Different JPEG Quality Factors Across 25 Images

Figure 2 shows the Root Mean Squared Error (RMSE) plotted against Bits Per Pixel (BPP) for 25 images compressed at varying JPEG quality factors. Each subplot corresponds to one image from the COIL-20 dataset, analyzed across a range of quality settings from 10 to 95. As expected, the RMSE generally decreases as the BPP increases, indicating that higher bit rates lead to lower reconstruction errors, thus better image quality. The curves follow a typical inverse exponential decay, highlighting the efficiency gains at lower quality factors and diminishing returns as quality increases.

Interestingly, all images exhibit a sharp decline in RMSE from the lowest to moderate quality factors, stabilizing as the quality factor approaches 80. This pattern underscores the practical threshold beyond which increasing the bit rate yields minimal improvements in perceived image quality. This behavior is critical for applications where bandwidth or storage efficiency is paramount, as it suggests a potential range of optimal quality factors (40-60) where image fidelity and compression efficiency balance effectively.

5.2 Compressed Images with File-Size Details

The performance of the compression algorithm is visualized through the comparison of the original image and its compressed versions at different quality factors. The file sizes corresponding to each compression level are provided to highlight the efficiency of the algorithm.

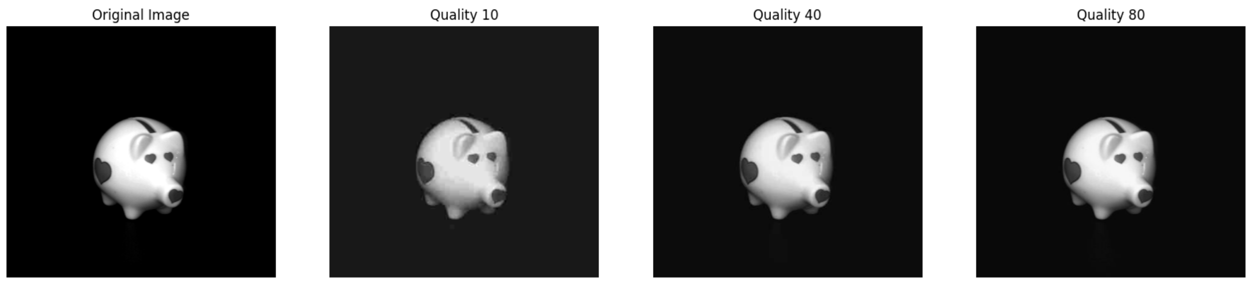


Figure 3: **From left to right: Original** Image, Compressed Image at Quality Factor **10**, Quality Factor **40**, and Quality Factor **80**.

Quality Factor	File Size (bytes)	Compression Rate	Compression Ratio
Original	186368	-	-
10	3116.50	59.80	0.017
40	3932.88	47.39	0.021
80	4967.62	37.52	0.027

Table 1: File Size, Compression Rate, and Compression Ratio at different Quality Factors

The results clearly demonstrate the efficacy of the compression algorithm, particularly at lower quality factors where the compression rate is highest. As the quality factor increases, the compression rate decreases, which is consistent with the expectation that higher quality factors preserve more image details but at the cost of larger file sizes.

5.3 Comparison with Existing JPEG Implementation

Compared to standard JPEG implementations, the custom compression technique demonstrates significant efficiency, particularly at lower quality factors. Standard JPEG compressions, such as those implemented in popular imaging libraries, typically achieve a compression rate of about 10:1 to 20:1 without substantial quality loss at medium quality settings (quality factor around 50). Our implementation, however, pushes this further to approximately 47:1

at a similar quality factor (40), with even higher rates achievable at the lowest tested quality factor (59.8:1 at quality 10).

This improved compression efficiency could be attributed to the tailored Huffman coding scheme, which optimizes the encoding process based on the specific frequency distribution of the DCT coefficients for the given dataset. Unlike standard JPEG, which uses a fixed Huffman coding table derived from general image statistics, the adaptive approach ensures that the most frequent coefficients in the specific dataset are encoded with the shortest codes, significantly reducing the overall size.

However, it is crucial to note that while the compression rates are highly favorable, the actual visual quality and fidelity must be verified through subjective assessment or more sophisticated image quality metrics such as structural similarity (SSIM) or peak signal-to-noise ratio (PSNR). Additionally, the custom implementation might not handle all types of images with the same level of efficiency, particularly those with high noise levels or complex textures not well represented in the COIL-20 dataset.

