CS663: Digital Image Processing

Project: Image Compression Engine

Team Members:
Hardik Khariwal (22B3954)
Prajwal Nayak (22B4246)
Reeyansh Shah (22B0412)

November 24, 2024

1 Problem Statement 1

The goal of this project is to build an image compression engine similar to the JPEG algorithm. The engine should effectively compress grayscale images while maintaining reasonable image quality. The implementation includes computing 2D Discrete Cosine Transform (DCT) coefficients for non-overlapping image patches, quantizing these coefficients, applying Huffman encoding, and finally reconstructing the image. The performance of the algorithm is evaluated using the Root Mean Squared Error (RMSE) versus Bits Per Pixel (BPP) curve.

1.1 Input Image

The algorithm begins with a grayscale image, which is a 2D array of pixel values, typically ranging from 0 to 255. In this project, we use the `camera` image from the `skimage` library, a common test image for image processing algorithms. The image is then shifted by 128 to center the pixel values around zero, which helps in the DCT process by reducing numerical instability.

1.2 2D Discrete Cosine Transform (DCT)

The first significant step in the compression process is the transformation of the image from the spatial domain to the frequency domain. This is done by applying the 2D DCT to non-overlapping 8x8 blocks of the image. The DCT helps separate the image into components of varying frequencies:

By transforming the image into frequency components, we can discard high-frequency components (which are often less perceptible to the human eye) and focus on the low-frequency components, which contribute the most to the image's visual quality.

1.2.1 Steps

1. **Block Processing:** The image is divided into non-overlapping 8x8 blocks. For each block, a 2D DCT is applied:
   - First, a 1D DCT is applied to each row (i.e., horizontal direction).
   - Then, a 1D DCT is applied to each column of the result (i.e., vertical direction).
2. **DCT Formula:** The DCT equation is given by:

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

   Where $x(x,y)$ is the pixel value, and $X(u,v)$ represents the DCT coefficients. The coefficients $\alpha(u)$ and $\alpha(v)$ are normalization factors to scale the DCT coefficients, and they are given by:

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } u = 0 \\ \frac{2}{\sqrt{N}} & \text{otherwise} \end{cases}$$

   After applying the DCT, the 8x8 block is transformed into DCT coefficients. The resulting coefficients represent the image in terms of its frequency components.

1.3 Quantization

Quantization is the process of reducing the precision of the DCT coefficients. This is done to reduce the size of the data, which is crucial for compression. The idea is to decrease the less important DCT coefficients more aggressively and retain the more important ones.

1.3.1 Quantization Matrix

The DCT coefficients are divided by a quantization matrix and rounded to the nearest integer. The quantization matrix controls how much each frequency component is compressed. Higher values in the quantization matrix correspond to more aggressive compression (i.e., the corresponding DCT coefficients are quantized to a smaller range). The quantization matrix used in JPEG compression is typically designed to match the human visual system's sensitivity to various frequencies.

The quantization matrix for QF=50 is given as:

