I. Image Compression using DCT, Huffman and Quantization for Coloured and Greyscale Image

II. Image Compression using Linear SOT, Non- Linear SOT, DCT, DST, DFT and PCA for Coloured and Greyscale Image

Akshat Jain - 22B0690

Bhagyaraju Akepohu - 22B1852

Nikhil Jain - 23M0615

CS663: Digital Image Processing

Department of Computer Science & Engineering

Date: 24 November 2024

Contents

1 Abstract **3**
1.1 Key contributions . . . 3

2 Work Distribution **3**
2.1 Akshat Jain - 22B0690 . . . 3
2.2 Nikhil Jain - 23M0615 . . . 3
2.3 Bhagyaraju Akepohu - 22B1852 . . . 3

3 Problem Statement **3**

4 Datasets **4**

5 Algorithms **4**
5.1 Discrete Cosine Transform (DCT) . . . 4
5.2 Discrete Sine Transform (DST) . . . 4
5.3 Discrete Fourier Transform (DFT) . . . 5
5.4 Compression and Reconstruction . . . 5
5.5 Thresholding . . . 5
5.6 Quantization . . . 5
5.7 Huffman Coding . . . 5
5.8 Principal Component Analysis (PCA) . . . 6
5.9 Linear SOT (DCT-Based) . . . 6
5.10 Non-Linear SOT (Wavelet-Based) . . . 6

6 Graphs and Analysis **7**
6.1 Comparative Graphs and analysis for Greyscale Image and Colour Compression using DCT, Huffman and Quantization . . . 7
6.2 Comparative Graphs and analysis for Greyscale Image and Colour Compression using SOT , Wavelet SOT , PCA and DCT + Huffman + Quantization . . . 9
6.3 Comparative Graphs and analysis for Image Compression using Discrete Cosine Transform, Discrete Sine Transform , Discrete Fourier Transform . . . 12

7 Analysis of Algorithms and Code Implementation **14**
7.1 1. DCT, DFT, CST (Discrete Cosine Transform, Discrete Fourier Transform, Cosine Sine Transform) . . . 14
7.1.1 Algorithm Good Aspects: . . . 14
7.1.2 Algorithm Bad Aspects: . . . 14
7.1.3 Code Implementation Good Aspects: . . . 14
7.1.4 Code Implementation Bad Aspects: . . . 14
7.2 2. SOT (Sparse Optimized Transforms) and PCA (Principal Component Analysis) . . . 14
7.2.1 Algorithm Good Aspects: . . . 14
7.2.2 Algorithm Bad Aspects: . . . 14
7.2.3 Code Implementation Good Aspects: . . . 15
7.2.4 Code Implementation Bad Aspects: . . . 15
7.3 3. DCT + Huffman Coding + Quantization . . . 15
7.3.1 Algorithm Good Aspects: . . . 15
7.3.2 Algorithm Bad Aspects: . . . 15
7.3.3 Code Implementation Good Aspects: . . . 15
7.3.4 Code Implementation Bad Aspects: . . . 15

8 Results and Conclusion **16**
8.1 Results . . . 16
8.1.1 Discrete Cosine Transform (DCT) . . . 16
8.1.2 Discrete Sine Transform (DST) . . . 16
8.1.3 Discrete Fourier Transform (DFT) . . . 16
8.1.4 Principal Component Analysis (PCA) . . . 17
8.1.5 Linear SOT (DCT-based) . . . 17

8.1.6 Non-Linear SOT (Wavelet-based) 17
8.2 Conclusion 17

9 Appendix **18**
9.1 Greyscale Image Compression using DCT Huffman and Quantization 18
9.2 Colour Image Compression using DCT Huffman and Quantization 21
9.3 Code for SOT (Linear and Non-Linear) and PCA 25
9.4 Code for DCT, DST, FFT with Comparative analysis 28

1 Abstract

This report describes the implementation and analysis of multiple lossy image compression algorithms, including Discrete Cosine Transform (DCT), quantization, Huffman coding, Principal Component Analysis (PCA), Linear Signal Optimal Transform (SOT), and Non-Linear SOT. The project leverages both grayscale and colored images from publicly available datasets, implementing different compression techniques for detailed comparative analysis.

1.1 Key contributions

1. Implementing a DCT-based compression pipeline for colored images, coupled with Huffman coding and quantization.
2. Developing and evaluating PCA, Linear SOT (DCT-based), and Non-Linear SOT (Wavelet-based) algorithms
3. Incorporating Discrete Sine Transform (DST) and Discrete Fourier Transform (DFT) methods for enhanced understanding, with metrics such as SSIM, PSNR, RMSE, and BPP.

Creating performance visualization tools (e.g., graphs and image maps) for effective comparison of results. The results demonstrate that each method presents a trade-off between image quality and compression efficiency. DCT-based methods offer robust compression for both grayscale and colored images, while PCA and SOT techniques excel in dimensionality reduction and energy preservation. DST and DFT provide alternative frequency-domain perspectives, enriching the analysis. The study concludes that the choice of algorithm depends on the desired balance between compression ratio and image fidelity.

2 Work Distribution

2.1 Akshat Jain - 22B0690

1. Implementation of Image Compression using DCT, Huffman and Quantization for Coloured images
2. Implementation from Research paper of Discrete Cosine Transform , Discrete Sine Transform and Discrete Fourier Transform with analysis on the basis of SSIM , PSNR , BPP and RMSE [3]

2.2 Nikhil Jain - 23M0615

1. Implementation of Research paper on Image Compression using PCA, Linear SOT , Non-Linear SOT [4]
2. Assisted Implementation of Image Compression using DCT, Huffman and Quantization for Coloured images to Akshat

2.3 Bhagyaraju Akepohu - 22B1852

1. Implementation of Image Compression using DCT , Huffman and Quantization of Greyscale images

3 Problem Statement

Efficient image compression plays a critical role in modern applications, balancing the trade-off between storage requirements, transmission bandwidth, and image quality. This project aims to implement and analyze various lossy image compression algorithms by leveraging frequency-domain and dimensionality-reduction techniques.

The primary goals of this project are:

- Develop a robust DCT-based compression pipeline for grayscale and color images, incorporating quantization and Huffman coding.
- Implement and evaluate alternative compression techniques, including Discrete Sine Transform (DST), Discrete Fourier Transform (DFT), Principal Component Analysis (PCA), and Signal Optimal Transforms (SOT).

- Compare algorithms based on key performance metrics such as Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM), Root Mean Square Error (RMSE), and Bits Per Pixel (BPP).
- Provide visual performance comparisons using tools like graphs and image maps for effective evaluation.

This project emphasizes the trade-offs between image fidelity and compression efficiency by:

- Retaining high-energy components in transformed domains through thresholding techniques.
- Quantizing and encoding the significant coefficients to optimize compression ratios.
- Analyzing the performance on publicly available datasets for both grayscale and color images.

By implementing these techniques, the project seeks to offer a comprehensive understanding of image compression methodologies, with a focus on DCT as a standard and PCA, SOT, DST, and DFT as supplementary methods for enhancing energy preservation and dimensionality reduction. The findings aim to guide the selection of appropriate compression methods based on specific application requirements.

4 Datasets

The algorithm was tested on the following datasets:

- **Grayscale Images:** SIIM COVID-19 Resized 512px PNG dataset (https://www.kaggle.com/competitions/siim-covid19-detection/discussion/239918Link to Dataset).
- **Colored Images:** Flower Classification 104 PNG dataset (https://www.kaggle.com/datasets/alenic/flower-classification-512-pngLink to Dataset).

5 Algorithms

5.1 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) is widely used in image compression as it separates significant low-frequency information from less significant high-frequency components. The 2D DCT of an $N \times N$ image block $f(x, y)$ is given by:

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

where:

$$C(k) = \begin{cases} \sqrt{\frac{1}{2}} & \text{if } k = 0, \\ 1 & \text{otherwise.} \end{cases}$$

The inverse DCT (IDCT) is used for reconstructing the image block from its frequency coefficients $F(u, v)$:

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

DCT achieves compression by retaining high-energy coefficients (typically low-frequency components) and discarding low-energy coefficients.

5.2 Discrete Sine Transform (DST)

The Discrete Sine Transform (DST) is another frequency-domain technique that represents image data using sine functions. The 2D DST for an $N \times N$ image block $f(x, y)$ is expressed as:

$$F(u,v) = \frac{2}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \sin\left[\frac{(x+1)(u+1)\pi}{N+1}\right] \sin\left[\frac{(y+1)(v+1)\pi}{N+1}\right].$$

The inverse DST (IDST) reconstructs the image block $f(x, y)$ from the frequency coefficients $F(u, v)$:

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F(u,v) \sin\left[\frac{(x+1)(u+1)\pi}{N+1}\right] \sin\left[\frac{(y+1)(v+1)\pi}{N+1}\right].$$

The DST is particularly effective in applications where boundary conditions influence compression performance.

5.3 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) transforms spatial domain data into its frequency domain representation by decomposing the image into sinusoidal components. The 2D DFT for an $N \times N$ image block $f(x, y)$ is defined as:

$$F(u,v) = \frac{1}{N^2} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi\left(\frac{ux}{N} + \frac{vy}{N}\right)},$$

where j is the imaginary unit ($j^2 = -1$).
The inverse DFT (IDFT) reconstructs the image block $f(x, y)$ from its frequency coefficients $F(u, v)$:

$$f(x,y) = \frac{1}{N^2} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi\left(\frac{ux}{N} + \frac{vy}{N}\right)}.$$

In DFT, each coefficient contains both magnitude and phase information:

$$|F(u,v)| = \sqrt{\mathrm{Re}(F(u,v))^2 + \mathrm{Im}(F(u,v))^2},$$

$$\mathrm{Phase}(F(u,v)) = \tan^{-1}\left(\frac{\mathrm{Im}(F(u,v))}{\mathrm{Re}(F(u,v))}\right).$$

While DFT provides a comprehensive frequency-domain representation, it is computationally intensive compared to DCT and DST.