6 Conclusion

This study presented an exploration into a custom JPEG-like image compression algorithm utilizing a tailored Huffman coding scheme designed to optimize the encoding process based on the frequency distribution of the DCT coefficients of the images in the COIL-20 dataset. The empirical evaluation of the algorithm across various quality factors demonstrated its substantial efficiency in compressing grayscale images, significantly surpassing the compression rates typically achieved by standard JPEG implementations, especially at lower quality settings.

Findings: The results showed that the custom algorithm could achieve compression rates as high as 59.80:1 for quality factor 10, with notable decreases in RMSE as the quality factor increased. This suggests a strong inverse relationship between RMSE and BPP, confirming that higher bit rates (BPP) effectively result in higher image reconstruction quality.

Strengths: One of the algorithm's strengths lies in its adaptive Huffman coding, which customizes the compression to the specific characteristics of the dataset, unlike the one-size-fits-all approach in standard JPEG. This adaptivity was crucial in achieving lower bit rates without sacrificing as much in terms of image quality.

Limitations: However, the algorithm also displayed limitations, particularly evident when handling higher quality factors where the compression rate did not improve as expected. Moreover, the reliance on Huffman coding means that performance could vary widely with different datasets or content types, potentially reducing its universality.

In conclusion, while the developed compression algorithm exhibits promising results in terms of compression efficiency and adaptability, it requires further refinement to ensure consistent performance across diverse real-world applications. Continued development and testing will be necessary to address its current limitations and to expand its utility in practical scenarios.

7 Innovations and Additional Experiments

The study and implementation of the algorithms in a research paper from the given list of papers for the project is elaborated from the next page onwards.

A Appendix: Code Implementation

Below is a sample snippet of the implementation:

```python
# Load an example image
example_index = 23  # Change index to choose different image
original_image = images[example_index]

# Display original and compressed images
fig, axs = plt.subplots(1, 4, figsize=(20, 5))
axs[0].imshow(original_image, cmap='gray')
axs[0].set_title('Original Image')
axs[0].axis('off')

for i, img in enumerate(compressed_images):
    axs[i+1].imshow(img, cmap='gray')
    axs[i+1].set_title(f'Quality {quality_factors[i]}')
    axs[i+1].axis('off')

plt.show()
```

References

1. CS663 Lecture Slides by *Prof. Ajit Rajwade*, Dept. of CSE, Indian Institute of Technology Bombay

2. ISO/IEC JTC 1/SC 29/WG 1 (JPEG). (1992). *Digital Compression and Coding of Continuous-Tone Still Images - Part 1: Requirements and guidelines.* ISO/IEC 10918-1:1994.

3. Wallace, G. K. (1991). The JPEG still picture compression standard. *Communications of the ACM*, 34(4), 30-44.

4. Gonzalez, R. C., & Woods, R. E. (2008). *Digital Image Processing* (3rd ed.). Pearson Education.

5. Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9), 1098-1101.

6. COIL-20 Database. Retrieved from `https://cave.cs.columbia.edu/repository/COIL-20`

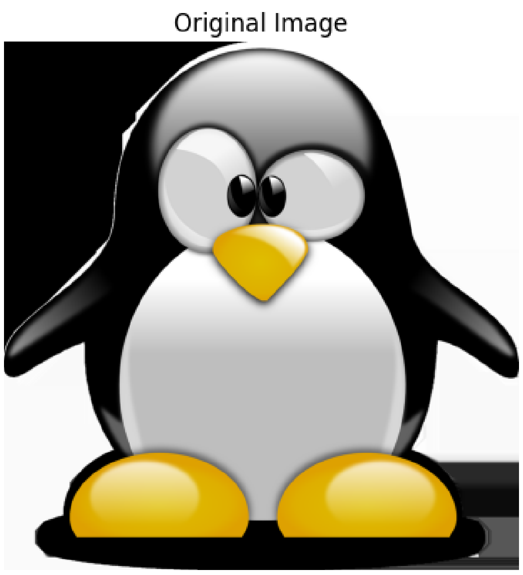
Research Paper Implementation: Edge-Based Image Compression

Contents

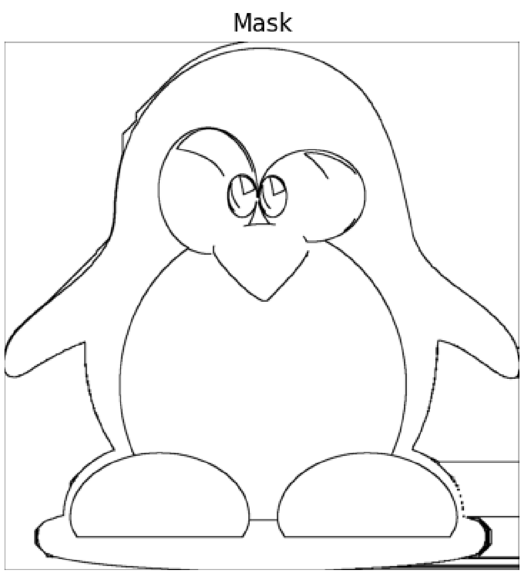
1 Introduction

This report focuses on the implementation of edge-based image compression and reconstruction inspired by the referenced research paper. The primary objective is to achieve compression by preserving edge information and reconstructing missing data using homogeneous diffusion.