$$Q_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

1.3.2 Quantization Process

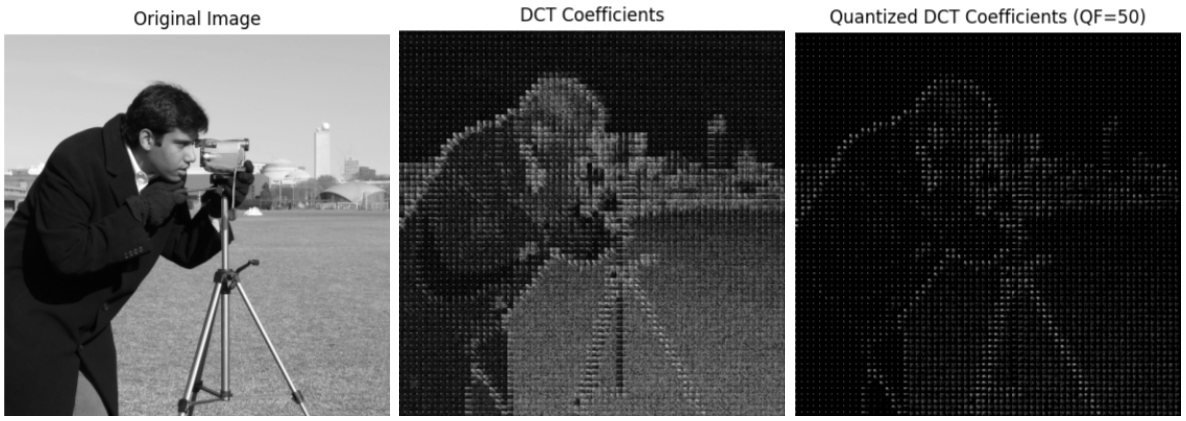
For each DCT coefficient, the value is divided by the corresponding entry in the quantization matrix and rounded to the nearest integer. The quantization matrix is scaled based on the quality factor QF. Lower values of QF result in more aggressive compression (larger values in the quantization matrix), leading to a smaller file size but a loss in image quality. Higher values of QF reduce the quantization, leading to better image quality but less compression.

$$Q(u, v) = \text{round}\left(\frac{D(u, v)}{M(u, v)}\right)$$

Where $D(u, v)$ is the DCT coefficient and $M(u, v)$ is the value in the quantization matrix $Q(u, v)$.

1.4 Entropy Encoding (Huffman Encoding)

After quantization, the DCT coefficients are encoded using Huffman coding, a lossless data compression algorithm. Huffman coding is used to efficiently store the quantized DCT coefficients by assigning shorter codes to more frequent coefficients and longer codes to less frequent coefficients.



1.4.1 Building the Huffman Tree

A frequency analysis of the quantized DCT coefficients is performed. A heap is used to build a binary tree where each leaf node represents a unique quantized DCT value, and the internal nodes represent the merging of two lower-frequency nodes. The tree structure is traversed to generate a set of variable-length codes for each quantized coefficient. These codes are then used to represent the quantized values in a more compressed form. **The huffman encoded data is stored as a .json form which data is again read back and can be used to reconstruct the data.**

1.5 Reconstruction (Decompression)

The decompression process involves reversing the compression steps:

Huffman Decoding: The compressed bitstream is decoded using a pre-defined Huffman table that maps encoded values back to the quantized DCT coefficients. This step effectively reverses the entropy encoding, recovering the numerical frequency-domain data for each 8x8 block. The result is a matrix of quantized coefficients that serves as input for the following stages of decompression.

Dequantization: In this step, the quantized DCT coefficients are multiplied by their corresponding values in the quantization matrix. This process approximates the original frequency-domain representation by reversing the simplification introduced during quantization. While dequantization restores much of the original data, some details are permanently lost due to the lossy nature of compression.