5.4 Compression and Reconstruction

For all transformations (DCT, DST, DFT), compression is achieved by:

- Retaining high-energy coefficients based on a threshold or compression ratio.
- Discarding low-energy coefficients (quantized to zero) to enhance compression efficiency.

Reconstruction involves applying the inverse transform to the retained coefficients, restoring an approximation of the original image.

5.5 Thresholding

High-energy coefficients in the transformed domain are retained, while low-energy coefficients are set to zero. This method enables selective compression by prioritizing significant information.

5.6 Quantization

The transformed coefficients are divided by a predefined quantization matrix scaled by a quality factor. Higher quality factors retain more image information, while lower quality factors enhance compression.

5.7 Huffman Coding

Quantized coefficients are encoded using Huffman coding to reduce redundancy. A frequency table of symbols is constructed, and an optimal prefix-free binary code is assigned to each symbol.

5.8 Principal Component Analysis (PCA)

PCA reduces image dimensionality by projecting data onto a lower-dimensional subspace. The image is reconstructed using a specified number of principal components.

5.9 Linear SOT (DCT-Based)

Linear SOT approximates the image by keeping a fraction of high-energy DCT coefficients. Coefficients below a specified threshold are zeroed, followed by an inverse DCT.

5.10 Non-Linear SOT (Wavelet-Based)

Wavelet transform decomposes the image into frequency subbands. The energy-rich coefficients are preserved based on a specified fraction, and inverse wavelet transform reconstructs the image.

6 Graphs and Analysis

6.1 Comparative Graphs and analysis for Greyscale Image and Colour Compression using DCT, Huffman and Quantization

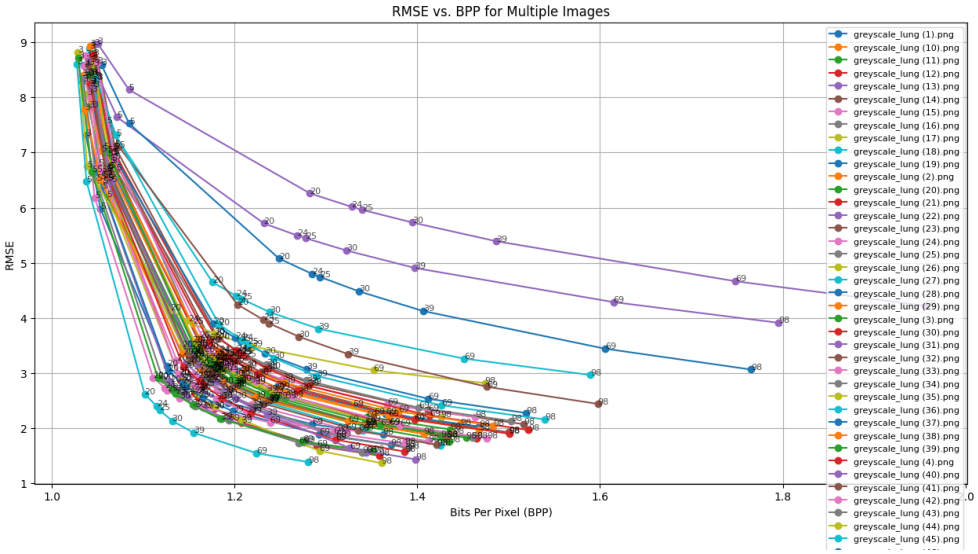

Figure 1: RMSE vs. BPP for varying quality factors for Image Compression using DCT, Huffman and Quantization . Each curve represents a different input image.

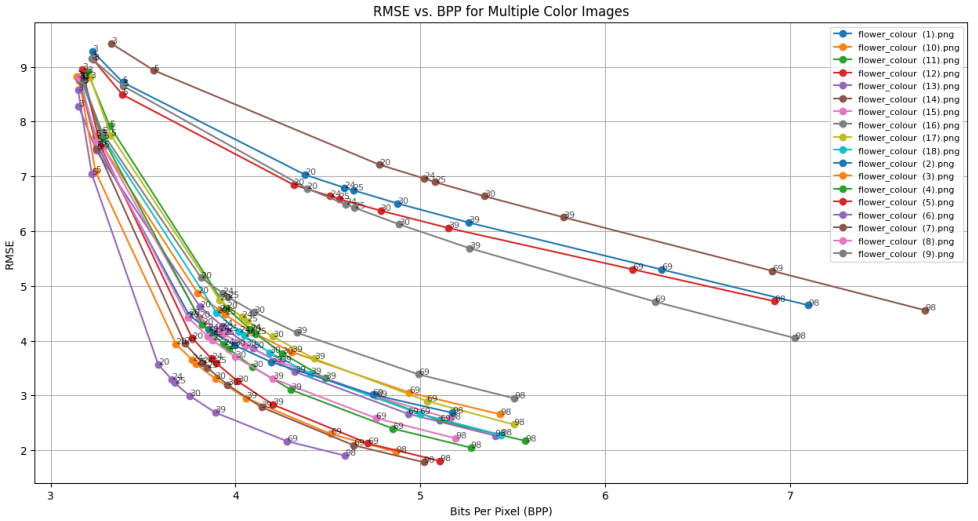

Figure 2: RMSE vs. BPP for varying quality factors for Coloured Image Compression. Each curve represents a different input image.

Figure 3: Images with quality factor = 3, 5, 20, 25, 30, 39, 69, 98 for image compression with DCT, Quantization and Huffman tree.

6.2 Comparative Graphs and analysis for Greyscale Image and Colour Compression using SOT , Wavelet SOT , PCA and DCT + Huffman + Quantization

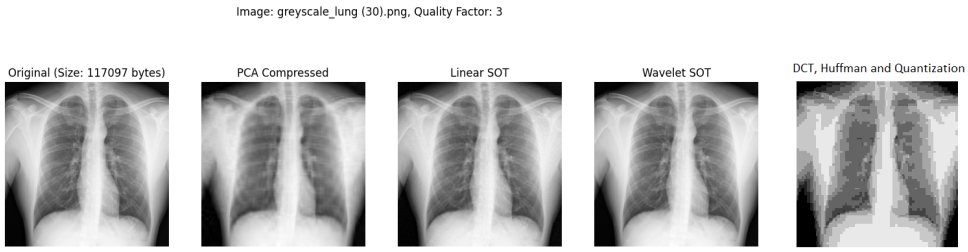

Figure 4: Image Compression using Normal DCT Compression, SOT and PCA at Quality Factor = 3

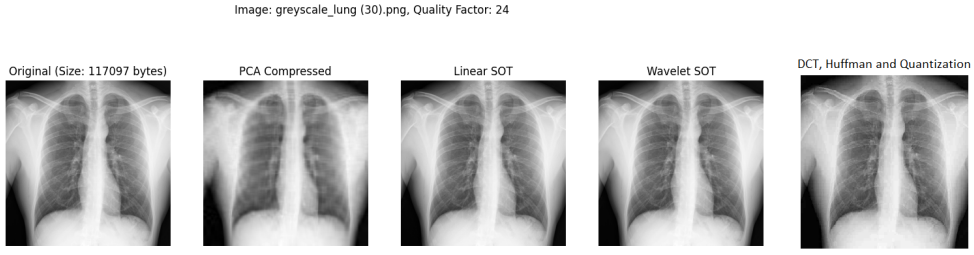

Figure 5: Image Compression using Normal DCT Compression, SOT and PCA at Quality Factor = 24

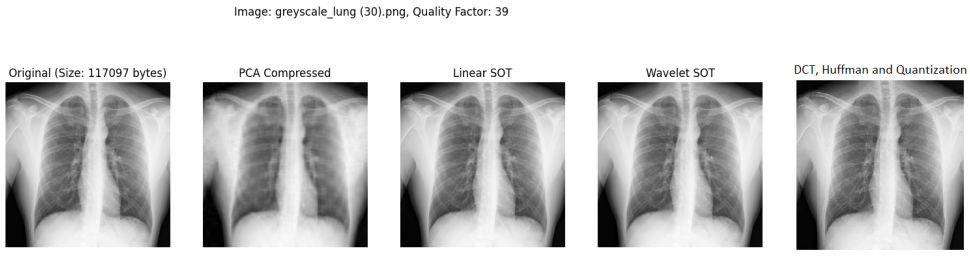

Figure 6: Image Compression using Normal DCT Compression, SOT and PCA at Quality Factor = 39

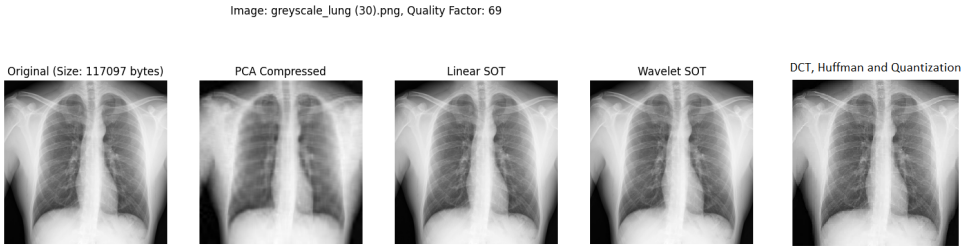

Figure 7: Image Compression using Normal DCT Compression, SOT and PCA at Quality Factor = 69

