Signals Project report

Presented ToDr. Micheal MelekEng. Sayed Kamel

Ahmed Kamal 9220075

Omar Sayed 9220538

Contents

Introduction

In today's digital era, the proliferation of high-resolution images has led to an unprecedented demand for efficient storage and transmission methods. Image compression techniques play a pivotal role in addressing this challenge by reducing the size of digital images without significantly compromising their visual quality. In this project, we delve into the realm of image compression, focusing on a fundamental technique known as 2D Discrete Cosine Transform (DCT) based compression.

Importance of Image Compression

Images, whether captured by digital cameras, generated by computer graphics, or transmitted over the internet, often consume substantial storage space and bandwidth. Consequently, efficient image compression is crucial for various applications, including:

1. Storage Optimization: By compressing images, we can significantly reduce the amount of disk space required to store vast collections of digital photographs, thereby enabling cost-effective archival and retrieval systems.

2. Bandwidth Conservation: When transmitting images over networks, such as the internet or wireless channels, bandwidth limitations can impede the timely delivery of data. Compressing images before transmission minimizes the amount of data transferred, resulting in faster downloads and reduced network congestion.

3. Real-time Applications: In scenarios where real-time processing is paramount, such as video streaming and medical imaging, compressed images facilitate rapid data transmission and analysis, enabling seamless interaction with digital content.

4. Mobile and Embedded Systems: With the proliferation of mobile devices and embedded systems, resource-constrained environments necessitate lightweight image compression techniques that strike a balance between computational complexity and compression efficiency.

Overview of the Project

In this project, we explore the application of 2D DCT-based image compression, a widely utilized technique renowned for its simplicity and effectiveness. By transforming image data into the frequency domain, 2D DCT enables the concentration of image energy in a reduced set of coefficients, thereby facilitating high compression ratios with minimal perceptual loss. Through a series of experiments, we investigate the impact of retaining varying numbers of DCT coefficients on image quality, quantified using the Peak Signal-to-Noise Ratio (PSNR). Additionally, we analyze the trade-offs between compression ratio and visual fidelity, providing insights into the practical implications of our compression algorithm.

By elucidating the principles of image compression and demonstrating its practical implementation, this project aims to equip readers with a comprehensive understanding of the underlying concepts and methodologies driving modern image compression techniques.

Background

Theory of Image Compression using 2D DCT

Image compression using 2D Discrete Cosine Transform (DCT) is based on the principle of transforming an image from its spatial domain representation into the frequency domain. Unlike spatial domain techniques like Run-Length Encoding or Huffman Coding, which exploit spatial redundancy, DCT focuses on the frequency content of the image. This transformation decomposes the image into its constituent frequencies, enabling efficient representation and compression.

Significance of Retaining Subset of DCT Coefficients

In image compression using 2D DCT, the image is divided into blocks, typically 8x8 pixels, and each block undergoes DCT transformation independently. The resulting DCT coefficients represent the contribution of different frequencies within each block. Notably, the majority of image energy is often concentrated in the lower-frequency coefficients, capturing essential features while discarding high-frequency noise or fine details. By retaining only a subset of these coefficients, particularly those in the top-left corner of the DCT matrix, we achieve high compression ratios while preserving perceptually significant information.

Relevance of PSNR in Image Quality Evaluation

Peak Signal-to-Noise Ratio (PSNR) serves as a standard metric for evaluating the quality of compressed images compared to their originals. PSNR quantifies the ratio between the maximum possible signal strength (peak value) and the distortion introduced during compression (mean square error). Higher PSNR values indicate lower perceptual distortion and thus better image quality. As such, PSNR serves as a crucial tool for assessing the trade-offs between compression efficiency and visual fidelity in image compression algorithms.

Methodology

Steps Involved in Image Compression Algorithm

1. Block Division: Divide the input image into non-overlapping blocks, typically 8x8 pixels in size.

2. 2D DCT Transformation: Apply 2D DCT to each block independently, converting spatial domain information into frequency domain representation.

3. Coefficient Retention: Retain only a subset of DCT coefficients, typically those in the top-left corner, while discarding others to achieve compression.