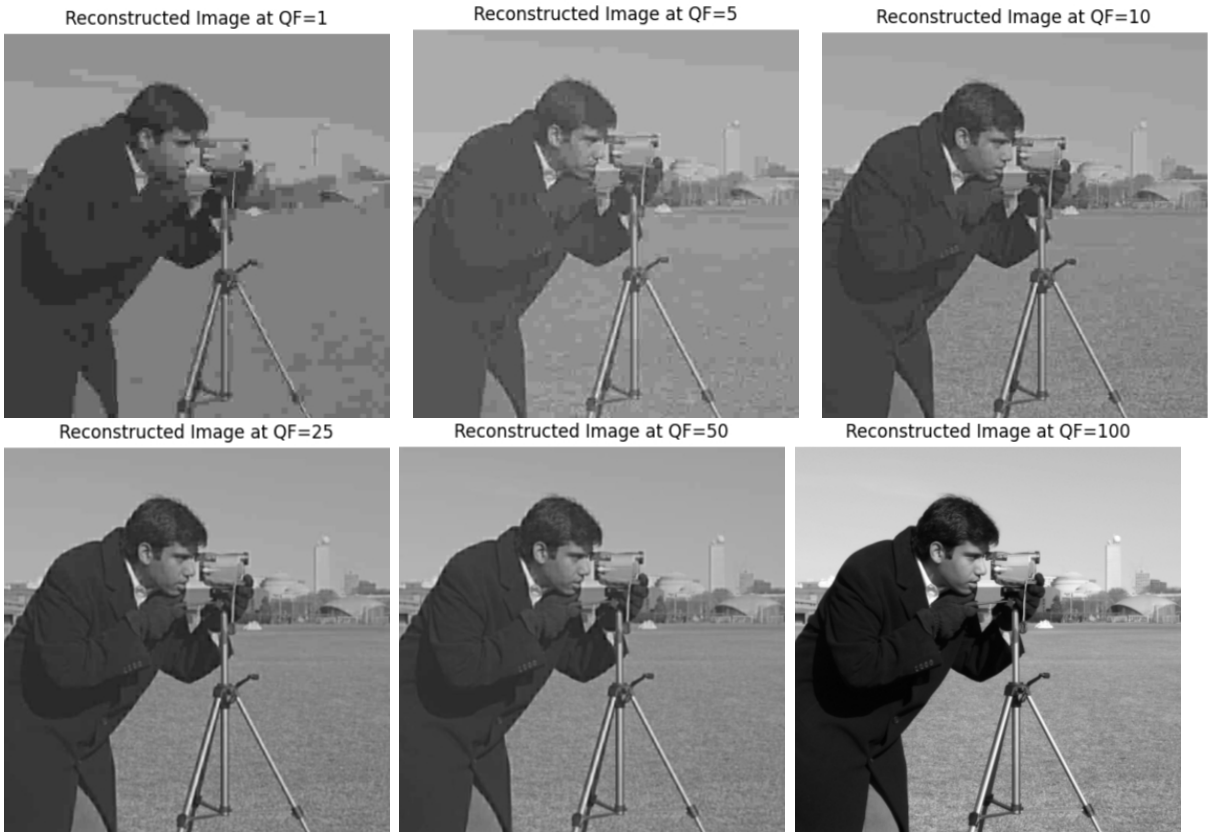
Inverse DCT: The inverse DCT (IDCT) is applied to each 8x8 block to transform the data back from the frequency domain to the spatial domain. This reconstructs the image, which is then displayed.

1.6 Quality Factor and Compression Performance

The quality factor (QF) controls the balance between image quality and compression efficiency:

- Lower QF: Higher compression but lower image quality.

- Higher QF: Better quality but lower compression.



The **Root Mean Squared Error (RMSE)** is used to measure the image quality:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(I_{\text{orig}}(i) - I_{\text{reconstructed}}(i))^2}$$

Where I_{orig} is the original image, $I_{\text{reconstructed}}$ is the compressed-reconstructed image, and N is the total number of pixels in the image.

The **Bits Per Pixel (BPP)** is used to measure the compression efficiency:

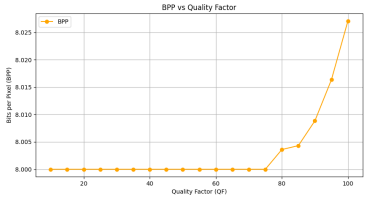
$$\text{BPP} = \frac{\text{Total number of bits used for encoding}}{\text{Total number of pixels in the image}}$$

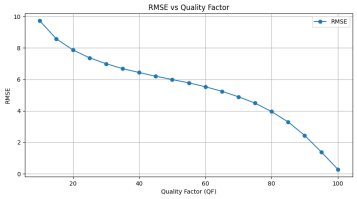
1.7 Evaluation

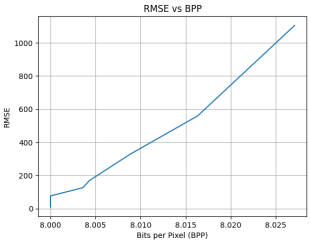
RMSE vs BPP Curve: We plot the relationship between RMSE and BPP for various QF values. This helps in analyzing the trade-off between compression efficiency (file size) and image quality. As QF increases, RMSE decreases (better quality), but BPP increases (larger file size).