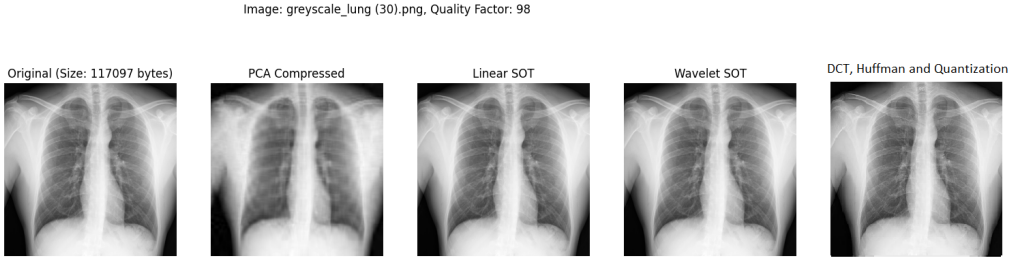

Figure 8: Image Compression using Normal DCT Compression, SOT and PCA at Quality Factor = 98

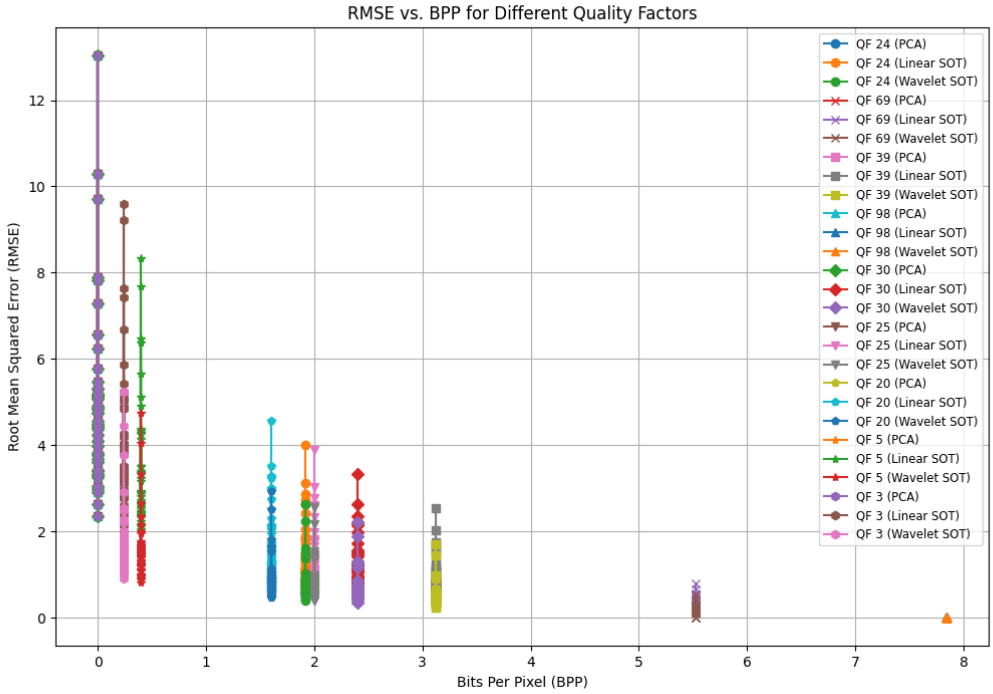

Figure 9: RMSE vs. BPP for varying quality factors for Image Compression using SOT and PCA. Each curve represents a different Method with different quality factors.

6.3 Comparative Graphs and analysis for Image Compression using Discrete Cosine Transform, Discrete Sine Transform , Discrete Fourier Transform

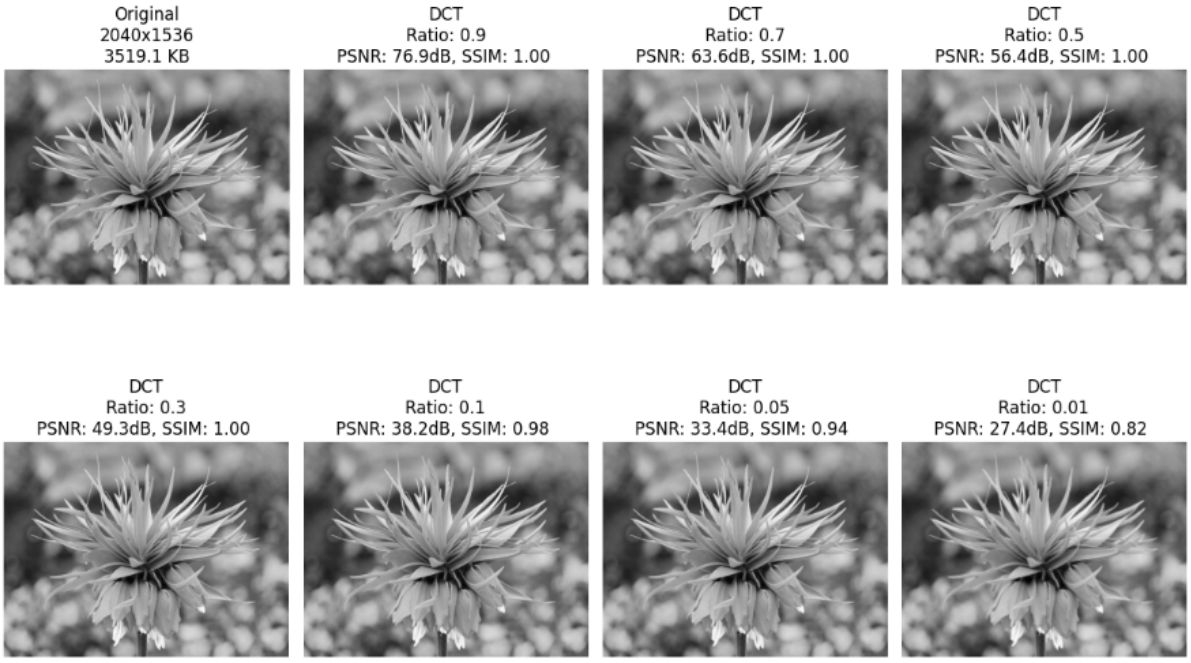

Figure 10: Image Compression using Discrete Cosine Transform for different Compression Ratios

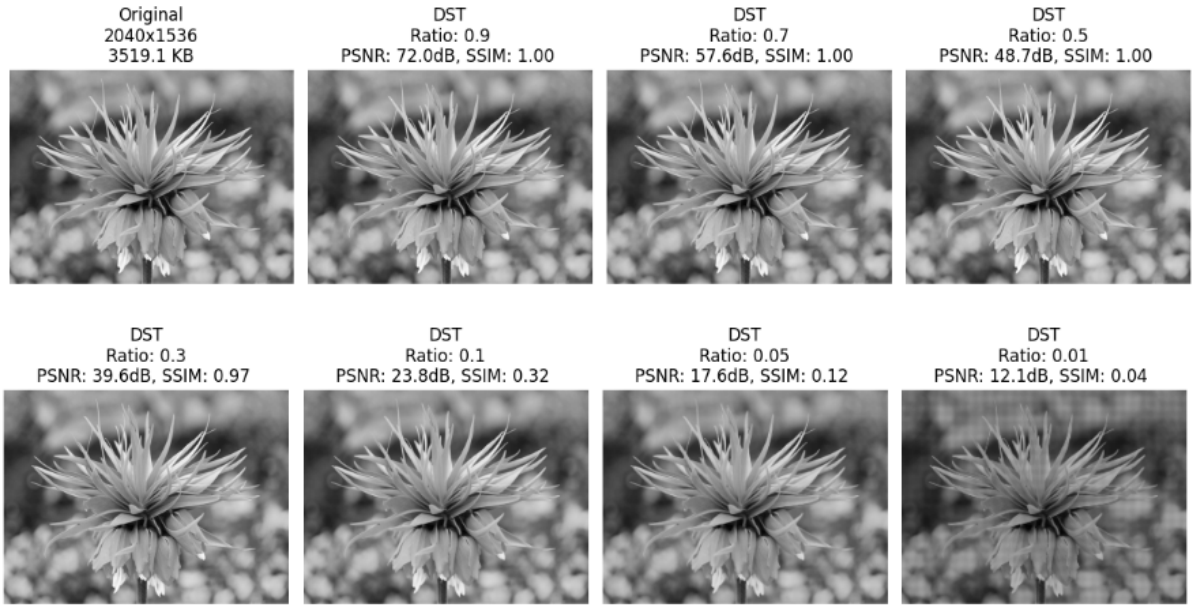

Figure 11: Image Compression using Discrete Sine Transform for different Compression Ratios