4. Compression: Store the retained coefficients efficiently, ensuring minimal storage requirements.

5. Decompression: Reconstruct the compressed image by applying inverse 2D DCT to the retained coefficients.

6. PSNR Calculation: Evaluate the quality of the decompressed image using PSNR, comparing it to the original image.

Implementation Details

- 2D DCT: Utilize the Fast Fourier Transform (FFT) algorithm or specialized DCT functions (e.g., scipy.fftpack.dct) to compute the 2D DCT of image blocks.

- Compression: Retain a specified number of DCT coefficients and discard the rest. Store the compressed image efficiently using appropriate data structures.

- Decompression: Reconstruct the image by applying inverse DCT to the retained coefficients, ensuring proper handling of boundary conditions.

- PSNR Calculation: Compute the PSNR value between the original and decompressed images to quantify the quality of compression.

Below is a brief overview of the key steps in the compression algorithm, along with code snippets illustrating the implementation of 2D DCT, compression, decompression, and PSNR calculation.

```
def dct_2d(processed_block):
    """
    Computes the 2D Discrete Cosine Transform (DCT) of a given block.

    Parameters:
        processed_block (numpy.ndarray): Input block (8x8).

    Returns:
        numpy.ndarray: 2D DCT of the input block.
    """
    return dct(dct(processed_block.T, norm='ortho').T, norm='ortho')

def idct_2d(processed_block):
    """
    Computes the inverse 2D Discrete Cosine Transform (IDCT) of a given block.

    Parameters:
        processed_block (numpy.ndarray): Input block (8x8).

    Returns:
        numpy.ndarray: Inverse 2D DCT of the input block.
    """
    return idct(idct(processed_block.T, norm='ortho').T, norm='ortho')

def compress(image, m):
    """
    Compresses an image by retaining only the top-left m x m DCT coefficients of each 8x8 block.

    Parameters:
        image (numpy.ndarray): Input image (height x width x channels).
        m (int): Number of coefficients to retain.

    Returns:
        numpy.ndarray: Compressed image with retained DCT coefficients.
    """
    blocks_shape = (image.shape[0] // 8, image.shape[1] // 8)
    compressed_image = np.zeros((m * blocks_shape[0], m * blocks_shape[1], 3))
    for channel in range(3):
        for row in range(blocks_shape[0]):
            for col in range(blocks_shape[1]):
                processed_block = image[8 * row:8 * (row + 1), 8 * col:8 * (col + 1), channel]
                dct_block = ImageCompressor.dct_2d(processed_block)
                dct_retained_block = dct_block[:m, :m]
                compressed_image[m * row:m * (row + 1), m * col:m * (col + 1), channel] =
dct_retained_block
    return compressed_image

  def decompress(compressed_image, m):
    """
    Decompresses an image by performing inverse DCT and reconstructing blocks.

    Parameters:
        compressed_image (numpy.ndarray): Compressed image with retained DCT coefficients.
        m (int): Number of coefficients retained during compression.

    Returns:
        numpy.ndarray: Decompressed image.
    """
```

```
        blocks_shape = (compressed_image.shape[0] // m, compressed_image.shape[1] // m)
        decompressed_image = np.zeros((8 * blocks_shape[0], 8 * blocks_shape[1], 3))
        for channel in range(3):
            for row in range(blocks_shape[0]):
                for col in range(blocks_shape[1]):
                    processed_block = np.zeros((8, 8))
                    processed_block[:m, :m] = compressed_image[m * row:m * (row + 1), m * col:m * (col + 1), 
channel]
                    idct_block = ImageCompressor.idct_2d(processed_block)
                    decompressed_image[8 * row:8 * (row + 1), 8 * col:8 * (col + 1), channel] = idct_block
        return decompressed_image

    def psnr(input_image, decompressed_image):
        """
        Calculates the Peak Signal-to-Noise Ratio (PSNR) between two images.

        Parameters:
            input_image (numpy.ndarray): Original input image.
            decompressed_image (numpy.ndarray): Decompressed image.

        Returns:
            float: PSNR value.
        """
        SE = (input_image - decompressed_image) ** 2
        MSE = np.mean(SE)
        peak = 255
        PSNR = 10 * log10((peak ** 2) / MSE)
        return PSNR
```

Results

Size Comparisons

- Provide tables or charts comparing the sizes of the original and compressed images for different values of m (compression levels). This demonstrates the effectiveness of the compression algorithm in reducing image size.