1.8 Conclusion

- The algorithm successfully compresses the image using DCT and Huffman encoding.
- The quality factor significantly affects both the compression efficiency and the image quality.







- Lower QF values offer higher compression but result in more noticeable image artifacts.
- Higher QF values preserve the image quality but at the cost of lower compression.

1.9 Good Aspects

- **Effective Compression:** The algorithm effectively reduces the image size, achieving significant compression ratios.
- **Parameter Tuning:** The use of a quality factor allows for adjustable trade-offs between compression efficiency and image quality.
- **Huffman Encoding:** The use of Huffman encoding ensures lossless compression of quantized DCT coefficients.

1.10 Bad Aspects

- **Block Artifacts:** At lower quality factors, the 8x8 block-based compression can introduce visible artifacts in the reconstructed image.
- **Fixed Quantization Matrix:** The use of a standard quantization matrix may not be optimal for all types of images, leading to suboptimal compression performance for certain images.
- **Computational Complexity:** The DCT and Huffman encoding steps are computationally intensive, which might be a limitation for real-time applications.

1.11 Dataset Description

The algorithm was tested on a variety of grayscale images to evaluate its performance under different conditions:

1.11.1 Test Images

We used muliptle images to evaluate image compression algorithms, including **Lena**, a standard test image in image processing, **Cameraman**, frequently used for testing compression, **Barbara**, known for its high-frequency details, **Peppers**, with a mix of textures and colors, and **Mandrill**, which contains complex textures and high-frequency details and an old image which wasnt clear and had noise.

We see how the algorithm malfunctions a little in case of the old image, because of the noise, where the RMSE is increasing with QF. The images, json files and the restores images are in the folder.

Table 1: RMSE Values for Images at Different Quality Factors

Image	QF=1	QF=5	QF=10	QF=25	QF=50
lena.jpg	128.2361	128.3597	128.2867	128.1384	128.1108
mandrill.jpg	129.9680	129.6889	129.1860	128.6782	128.3965
barbara.png	129.2497	129.5092	128.7761	128.3058	128.1595
peppers.png	128.3849	128.8275	128.0515	128.0533	128.0203
old image.png	127.1942	127.8495	128.7615	128.2856	128.0466

2 Problem Statement 2

This project implements the lossy image compression method presented in the paper "Edge-Based Image Compression with Homogeneous Diffusion" by Markus Mainberger and Joachim Weickert. The method focuses on compressing cartoon-like images by utilizing edge information. Edges are detected using the Marr–Hildreth operator followed by hysteresis thresholding, with their locations stored losslessly using JBIG. The grey or color values adjacent to the edges are quantized, subsampled, and compressed using PAQ coding. During decoding, missing image data is reconstructed by solving the Laplace equation through homogeneous diffusion. The approach is expected to outperform JPEG and JPEG2000 standards for cartoon-like images.

2.1 Algorithm

The algorithm provided implements an edge detection and masking process on input images, using a combination of the Canny edge detection algorithm and a neighborhood-based approach to generate a refined mask. The process begins by reading an image and converting it to grayscale. The Canny edge detection algorithm is applied to detect edges in the image. Once edges are detected, the algorithm initializes a mask that will be refined through a series of steps. The mask is modified by iterating over the detected edge pixels and analyzing their neighboring pixels in a defined window. If a pixel's surrounding neighborhood does not contain any previously selected edge pixels, it is marked as part of the final mask. This step ensures that only isolated edge pixels are selected and helps to eliminate redundant edge pixels from the mask.