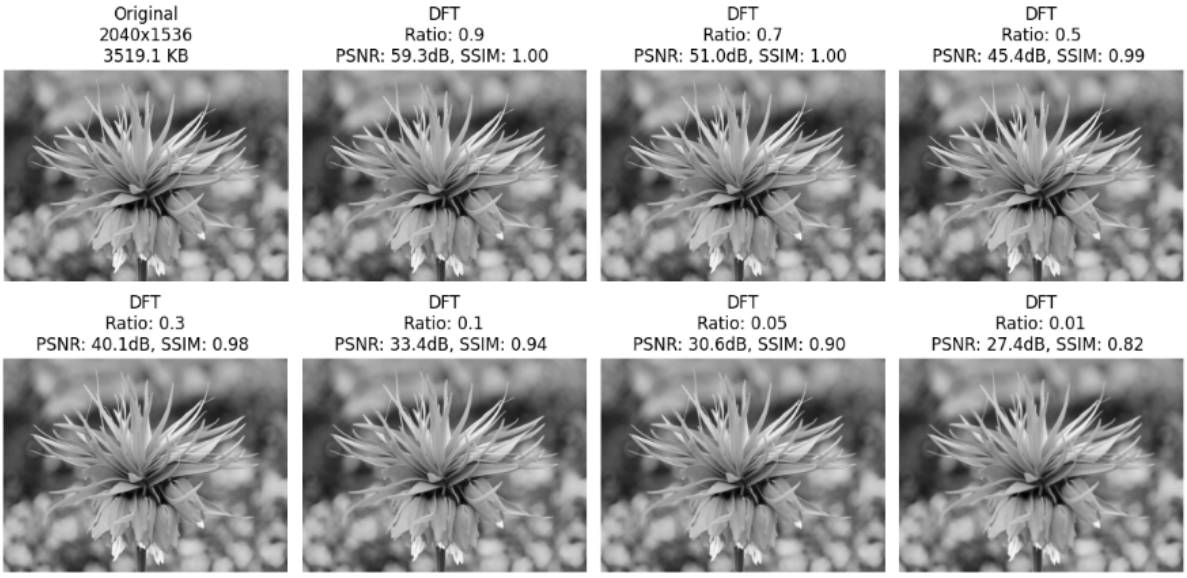

Figure 12: Image Compression using Discrete Fourier Transform for different Compression Ratios

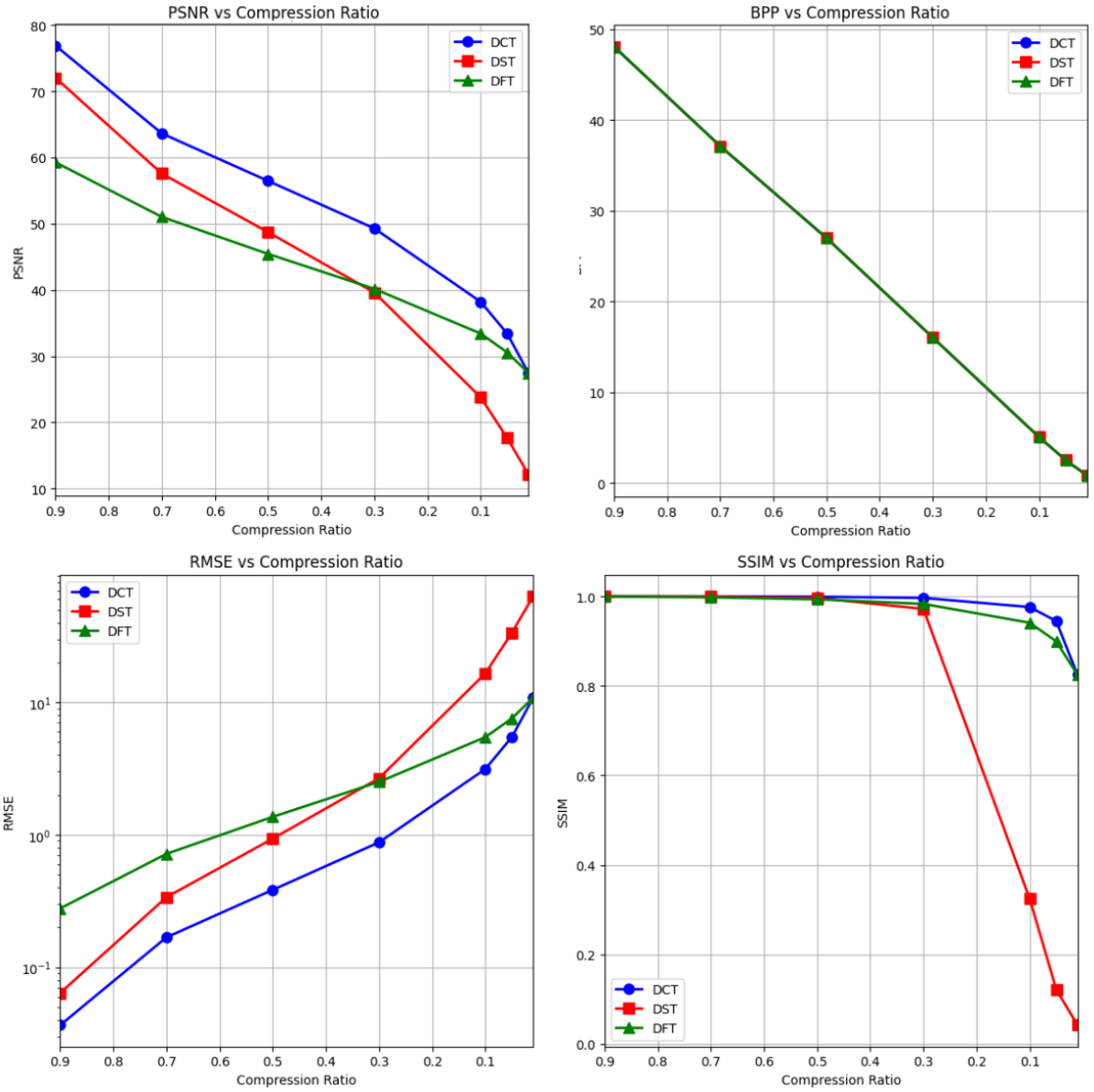

Figure 13: Analysis of Image Compression using DCT, DST, DFT for different Compression Ratios

7 Analysis of Algorithms and Code Implementation

7.1 1. DCT, DFT, CST (Discrete Cosine Transform, Discrete Fourier Transform, Cosine Sine Transform)

7.1.1 Algorithm Good Aspects:

- **Energy Compaction:** DCT and CST efficiently compact the signal's energy, reducing the amount of data that needs to be stored or transmitted.
- **Efficient for Compression:** These transforms are ideal for compression tasks as they concentrate most of the image's significant data into a few coefficients, improving storage efficiency.

7.1.2 Algorithm Bad Aspects:

- **Block Artifacts:** DCT-based methods often introduce blockiness, especially when images are heavily compressed, leading to visible artifacts.
- **Loss of Spatial Information:** DCT and DFT may lose spatial relationships in images, which can degrade the visual quality of reconstructed images.

7.1.3 Code Implementation Good Aspects:

- **Efficient Block Handling:** The code efficiently handles block division and merging, which is crucial for applying DCT and quantization to image data.
- **Use of Libraries:** The implementation leverages well-known libraries like `scipy` for DCT and `numpy` for matrix operations, making the code efficient and reliable.

7.1.4 Code Implementation Bad Aspects:

- **Complexity in Handling Edge Cases:** The code doesn't address edge cases like non-square image sizes or blocks that don't fit perfectly into 8x8 blocks, which can lead to errors.
- **Hardcoded Quantization Matrix:** The use of a static quantization matrix without adaptability for different types of images could limit flexibility and result in suboptimal performance for diverse image types.
- **Block Artifacts Not Handled:** While the block division is done, the code does not implement any anti-blocking measures, which may lead to visible block artifacts in compressed images.

7.2 2. SOT (Sparse Optimized Transforms) and PCA (Principal Component Analysis)

7.2.1 Algorithm Good Aspects:

- **Efficient Dimensionality Reduction:** PCA effectively reduces dimensionality while preserving the variance, leading to a compact representation of the data.
- **Noise Reduction (SOT):** Sparse optimization methods can help reduce noise by focusing on the most significant features of the data, leading to clearer reconstructions.

7.2.2 Algorithm Bad Aspects:

- **Loss of Fine Details:** Both SOT and PCA may result in a loss of fine details, especially when too much dimensionality reduction is performed.
- **Overfitting (SOT):** Sparse optimization techniques can be prone to overfitting if the model is not carefully regularized, reducing generalization.

7.2.3 Code Implementation Good Aspects:

- **Efficient Decomposition and Reconstruction:** The code correctly implements dimensionality reduction and reconstruction, effectively leveraging PCA for image compression.
- **Scalable Approach:** The code's modular structure allows easy adaptation to different data sizes or complexity levels, making it reusable.

7.2.4 Code Implementation Bad Aspects:

- **Limited Implementation of SOT:** While PCA is well-implemented, the sparse optimization part (SOT) is not as clearly defined in the code, making it harder to understand or tweak for specific use cases.
- **No Support for Dynamic Tuning:** The algorithm does not include a way to dynamically adjust the sparsity or PCA components based on the input data, limiting its flexibility.
- **Scalability Challenges:** The implementation may struggle with very large datasets due to the computational complexity of both PCA (eigenvalue decomposition) and sparse optimization.

7.3 3. DCT + Huffman Coding + Quantization

7.3.1 Algorithm Good Aspects:

- **Combining Strengths:** This combination of DCT, Huffman coding, and quantization results in a powerful lossy compression algorithm, balancing efficiency and image quality.
- **Standard in Image Compression:** This approach is the basis of many widely used image compression standards (JPEG), demonstrating its effectiveness and reliability.

7.3.2 Algorithm Bad Aspects:

- **Loss of High-Frequency Data:** The compression inevitably loses high-frequency components, especially with aggressive quantization, leading to visual degradation.
- **Block Artifacts:** The combination of DCT and quantization may lead to block artifacts, especially when high compression ratios are applied, degrading the image quality.

7.3.3 Code Implementation Good Aspects:

- **Combining Multiple Compression Techniques:** The code integrates DCT, Huffman coding, and quantization well, resulting in a comprehensive and effective image compression pipeline.
- **Efficient Compression/Decompression:** The implementation of compression and decompression processes is efficient, ensuring that the algorithm can be applied to real-world image compression tasks.