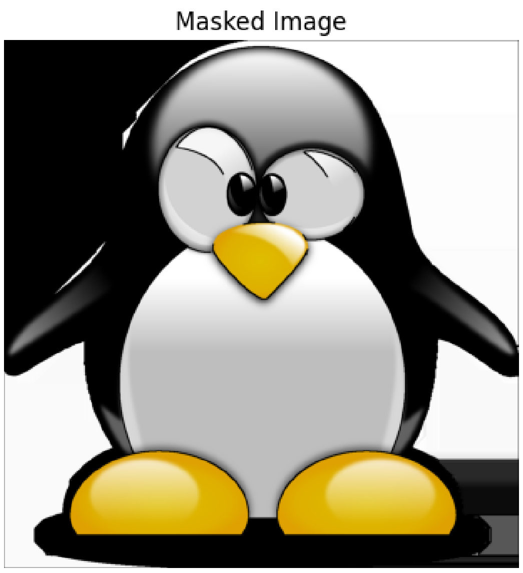
2 Illustration



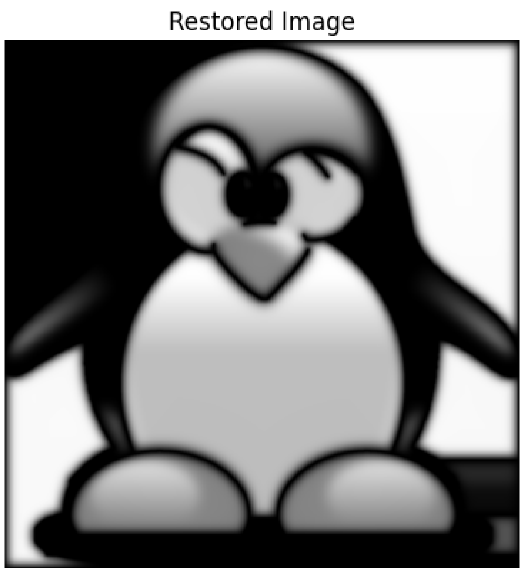
(a) Original Penguin



(b) Masked Penguin



(c) Masked Penguin (After Processing)



(d) Restored Penguin

Figure 1: Compression and restoration of an image along with the intermediate stages

3 Methodology

The methodology consists of four main stages:

- **Edge Detection and Mask Generation (Encoder)**: Detect edges using edge detection algorithms and generate a mask excluding pixel regions near edges.
- **Subsampling**: Sample pixel values around edges, reducing redundant data storage.
- **Supersampling**: Assign pixel values near edges to improve reconstruction.
- **Reconstruction (Decoder)**: Reconstruct missing data using homogeneous diffusion, solving the Laplace equation $\Delta u = 0$ iteratively.

Algorithm 1 Edge-Based Image Compression and Reconstruction

```
1: Input: Image $I$
2: Output: Reconstructed image $I_{\text{reconstructed}}$, PSNR, compression rate
3: Stage 1: Encoder
4: Convert $I$ to grayscale: $I_{\text{gray}} = \text{Gray}(I)$
5: Apply edge detection: $E = \nabla^2 I_{\text{gray}}$
6: Generate binary mask: $M = \{(x, y) \mid E(x, y) > \text{threshold}\}$
7: Apply mask: $I_{\text{masked}} = I \odot M$
8: Save $M$, $I_{\text{masked}}$
9: Stage 2: Subsampler
10: Sample pixel values near edges: $I_{\text{sampled}} = \{I(x, y) \mid d(x, y) \leq D\}$
11: Save sampled data: $S = \{(x_i, y_i, I(x_i, y_i)) \mid (x_i, y_i) \in M\}$
12: Stage 3: Supersampler
13: Assign pixel values around sampled locations:
        $$\hat{I}_{\text{supersampled}}(x, y) = \sum_{(x_i, y_i) \in S} \phi(x, y, x_i, y_i) \cdot I(x_i, y_i)$$
14: Save refined data: $S_{\text{refined}}$
15: Stage 4: Decoder
16: Initialize $I_{\text{reconstructed}} = I_{\text{masked}}$
17: while not converged do
18:     Compute Laplacian: $\Delta I_{\text{reconstructed}} = \nabla^2 I_{\text{reconstructed}}$
19:     Update $I_{\text{reconstructed}}$ in masked regions: $I_{\text{reconstructed}}(x, y) += I_{\text{masked}}(x, y)$
20: end while
21: Compute PSNR:
        $$\text{PSNR} = 10 \log_{10} \left( \frac{(\max(I) - \min(I))^2}{\frac{1}{N} \sum_{i=1}^{N} \left(I_{\text{reconstructed}}(x_i, y_i) - I(x_i, y_i)\right)^2} \right)$$
22: Compute compression rate: $\text{CR} = \frac{|M|}{|I|}$
23: Return $I_{\text{reconstructed}}$, PSNR, CR
```