In the second phase, the algorithm refines the mask by excluding neighboring pixels of selected edges, using a smaller window size to apply the exclusion. This ensures that areas near edges are excluded from the mask, preserving the integrity of the edge-detection process. The final refined mask is then used to create a masked image, where the pixels that lie outside of the detected edges are set to zero. This results in an image that highlights only the regions near the detected edges, which can be useful for applications such as compression or feature extraction. The final mask and the masked image are then displayed, and the mask is saved for future use in binary format.

2.2 Code Files Explaination

2.2.1 Supersampling (supersampler.m)

Objective: This script detects edges in an input image using the Canny edge detector and then generates a final mask that selects certain edge pixels using a neighborhood-

based thresholding process.

Key Operations:

- **Image Reading and Edge Detection:** Reads an image (im1.png) and converts it to grayscale to perform edge detection using the Canny method.
- **Mask Initialization:**

```
msk_im = uint8(zeros(m, n));
mask_im = padarray(msk_im, [d, d]);
final_mask = uint8(zeros(m, n));
```

  Initializes a mask (msk_im) and a padded version (mask_im), followed by a final mask (final_mask) that will store the selected edge pixels.

- **Neighborhood Thresholding:** The script iterates over each pixel and checks if it is an edge pixel, updating the final mask based on its neighbors. If no neighboring edge pixels are selected, it marks that pixel as an edge in the final mask.

Files Generated:

- final_mask.png: A binary mask showing selected edge pixels.

2.3 Subsampling (subsampler.m)

Objective: Similar to supersampler.m, this script performs edge detection using the Canny edge detector, applies a mask to select edge pixels, and further processes the image by excluding nearby pixels around the edges. **Files Generated:**

- final_mask.png: A refined binary mask with selected edge pixels after excluding the neighborhood.
- res_im.png: The image with the mask applied, where edge regions are highlighted.

2.4 Encoder (encoder.m)

Objective: This function takes an image and generates a mask based on detected edges using the Canny method. It then applies the mask to the image to extract the edge regions, saving both the mask and the masked image.

Key Operations:

- **Edge Detection:** Uses the Canny edge detector to find edges in the image.
- **Mask Generation:** A binary mask is created by excluding nearby pixels around detected edges.
- **Save Files:** The mask and the masked image are saved:

Files Generated:

- im2.pbm: Binary mask (mask) based on edge detection.
- res_im2.png: Masked image where edge regions are preserved.

2.4.1 Decoder (decoder.m)

Objective: This function takes the mask and the masked image (with some pixels missing) and estimates the original image using homogeneous diffusion. It then calculates the PSNR value to measure image quality.

Key Operations:

- **Image and Mask Reading:** Reads the masked image and the mask.
- **Homogeneous Diffusion:** The image is iteratively updated using the Laplace operator to diffuse values into the missing pixels based on the mask. The PSNR is calculated at each iteration.
- **Final PSNR Calculation:** After the diffusion process, the PSNR value is calculated to measure the quality of the reconstructed image.

Files Generated:

- restored_im2.png: The estimated original image after homogeneous diffusion.

2.4.2 Resulting Files After Running the Code

- final_mask.png: Generated by supersampler.m and subsampler.m. It marks the edge pixels based on the Canny edge detection and neighborhood thresholding.
- res_im.png: The masked image generated by supersampler.m and subsampler.m, which shows the regions of interest (edge pixels).
- im2.pbm: The mask generated by encoder.m.
- res_im2.png: The image with the mask applied, highlighting edge pixels.
- restored_im2.png: The final output generated by decoder.m, which is the restored image using homogeneous diffusion.

2.4.3 Conclusion

The overall flow involves:

- **Edge Detection:** Using the Canny method to detect edges.
- **Mask Generation:** Creating masks based on edge pixels and excluding neighbors.
- **Compression/Encoding:** The encoder generates a mask and applies it to the image.
- **Restoration:** The decoder uses homogeneous diffusion to restore the original image and measures the quality using PSNR.

Each file has a clear role in the process, with the encoder generating and saving the mask and masked image, and the decoder reconstructing the image from the masked version.