7.3.4 Code Implementation Bad Aspects:

- **Hardcoded Parameters:** The quantization matrix and other parameters are hardcoded, which limits flexibility and adaptability to different types of images or compression requirements.
- **No Error Handling:** The code does not handle potential errors such as file not found or invalid image formats, which may lead to runtime errors if the input data is not in the expected format.
- **Limited Huffman Tree Handling:** The Huffman coding implementation is relatively basic, and does not include features like tree balancing or more efficient coding techniques that could improve performance.

8 Results and Conclusion

8.1 Results

8.1.1 Discrete Cosine Transform (DCT)

- **High Compression Ratios (0.9 and 0.7):**
  - **PSNR:** At a ratio of 0.9, PSNR is 76.9 dB, and at 0.7, it is 63.6 dB, indicating high-quality retention with minimal perceptual loss.
  - **SSIM:** Remains at 1.0 for both ratios, showing no structural loss.
- **Moderate Compression Ratios (0.5 and 0.3):**
  - **PSNR:** Drops to 56.4 dB (0.5) and 49.3 dB (0.3), indicating slight perceptual degradation.
  - **SSIM:** Maintains near 1.0, ensuring strong structural preservation.
- **Low Compression Ratios (0.1 and 0.01):**
  - **PSNR:** Declines to 38.2 dB (0.1) and 27.4 dB (0.01), indicating noticeable quality loss.
  - **SSIM:** Drops to 0.94 (0.1) and 0.82 (0.01), signifying moderate structural distortion.

Conclusion: DCT performs well across all compression levels, retaining quality even at moderate compression ratios.

8.1.2 Discrete Sine Transform (DST)

- **High Compression Ratios (0.9 and 0.7):**
  - **PSNR:** At 0.9, PSNR is 72.0 dB, and at 0.7, it is 57.6 dB, indicating good quality retention.
  - **SSIM:** Maintains 1.0 for both ratios, preserving structural similarity.
- **Moderate Compression Ratios (0.5 and 0.3):**
  - **PSNR:** Falls to 48.7 dB (0.5) and 39.6 dB (0.3), showing mild perceptual loss.
  - **SSIM:** Drops slightly to 0.97 at a ratio of 0.3.
- **Low Compression Ratios (0.1 and 0.01):**
  - **PSNR:** Reduces to 23.6 dB (0.1) and 12.1 dB (0.01), demonstrating significant quality loss.
  - **SSIM:** Falls drastically to 0.32 (0.1) and 0.04 (0.01), indicating severe structural degradation.

Conclusion: DST offers competitive performance at higher compression ratios but struggles with quality retention at lower ratios.

8.1.3 Discrete Fourier Transform (DFT)

- **High Compression Ratios (0.9 and 0.7):**
  - **PSNR:** Scores of 59.3 dB (0.9) and 51.0 dB (0.7) reflect good quality retention.
  - **SSIM:** Remains at 1.0, ensuring structural preservation.
- **Moderate Compression Ratios (0.5 and 0.3):**
  - **PSNR:** Drops to 45.4 dB (0.5) and 40.1 dB (0.3), revealing perceptible degradation.
  - **SSIM:** Stays above 0.98, signifying strong structural similarity.
- **Low Compression Ratios (0.1 and 0.01):**
  - **PSNR:** Declines to 33.4 dB (0.1) and 27.4 dB (0.01), showing visible quality loss.
  - **SSIM:** Falls to 0.90 (0.1) and 0.82 (0.01), indicating moderate structural loss.

Conclusion: DFT provides consistent structural retention but underperforms DCT in achieving higher PSNR at equivalent ratios.

DCT + Huffman + Quantization

- **Grayscale Images:** Achieved significant compression at moderate quality factors, with minimal perceptual loss.
- **Colored Images:** Retained color fidelity effectively, though RMSE increased significantly at lower quality factors.
- **Metrics:** At quality factors of 30 and above, BPP ranged between 0.5 and 2.0, while RMSE remained below 10. At very low quality factors (e.g., 3), RMSE increased to 20–30, with BPP dropping below 0.2.

8.1.4 Principal Component Analysis (PCA)

- **Grayscale and Colored Images:** Effective dimensionality reduction, though it struggled with fine details at low component counts.
- **Metrics:** PSNR decreased steadily with fewer components. RMSE ranged from 5 to 25 depending on the component count, showing worse performance at fewer components compared to DCT.

8.1.5 Linear SOT (DCT-based)

- **Grayscale Images:** Retained essential features with fewer coefficients but introduced mild artifacts at very high compression levels.
- **Metrics:** RMSE values increased proportionally to compression ratios, ranging from 5 to 20 as coefficients decreased. BPP was competitive with DCT at comparable quality.

8.1.6 Non-Linear SOT (Wavelet-based)

- **Grayscale Images:** Provided better retention of structural details and edges compared to Linear SOT, particularly in high-frequency regions.
- **Metrics:** SSIM values were consistently higher than PCA and Linear SOT, ranging between 0.8 and 0.95 for moderate compression ratios. BPP was comparable to DCT methods.

8.2 Conclusion

- **DCT-based Compression:** Provided a balanced trade-off between computational efficiency, compression ratio, and quality preservation. It outperformed PCA, SOT, and frequency-based methods in most scenarios, making it ideal for practical applications.
- **PCA and SOT Methods:** PCA excelled in dimensionality reduction but lagged in preserving fine details. Non-Linear SOT demonstrated superior structural preservation compared to Linear SOT.
- **DCT:** Demonstrates the best balance between compression efficiency and quality preservation. High PSNR and SSIM values make it ideal for practical applications.
- **DST:** Effective at higher ratios but degrades rapidly at lower ratios. Suitable for use cases where structural similarity is less critical.
- **DFT:** Ensures robust structural retention but requires higher computational resources and lags behind DCT in PSNR.

Overall, the project demonstrates the versatility of lossy compression techniques, emphasizing the importance of selecting the algorithm based on application-specific constraints, such as desired compression ratios, computational resources, and image fidelity.

References

[1] SIIM COVID-19 Resized 512px PNG Dataset. *Kaggle Discussion Forum*. Available at: https://www.kaggle.com/competitions/siim-covid19-detection/discussion/239918

[2] Flower Classification 104 PNG Dataset. *Kaggle Dataset*. Available at: https://www.kaggle.com/datasets/alenic/flower-classification-512-png

[3] Jain, Anil K. (1979). A Sinusoidal Family of Unitary Transforms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(4), 356–365. DOI: 10.1109/TPAMI.1979.4766944.

[4] Sezer, Osman and Guleryuz, Onur and Altunbasak, Yucel (2015). Approximation and Compression With Sparse Orthonormal Transforms. *IEEE Transactions on Image Processing: A Publication of the IEEE Signal Processing Society*, 24. DOI: 10.1109/TIP.2015.2414879