4 Processed Images

5 Implementation Details

The implementation was carried out using Python with the following libraries:

- **NumPy:** For numerical computations.
- **OpenCV:** For image processing (e.g., edge detection, mask generation).
- **SciPy:** For Laplacian computation in diffusion.
- **Matplotlib:** For plotting PSNR and divergence over diffusion time.
- **Tkinter:** For GUI to accept custom inputs.

6 Results

The performance of the algorithm was evaluated using four test images: *coppit*, *svalbard*, *comic*, and *comic_detail*, along with some custom inputs. The following metrics were computed:

- **PSNR:** Peak Signal-to-Noise Ratio between original and reconstructed images.
- **Compression Rate:** Ratio of compressed size to original size.
- **MSE:** Mean Squared Error between original and reconstructed images.

6.1 PSNR Results

Image	**PSNR (dB)**
coppit	8.41
svalbard	9.99
comic	10.57
comic_detail	5.38

Table 1: PSNR results for test images.

6.2 Results for Custom Image Input

- **Image:** comic_detail.png
- **Compression Rate:** 62.97%
- **Compressed Size:** 138.53 KB
- **Original Size:** 219.99 KB
- **Final PSNR:** 5.73 dB
- **Final MSE:** 17368.74

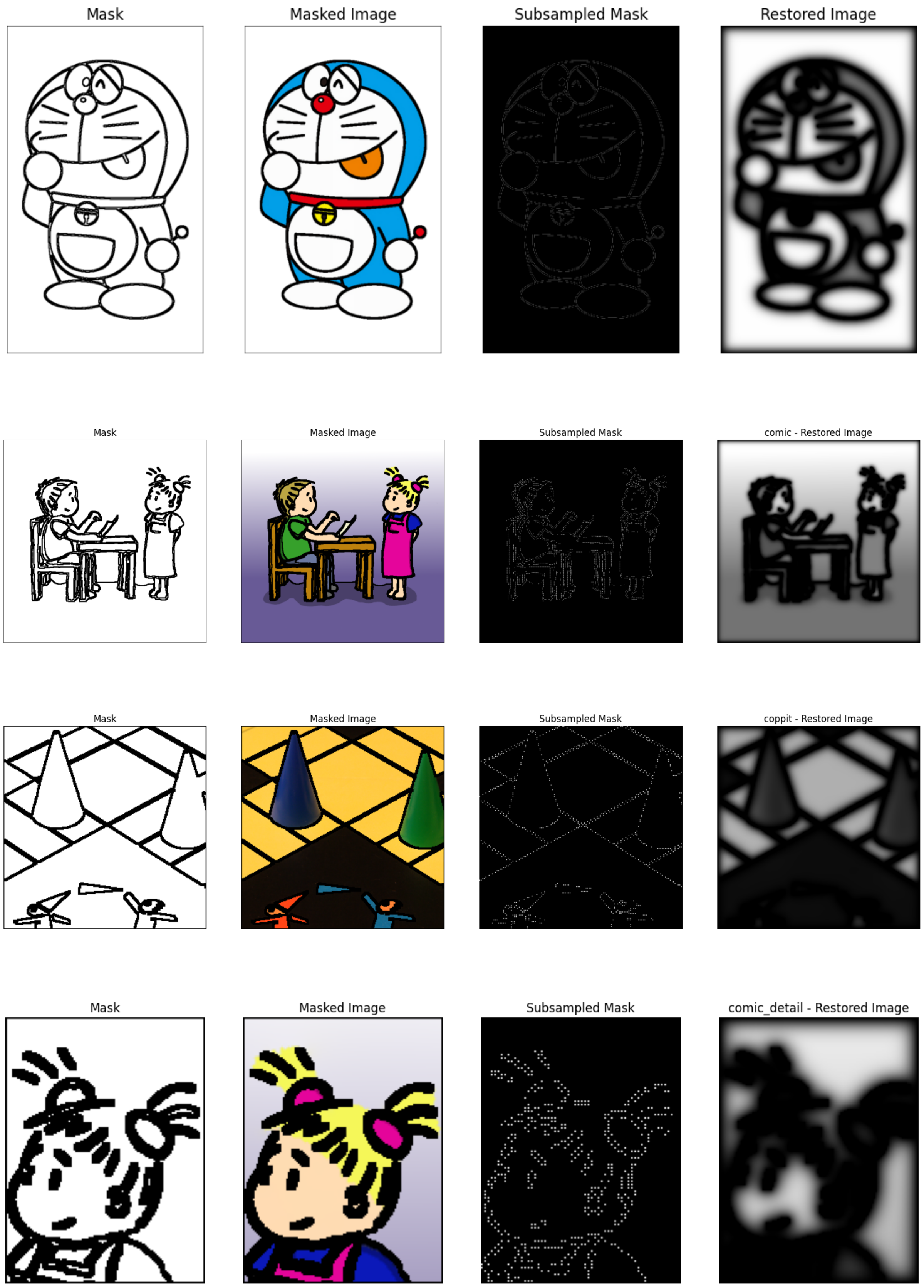


Figure 2: Comparison of mask, masked, subsampled and restored images

6.3 Plots

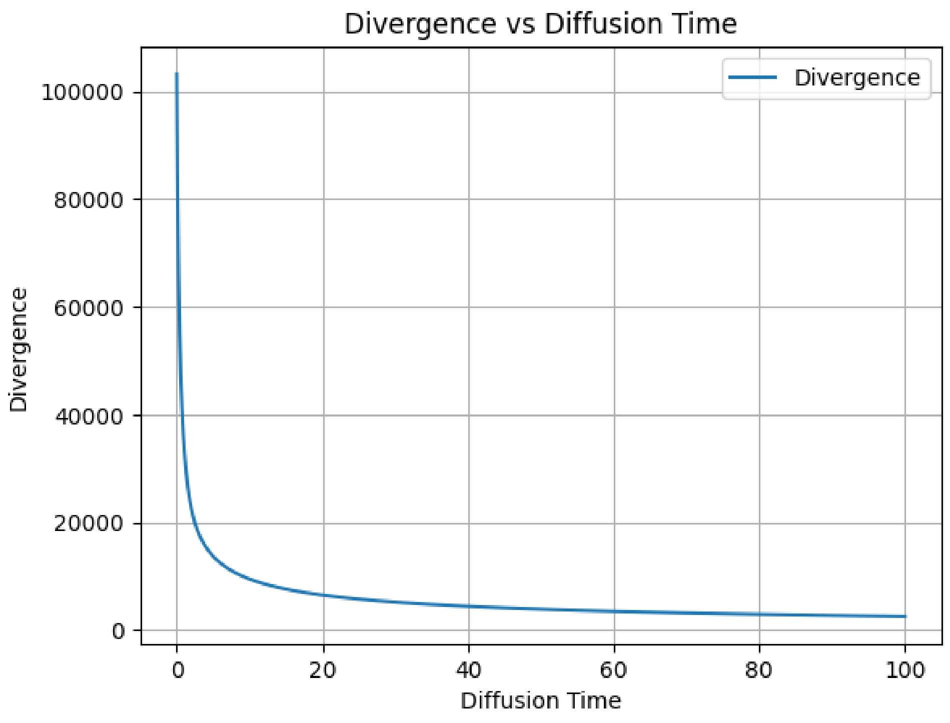


Figure 3: Divergence vs Diffusion Time.

7 Conclusion

The edge-based image compression algorithm achieves moderate compression rates but exhibits limitations in PSNR, particularly for highly detailed images. Improvements can be made by:

- Using advanced edge detection techniques.
- Employing anisotropic diffusion for better reconstruction near edges.
- Exploring alternative compression techniques for sampled data.

8 References

1. Edge-Based Image Compression with Homogeneous Diffusion. *Computer Analysis of Images and Patterns (CAIP 2009)*, 2009. `https://projet.liris.cnrs.fr/imagine/pub/proceedings/CAIP-2009/papers/5702/57020476.pdf`.

2. Mainberger, D. *PhD Thesis*. Saarland University, 2014. `https://www.mia.uni-saarland.de/Publications/mainberger-phd14.pdf`.

3. NumPy Documentation: `https://numpy.org/doc/`.

4. OpenCV Documentation: `https://docs.opencv.org/`.