2.5 Results

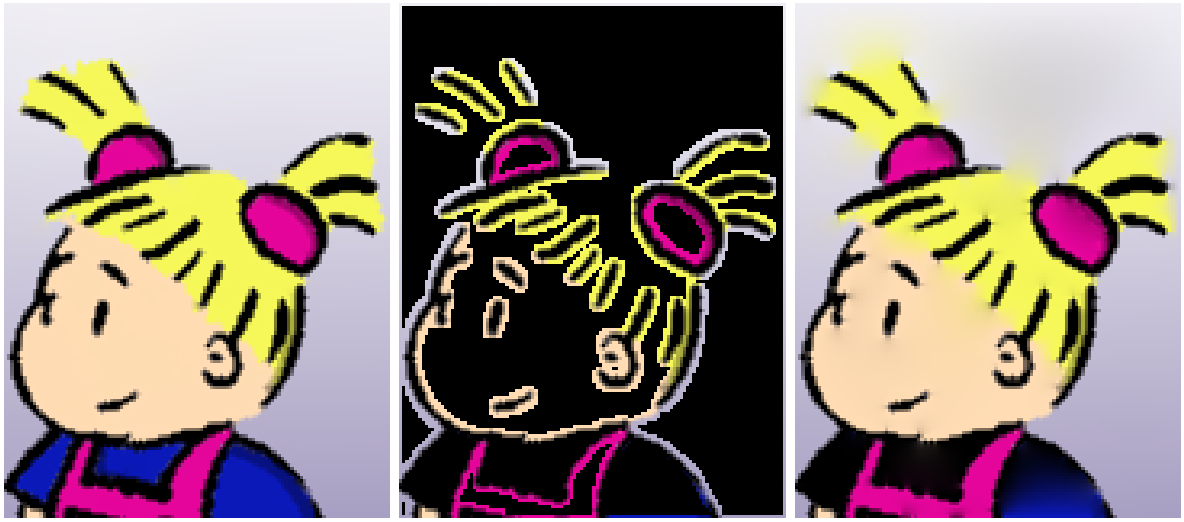


Figure 1: Original Image, image with the mask applied, highlighting edge pixels, and restored Image : For the above image, RMSE is 4.9597004417829735. The image looks pretty well restored except for the shirt part.

Figure 2: Original Image, image with the mask applied, highlighting edge pixels, and restored Image: For the above image, RMSE between the images is 7.434777116673798. We see how the method doesnt capture slow color gradients very well, which is the background.

2.6 Pros and Cons of the Image Compression Method

2.6.1 Pros for Cartoon Images

The image compression method works well for cartoon images due to the following reasons:

- The compression method focuses on preserving these edges, which are the most critical features in cartoons, while simplifying the background regions.

Figure 3: Original Image, image with the mask applied, highlighting edge pixels, and restored Image: For the above image, RMSE between the images: 8.506595296809481. We see how the method doesnt capture slow color gradients very well, which is the background, but has caught the colors of the characters which are monochromatic pretty well.

- The approach of removing or simplifying less important image details (background) and restoring important regions (edges) aligns well with the nature of cartoon images, where edges define the main structure.
- The homogeneous diffusion method, used for restoring missing pixels, works effectively when applied to uniform regions, which are common in cartoon backgrounds.

2.6.2 Cons with Gradient-Based Images

However, the method has some drawbacks when applied to images with gradients:

- Gradients may not be captured effectively in the mask, causing loss of important details in smooth areas of the image.
- The homogeneous diffusion used for restoring missing pixels is better suited for uniform regions and does not handle smooth transitions well, leading to potential blurring or pixelation in gradient areas.
- The focus on edge preservation can result in loss of fine details and artifacts in regions with gradual color changes, which is common in photographs or landscapes.

3 Contributions

Table 2: Contributions of Team Members

Team Member	Contribution
Hardik	Coding, and implementation of the research paper
Prajwal	Testing, correcting codes, and making of the report
Reeyansh	Implementation of JPEG compression and coding