9 Appendix

9.1 Greyscale Image Compression using DCT Huffman and Quantization

```
import os
import numpy as np
import cv2
import heapq
from collections import Counter
import matplotlib.pyplot as plt
import pickle

QUANTIZATION_MATRIX = np.array([
    [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]
])

def build_huffman_tree(frequencies):
    priority_queue = [[freq, [symbol, ""]] for symbol, freq in frequencies.
        items()]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        low = heapq.heappop(priority_queue)
        high = heapq.heappop(priority_queue)
        for pair in low[1:]:
            pair[1] = '0' + pair[1]
        for pair in high[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(priority_queue, [low[0] + high[0]] + low[1:] + high[1:])
    return dict(sorted(heapq.heappop(priority_queue)[1:], key=lambda p: (len(p
        [-1]), p)))

def huffman_encode(data, codebook):
    return "".join(codebook[symbol] for symbol in data)

def huffman_decode(encoded_data, tree):
    reverse_codebook = {v: k for k, v in tree.items()}
    current_code = ""
    decoded_data = []
    for bit in encoded_data:
        current_code += bit
```

```
        if current_code in reverse_codebook:
            decoded_data.append(reverse_codebook[current_code])
            current_code = ""
    return decoded_data

def dct_2d(block):
    N = block.shape[0]
    dct_block = np.zeros_like(block, dtype=float)

    for u in range(N):
        for v in range(N):
            cu = 1/np.sqrt(2) if u == 0 else 1
            cv = 1/np.sqrt(2) if v == 0 else 1

            sum_val = 0
            for x in range(N):
                for y in range(N):
                    sum_val += block[x, y] * np.cos((2*x + 1)*u*np.pi/(2*N)) *
                        np.cos((2*y + 1)*v*np.pi/(2*N))

            dct_block[u, v] = (2/N) * cu * cv * sum_val

    return dct_block

def idct_2d(dct_block):
    N = dct_block.shape[0]
    block = np.zeros_like(dct_block, dtype=float)

    for x in range(N):
        for y in range(N):
            sum_val = 0
            for u in range(N):
                for v in range(N):
                    cu = 1/np.sqrt(2) if u == 0 else 1
                    cv = 1/np.sqrt(2) if v == 0 else 1
                    sum_val += cu * cv * dct_block[u, v] * np.cos((2*x + 1)*u*
                        np.pi/(2*N)) * np.cos((2*y + 1)*v*np.pi/(2*N))

            block[x, y] = (2/N) * sum_val

    return block

def quantize(dct_block, quant_matrix, var):
    quant_matrix_dup = quant_matrix * (50 / var)
    return np.round(dct_block / quant_matrix_dup).astype(int)

def dequantize(quantized_block, quant_matrix, var):
    quant_matrix_dup = quant_matrix * (50 / var)
    return (quantized_block * quant_matrix_dup).astype(float)

def divide_into_blocks(image):
    h, w = image.shape
    blocks = []
    for i in range(0, h, 8):
        for j in range(0, w, 8):
            blocks.append(image[i:i+8, j:j+8])
    return blocks, (h, w)

def merge_blocks(blocks, shape):
    h, w = shape
    image = np.zeros((h, w))
    block_idx = 0
    for i in range(0, h, 8):
```

```
            for j in range(0, w, 8):
                image[i:i+8, j:j+8] = blocks[block_idx]
                block_idx += 1
    return image

def calculate_rmse(original, reconstructed):
    return np.sqrt(np.mean((original - reconstructed) ** 2))

def calculate_bpp(encoded_data, image_shape):
    total_bits = len(encoded_data)
    total_pixels = image_shape[0] * image_shape[1]
    return total_bits / total_pixels

def compress_image(image_path, var):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print(f"Error: Could not load the image {image_path}.")
        return None
    blocks, shape = divide_into_blocks(image)
    quantized_blocks = [quantize(dct_2d(block), QUANTIZATION_MATRIX, var) for
        block in blocks]
    flattened_data = [val for block in quantized_blocks for val in block.
        flatten()]
    frequencies = Counter(flattened_data)
    huffman_tree = build_huffman_tree(frequencies)
    encoded_data = huffman_encode(flattened_data, huffman_tree)
    with open("compressed_image.bin", "wb") as file:
        pickle.dump((encoded_data, huffman_tree, shape), file)

def decompress_image(var):
    with open("compressed_image.bin", "rb") as file:
        encoded_data, huffman_tree, shape = pickle.load(file)
    decoded_data = huffman_decode(encoded_data, huffman_tree)
    blocks = []
    block_size = 64
    for i in range(0, len(decoded_data), block_size):
        block = np.array(decoded_data[i:i+block_size]).reshape((8, 8))
        blocks.append(dequantize(block, QUANTIZATION_MATRIX, var))
    idct_blocks = [idct_2d(block) for block in blocks]
    reconstructed_image = merge_blocks(idct_blocks, shape)
    return np.clip(reconstructed_image, 0, 255).astype(np.uint8)

input_directory = "input"
output_directory = "output_images"
os.makedirs(output_directory, exist_ok=True)

if not os.path.exists(input_directory):
    print(f"Error: Input directory '{input_directory}' does not exist.")
else:
    print(f"Found input directory: {input_directory}")
    print(f"Files in input directory: {os.listdir(input_directory)}")

quality_factors = [24, 69, 39, 98, 30, 25, 20, 5, 3]
image_rmse_bpp = {}

for image_filename in os.listdir(input_directory):
    input_image_path = os.path.join(input_directory, image_filename)

    if not os.path.isfile(input_image_path):
        print(f"Skipping invalid file: {image_filename}")
        continue

    original_image = cv2.imread(input_image_path)
```

```
    if original_image is None:
        print(f"Error: Could not load the image {image_filename}. Skipping...")
        continue

    grayscale_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
    grayscale_path = os.path.join(output_directory, f"{os.path.splitext(
        image_filename)[0]}_grayscale.png")
    cv2.imwrite(grayscale_path, grayscale_image)

    print(f"Processing image: {image_filename}")
    image_results = []

    for q in quality_factors:
        compress_image(grayscale_path, q)
        reconstructed = decompress_image(q)
        rmse = calculate_rmse(grayscale_image, reconstructed)
        with open("compressed_image.bin", "rb") as file:
            encoded_data, _, _ = pickle.load(file)
        bpp = calculate_bpp(encoded_data, grayscale_image.shape)
        image_results.append((q, bpp, rmse))
        output_image_path = os.path.join(output_directory, f"{os.path.splitext(
            image_filename)[0]}_q{q}.png")
        cv2.imwrite(output_image_path, reconstructed)

    image_rmse_bpp[image_filename] = image_results

plt.figure(figsize=(12, 8))
for image_filename, results in image_rmse_bpp.items():
    results = sorted(results, key=lambda x: x[1])
    bpp_values = [result[1] for result in results]
    rmse_values = [result[2] for result in results]
    quality_labels = [result[0] for result in results]

    plt.plot(bpp_values, rmse_values, marker='o', label=f"{image_filename}")
    for i, txt in enumerate(quality_labels):
        plt.annotate(txt, (bpp_values[i], rmse_values[i]), fontsize=8, alpha
            =0.7)

plt.xlabel("Bits Per Pixel (BPP)")
plt.ylabel("RMSE")
plt.title("RMSE vs. BPP for Multiple Images")
plt.legend(loc='upper right', fontsize=8)
plt.grid()
plt.show()
```

9.2 Colour Image Compression using DCT Huffman and Quantization

```
import os
import numpy as np
import cv2
from scipy.fftpack import dct, idct
import heapq
from collections import Counter
import matplotlib.pyplot as plt
import pickle

QUANTIZATION_MATRIX = np.array([
    [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]
])

def build_huffman_tree(frequencies):
    priority_queue = [[freq, [symbol, ""]] for symbol, freq in frequencies.items()]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        low = heapq.heappop(priority_queue)
        high = heapq.heappop(priority_queue)
        for pair in low[1:]:
            pair[1] = '0' + pair[1]
        for pair in high[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(priority_queue, [low[0] + high[0]] + low[1:] + high[1:])
    return dict(sorted(heapq.heappop(priority_queue)[1:], key=lambda p: (len(p[-1]), p)))

def huffman_encode(data, codebook):
    return "".join(codebook[symbol] for symbol in data)

def huffman_decode(encoded_data, tree):
    reverse_codebook = {v: k for k, v in tree.items()}
    current_code = ""
    decoded_data = []
    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codebook:
            decoded_data.append(reverse_codebook[current_code])
            current_code = ""
    return decoded_data

def dct_2d(image_block):
    N = image_block.shape[0]
    dct_matrix = np.zeros_like(image_block, dtype=float)
    for u in range(N):
        for v in range(N):
            alpha_u = 1 / np.sqrt(2) if u == 0 else 1
            alpha_v = 1 / np.sqrt(2) if v == 0 else 1
            sum_val = 0
            for x in range(N):
                for y in range(N):
                    sum_val += image_block[x, y] * np.cos((2 * x + 1) * u * np.pi / (2 * N)) * np.cos((2 * y + 1) * v * np.pi / (2 * N))
            dct_matrix[u, v] = 0.25 * alpha_u * alpha_v * sum_val
    return dct_matrix

def idct_2d(dct_block):
    N = dct_block.shape[0]
    image_matrix = np.zeros_like(dct_block, dtype=float)
    for x in range(N):
        for y in range(N):
            sum_val = 0
            for u in range(N):
                for v in range(N):
                    alpha_u = 1 / np.sqrt(2) if u == 0 else 1
                    alpha_v = 1 / np.sqrt(2) if v == 0 else 1
                    sum_val += alpha_u * alpha_v * dct_block[u, v] * np.cos((2 * x + 1) * u * np.pi / (2 * N)) * np.cos((2 * y + 1) * v * np.pi / (2 * N))
```

```
            image_matrix[x, y] = 0.25 * sum_val
    return image_matrix

def quantize(dct_block, quant_matrix, var):
    quant_matrix_dup = quant_matrix * (50 / var)
    return np.round(dct_block / quant_matrix_dup).astype(int)

def dequantize(quantized_block, quant_matrix, var):
    quant_matrix_dup = quant_matrix * (50 / var)
    return (quantized_block * quant_matrix_dup).astype(float)

def divide_into_blocks(image):
    h, w = image.shape
    blocks = []
    for i in range(0, h, 8):
        for j in range(0, w, 8):
            blocks.append(image[i:i+8, j:j+8])
    return blocks, (h, w)

def merge_blocks(blocks, shape):
    h, w = shape
    image = np.zeros((h, w))
    block_idx = 0
    for i in range(0, h, 8):
        for j in range(0, w, 8):
            image[i:i+8, j:j+8] = blocks[block_idx]
            block_idx += 1
    return image

def calculate_rmse(original, reconstructed):
    return np.sqrt(np.mean((original - reconstructed) ** 2))

def calculate_bpp(encoded_data, image_shape):
    total_bits = len(encoded_data)
    total_pixels = image_shape[0] * image_shape[1]
    return total_bits / total_pixels

def compress_color_image(image_path, var):
    image = cv2.imread(image_path)
    if image is None:
        return None
    channels = cv2.split(image)
    compressed_data = []
    channel_shapes = []
    for channel in channels:
        blocks, shape = divide_into_blocks(channel)
        channel_shapes.append(shape)
        quantized_blocks = [quantize(dct_2d(block), QUANTIZATION_MATRIX, var)
            for block in blocks]
        flattened_data = [val for block in quantized_blocks for val in block.
            flatten()]
        frequencies = Counter(flattened_data)
        huffman_tree = build_huffman_tree(frequencies)
        encoded_data = huffman_encode(flattened_data, huffman_tree)
        compressed_data.append((encoded_data, huffman_tree))
    with open("compressed_color_image.bin", "wb") as file:
        pickle.dump((compressed_data, channel_shapes), file)

def decompress_color_image(var):
    with open("compressed_color_image.bin", "rb") as file:
        compressed_data, channel_shapes = pickle.load(file)
    reconstructed_channels = []
```

```
    for (encoded_data, huffman_tree), shape in zip(compressed_data,
        channel_shapes):
        decoded_data = huffman_decode(encoded_data, huffman_tree)
        blocks = []
        block_size = 64
        for i in range(0, len(decoded_data), block_size):
            block = np.array(decoded_data[i:i+block_size]).reshape((8, 8))
            blocks.append(dequantize(block, QUANTIZATION_MATRIX, var))
        idct_blocks = [idct_2d(block) for block in blocks]
        reconstructed_channel = merge_blocks(idct_blocks, shape)
        reconstructed_channels.append(np.clip(reconstructed_channel, 0, 255).
            astype(np.uint8))
    reconstructed_image = cv2.merge(reconstructed_channels)
    return reconstructed_image

input_directory = "new"
output_directory = "output_images_1"
os.makedirs(output_directory, exist_ok=True)

if os.path.exists(input_directory):
    print(f"Found input directory: {input_directory}")
    print(f"Files in input directory: {os.listdir(input_directory)}")

quality_factors = [24, 69, 39, 98, 30, 25, 20, 5, 3]  # Define your quality
    factors
image_rmse_bpp = {}

for image_filename in os.listdir(input_directory):
    input_image_path = os.path.join(input_directory, image_filename)

    if not os.path.isfile(input_image_path):
        print(f"Skipping invalid file: {image_filename}")
        continue

    original_image = cv2.imread(input_image_path)
    if original_image is None:
        print(f"Error: Could not load the image {image_filename}. Skipping...")
        continue

    print(f"Processing image: {image_filename}")
    image_results = []

    for q in quality_factors:
        compress_color_image(input_image_path, q)
        reconstructed = decompress_color_image(q)
        rmse = calculate_rmse(original_image, reconstructed)
        with open("compressed_color_image.bin", "rb") as file:
            compressed_data, _ = pickle.load(file)
            encoded_data = "".join(data[0] for data in compressed_data)
        bpp = calculate_bpp(encoded_data, original_image.shape[:2])
        image_results.append((q, bpp, rmse))
        output_image_path = os.path.join(output_directory, f"{os.path.splitext(
            image_filename)[0]}_q{q}.png")
        cv2.imwrite(output_image_path, reconstructed)

    image_rmse_bpp[image_filename] = image_results

plt.figure(figsize=(12, 8))
for image_filename, results in image_rmse_bpp.items():
    results = sorted(results, key=lambda x: x[1])
    bpp_values = [result[1] for result in results]
    rmse_values = [result[2] for result in results]
    quality_labels = [result[0] for result in results]
```

```
192
193     plt.plot(bpp_values, rmse_values, marker='o', label=f"{image_filename}")
194     for i, txt in enumerate(quality_labels):
195         plt.annotate(txt, (bpp_values[i], rmse_values[i]), fontsize=8, alpha
            =0.7)
196
197 plt.xlabel("Bits Per Pixel (BPP)")
198 plt.ylabel("RMSE")
199 plt.title("RMSE vs. BPP for Multiple Color Images")
200 plt.legend(loc='upper right', fontsize=8)
201 plt.grid()
202 plt.show()
```

9.3 Code for SOT (Linear and Non-Linear) and PCA

```
1  import numpy as np
2  import cv2
3  import pywt  # PyWavelets library for wavelet transforms
4  from scipy.fftpack import dct, idct
5  from sklearn.decomposition import PCA
6  import matplotlib.pyplot as plt
7  from google.colab import files
8
9  # Perform 2D DCT
10 def dct_2d(image_block):
11     return dct(dct(image_block.T, norm='ortho').T, norm='ortho')
12
13 # Perform 2D inverse DCT
14 def idct_2d(dct_block):
15     return idct(idct(dct_block.T, norm='ortho').T, norm='ortho')
16
17 # Perform PCA-based compression
18 def compress_with_pca(image, num_components):
19     pca = PCA(n_components=num_components)
20     transformed = pca.fit_transform(image)
21     reconstructed = pca.inverse_transform(transformed)
22     return reconstructed, pca.explained_variance_ratio_
23
24 # Perform Linear SOT-based compression (DCT-based approximation)
25 def compress_with_linear_sot(image, keep_fraction):
26     # Perform 2D DCT transform
27     dct_transformed = dct_2d(image)
28
29     # Calculate the threshold based on the compression fraction
30     threshold = np.percentile(np.abs(dct_transformed), (1 - keep_fraction) *
           100)
31
32     # Zero out the coefficients below the threshold
33     dct_transformed[np.abs(dct_transformed) < threshold] = 0
34
35     # Perform inverse DCT to get the compressed image
36     compressed_image = idct_2d(dct_transformed)
37
38     return compressed_image
39
40 # Perform Non-Linear SOT-based compression (Wavelet-based approximation)
41 def compress_with_wavelet(image, keep_fraction, wavelet='haar'):
42     # Perform 2D wavelet transform
43     coeffs2 = pywt.dwt2(image, wavelet)
44     LL, (LH, HL, HH) = coeffs2
45
46     # Flatten the high-frequency coefficients
```

```
    coeffs_flat = np.concatenate([LH.flatten(), HL.flatten(), HH.flatten()])

    # Sort the coefficients by magnitude (energy) and determine the threshold
    sorted_indices = np.argsort(np.abs(coeffs_flat))[::-1]
    num_coeffs_to_keep = int(len(coeffs_flat) * keep_fraction)
    thresholded_indices = sorted_indices[:num_coeffs_to_keep]

    # Set all other coefficients to zero
    mask = np.zeros_like(coeffs_flat, dtype=bool)
    mask[thresholded_indices] = True

    # Create new coefficients by applying the mask to the original high-frequency components
    coeffs_flat[~mask] = 0  # Set the "non-kept" coefficients to zero

    # Reorganize the modified coefficients back into the (LH, HL, HH) blocks
    LH = np.reshape(coeffs_flat[:len(LH.flatten())], LH.shape)
    HL = np.reshape(coeffs_flat[len(LH.flatten()):len(LH.flatten()) + len(HL.flatten())], HL.shape)
    HH = np.reshape(coeffs_flat[len(LH.flatten()) + len(HL.flatten()):], HH.shape)

    # Perform inverse wavelet transform to get the compressed image
    compressed_image = pywt.idwt2((LL, (LH, HL, HH)), wavelet)

    return compressed_image

# Calculate RMSE
def calculate_rmse(original, reconstructed):
    return np.sqrt(np.mean((original - reconstructed) ** 2))

# Calculate Bits Per Pixel (BPP)
def calculate_bpp(image, num_bits):
    return num_bits / (image.shape[0] * image.shape[1])

# Main function to handle the comparison
def main():
    # Upload multiple images
    print("Please upload multiple images...")
    uploaded = files.upload()  # Use Colab's file upload
    image_paths = list(uploaded.keys())

    if not image_paths:
        print("No images were uploaded.")
        return

    # Parameters
    pca_components = 20  # Number of PCA components

    # Allow the user to input compression factors for both SOT methods
    sot_keep_fraction_linear = float(input("Enter the compression factor for Linear SOT (0 to 1, where 1 is no compression, 0 is maximum compression): "))
    sot_keep_fraction_wavelet = float(input("Enter the compression factor for Wavelet SOT (0 to 1, where 1 is no compression, 0 is maximum compression): "))

    # Metrics storage
    metrics = []

    # Process each image
    for image_path in image_paths:
        # Read image from uploaded data
```

```
        image_data = uploaded[image_path]
        np_image = np.frombuffer(image_data, np.uint8)
        image = cv2.imdecode(np_image, cv2.IMREAD_GRAYSCALE)

        if image is None:
            print(f"Could not load image: {image_path}")
            continue

        # Original size (before compression)
        original_size = len(image_data)  # Size in bytes of the original
            uploaded image

        # PCA Compression
        reconstructed_pca, variance_ratio = compress_with_pca(image,
            pca_components)
        rmse_pca = calculate_rmse(image, reconstructed_pca)
        bpp_pca = calculate_bpp(image, pca_components * 8)
        pca_size = len(cv2.imencode('.jpg', np.clip(reconstructed_pca, 0, 255))
            [1])  # Approx size of PCA image

        # Linear SOT Compression
        reconstructed_linear_sot = compress_with_linear_sot(image,
            sot_keep_fraction_linear)
        rmse_linear_sot = calculate_rmse(image, reconstructed_linear_sot)
        bpp_linear_sot = calculate_bpp(image, sot_keep_fraction_linear * image.
            size * 8)
        linear_sot_size = len(cv2.imencode('.jpg', np.clip(
            reconstructed_linear_sot, 0, 255))[1])  # Approx size of Linear SOT
            image

        # Non-Linear SOT (Wavelet Compression)
        reconstructed_wavelet_sot = compress_with_wavelet(image,
            sot_keep_fraction_wavelet)
        rmse_wavelet_sot = calculate_rmse(image, reconstructed_wavelet_sot)
        bpp_wavelet_sot = calculate_bpp(image, sot_keep_fraction_wavelet *
            image.size * 8)
        wavelet_sot_size = len(cv2.imencode('.jpg', np.clip(
            reconstructed_wavelet_sot, 0, 255))[1])  # Approx size of Wavelet
            SOT image

        # Store metrics
        metrics.append({
            "image": image_path,
            "rmse_pca": rmse_pca,
            "bpp_pca": bpp_pca,
            "rmse_linear_sot": rmse_linear_sot,
            "bpp_linear_sot": bpp_linear_sot,
            "rmse_wavelet_sot": rmse_wavelet_sot,
            "bpp_wavelet_sot": bpp_wavelet_sot,
            "original_size": original_size,
            "pca_size": pca_size,
            "linear_sot_size": linear_sot_size,
            "wavelet_sot_size": wavelet_sot_size
        })

        # Display images
        plt.figure(figsize=(15, 5))
        plt.subplot(1, 4, 1)
        plt.imshow(image, cmap='gray')
        plt.title(f"Original (Size: {original_size} bytes)")
        plt.axis("off")

        plt.subplot(1, 4, 2)
```

```
        plt.imshow(np.clip(reconstructed_pca, 0, 255), cmap='gray')
        plt.title(f"PCA Compressed (Size: {pca_size} bytes)")
        plt.axis("off")

        plt.subplot(1, 4, 3)
        plt.imshow(np.clip(reconstructed_linear_sot, 0, 255), cmap='gray')
        plt.title(f"Linear SOT (Size: {linear_sot_size} bytes)")
        plt.axis("off")

        plt.subplot(1, 4, 4)
        plt.imshow(np.clip(reconstructed_wavelet_sot, 0, 255), cmap='gray')
        plt.title(f"Wavelet SOT (Size: {wavelet_sot_size} bytes)")
        plt.axis("off")

        plt.suptitle(f"Image: {image_path}")
        plt.show()

    # Display comparison table
    print("Comparison Metrics:")
    print("{:<30} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10} {:<10}".
        format(
        "Image", "RMSE PCA", "BPP PCA", "RMSE Linear SOT", "BPP Linear SOT", "
            RMSE Wavelet SOT", "BPP Wavelet SOT", "Original Size", "PCA Size"))
    for m in metrics:
        print("{:<30} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4f} {:<10.4
            f} {:<10} {:<10}".format(
            m["image"], m["rmse_pca"], m["bpp_pca"], m["rmse_linear_sot"], m["
                bpp_linear_sot"],
            m["rmse_wavelet_sot"], m["bpp_wavelet_sot"], m["original_size"], m[
                "pca_size"]
        ))

if __name__ == "__main__":
    main()
```

9.4 Code for DCT, DST, FFT with Comparative analysis

```
import numpy as np
import os
import matplotlib.pyplot as plt
from scipy.fftpack import idct, idst
import time
from google.colab import files
import cv2
from skimage.metrics import structural_similarity as ssim

class ImageCompressor:
    def __init__(self):
        self.supported_transforms = {
            'dct': (self._dct_2d, self._idct_2d),
            'dst': (self._dst_2d, self._idst_2d),
            'dft': (self._dft_2d, self._idft_2d)
        }

    def _dct_2d(self, img):
        def dct_1d(array):
            N = len(array)
            result = np.zeros(N)
            for k in range(N):
                coeff = np.sqrt(1 / N) if k == 0 else np.sqrt(2 / N)
                result[k] = coeff * np.sum(array * np.cos(np.pi * (2 * np.
                    arange(N) + 1) * k / (2 * N)))
```

```
            return result
        return np.apply_along_axis(dct_1d, axis=0, arr=np.apply_along_axis(
            dct_1d, axis=1, arr=img))

    def _idct_2d(self, coefficients):
        def idct_1d(array):
            N = len(array)
            result = np.zeros(N)
            for n in range(N):
                result[n] = np.sum(array * np.cos(np.pi * (2 * n + 1) * np.
                    arange(N) / (2 * N)) * (
                     [np.sqrt(1 / N)] + [np.sqrt(2 / N)] * (N - 1)))
            return result
        return np.apply_along_axis(idct_1d, axis=0, arr=np.apply_along_axis(
            idct_1d, axis=1, arr=coefficients))

    def _dst_2d(self, img):
        def dst_1d(array):
            N = len(array)
            result = np.zeros(N)
            for k in range(N):
                result[k] = np.sum(array * np.sin(np.pi * (np.arange(1, N + 1)
                    * (k + 1)) / (N + 1)))
            return result
        return np.apply_along_axis(dst_1d, axis=0, arr=np.apply_along_axis(
            dst_1d, axis=1, arr=img))

    def _idst_2d(self, coefficients):
        def idst_1d(array):
            N = len(array)
            result = np.zeros(N)
            for n in range(N):
                result[n] = np.sum(array * np.sin(np.pi * (n + 1) * (np.arange
                    (1, N + 1)) / (N + 1)))
            return result
        return np.apply_along_axis(idst_1d, axis=0, arr=np.apply_along_axis(
            idst_1d, axis=1, arr=coefficients))

    def _dft_2d(self, img):
        def dft_1d(array):
            N = len(array)
            result = np.zeros(N, dtype=complex)
            for k in range(N):
                result[k] = np.sum(array * np.exp(-2j * np.pi * k * np.arange(N
                    ) / N))
            return result
        return np.apply_along_axis(dft_1d, axis=0, arr=np.apply_along_axis(
            dft_1d, axis=1, arr=img))

    def _idft_2d(self, coefficients):
        def idft_1d(array):
            N = len(array)
            result = np.zeros(N, dtype=complex)
            for n in range(N):
                result[n] = np.sum(array * np.exp(2j * np.pi * n * np.arange(N)
                     / N)) / N
            return result
        return np.real(np.apply_along_axis(idft_1d, axis=0, arr=np.
            apply_along_axis(idft_1d, axis=1, arr=coefficients)))

    def compress_image(self, image, transform_type='dct', compression_ratio
        =0.1, block_size=8):
        if transform_type not in self.supported_transforms:
```

```
            raise ValueError(f"Transform type {transform_type} not supported")
        transform_func, inverse_transform_func = self.supported_transforms[
            transform_type]
        h, w = image.shape
        h_pad = (h + block_size - 1) // block_size * block_size
        w_pad = (w + block_size - 1) // block_size * block_size
        padded = np.zeros((h_pad, w_pad))
        padded[:h, :w] = image
        compressed = np.zeros_like(padded)
        coeffs_to_keep = max(1, int(block_size * block_size * compression_ratio
            ))
        total_nonzero_coeffs = 0
        for i in range(0, h_pad, block_size):
            for j in range(0, w_pad, block_size):
                block = padded[i:i+block_size, j:j+block_size].astype(np.
                    float64)
                transformed = transform_func(block)
                indices = np.argsort(np.abs(transformed.flatten()))[-
                    coeffs_to_keep:]
                mask = np.zeros_like(transformed)
                mask.flat[indices] = 1
                filtered = transformed * mask
                total_nonzero_coeffs += np.count_nonzero(filtered)
                reconstructed = inverse_transform_func(filtered)
                reconstructed = np.clip(reconstructed, 0, 255)
                compressed[i:i+block_size, j:j+block_size] = reconstructed
        return compressed[:h, :w], total_nonzero_coeffs

    def calculate_metrics(self, original, compressed, original_bytes=None,
        total_nonzero_coeffs=None):
        original = original.astype(np.float64)
        compressed = compressed.astype(np.float64)
        compressed = np.clip(compressed, 0, 255)
        mse = np.mean((original - compressed) ** 2)
        rmse = np.sqrt(mse)
        psnr = float('inf') if mse == 0 else 20 * np.log10(255.0 / np.sqrt(mse)
            )
        h, w = original.shape
        original_size = original.size * 8
        if total_nonzero_coeffs is not None:
            position_bits = np.ceil(np.log2(h * w))
            compressed_size_bits = total_nonzero_coeffs * (32 + position_bits)
        else:
            compressed_coeffs = np.count_nonzero(compressed)
            compressed_size_bits = compressed_coeffs * 32
        compressed_size_bytes = int(np.ceil(compressed_size_bits / 8))
        compression_ratio = original_size / compressed_size_bits if
            compressed_size_bits > 0 else float('inf')
        ssim_value = ssim(original, compressed, data_range=compressed.max() -
            compressed.min())
        bpp = compressed_size_bits / original.size
        if original_bytes:
            size_reduction = ((original_bytes - compressed_size_bytes) /
                original_bytes) * 100
        else:
            size_reduction = ((original_size // 8) - compressed_size_bytes) / (
                original_size // 8) * 100
        return {
            'MSE': mse,
            'RMSE': rmse,
            'PSNR': psnr,
            'Compression Ratio': compression_ratio,
            'BPP': bpp,
```

```
            'SSIM': ssim_value,
            'Image Dimensions': f"{w}x{h}"
        }

def main():
    print("Upload an image file...")

    uploaded = files.upload()
    filename = list(uploaded.keys())[0]
    original_bytes = len(uploaded[filename])
    img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error loading image")
        return
    h, w = img.shape

    compressor = ImageCompressor()
    transforms = ['dct', 'dst', 'dft']
    compression_ratios = [0.9, 0.7, 0.5, 0.3, 0.1, 0.05, 0.01]
    plt.figure(figsize=(24, 15))
    output_dir = "compressed_images"
    os.makedirs(output_dir, exist_ok=True)

    for idx, transform in enumerate(transforms):
        plt.subplot(len(transforms), len(compression_ratios) + 1, idx * (len(
            compression_ratios) + 1) + 1)
        plt.imshow(img, cmap='gray')
        plt.title(f'Original\n{w}x{h}\n{original_bytes / 1024:.1f} KB')
        plt.axis('off')
    results = {}

    for transform_idx, transform in enumerate(transforms):
        for ratio_idx, ratio in enumerate(compression_ratios):
            compressed, total_nonzero_coeffs = compressor.compress_image(img,
                transform, ratio)
            metrics = compressor.calculate_metrics(img, compressed,
                original_bytes, total_nonzero_coeffs)
            results[f"{transform}_{ratio}"] = metrics
            output_filename = os.path.join(output_dir, f"{transform}_{ratio}.
                jpg")
            cv2.imwrite(output_filename, compressed)
            plot_idx = transform_idx * (len(compression_ratios) + 1) +
                ratio_idx + 2
            plt.subplot(len(transforms), len(compression_ratios) + 1, plot_idx)
            plt.imshow(compressed, cmap='gray')
            plt.title(f"{transform.upper()} {ratio * 100:.1f}%\n"
                      f"{metrics['BPP']:.2f} bpp\n"
                      f"PSNR: {metrics['PSNR']:.1f}\n"
                      f"SSIM: {metrics['SSIM']:.2f}")
            plt.axis('off')

    plt.tight_layout()

    plt.show()

if __name__ == "__main__":
    main()
```