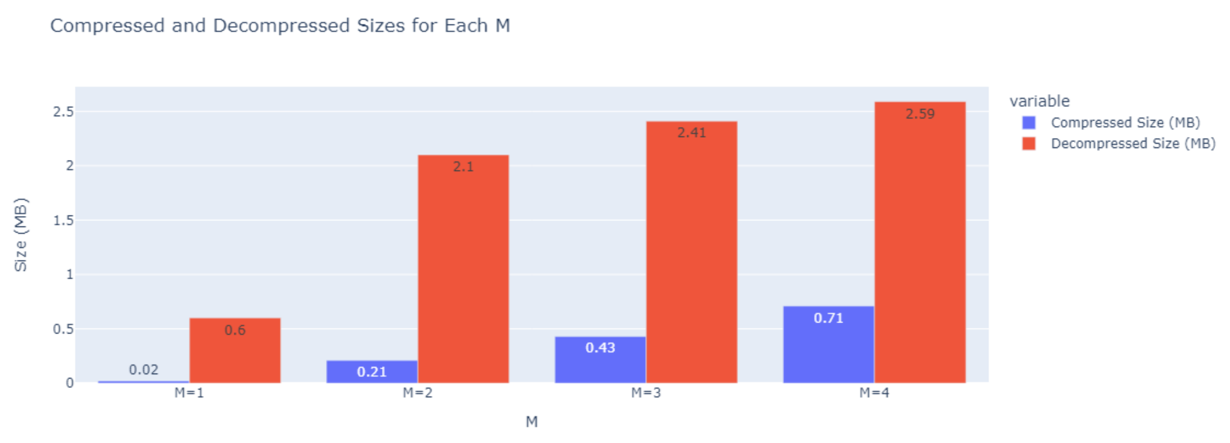

Figure 1 sizes of compressed and decompressed images for different values of m

PSNR vs m Plot

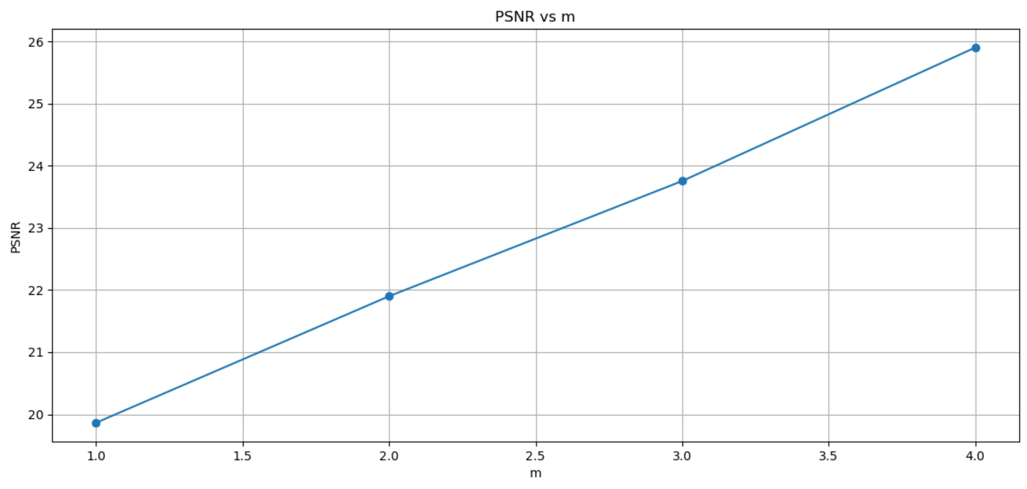

Figure 2 PSNR VS M

By amplifying (m), we tap into a richer spectrum of frequencies, thereby amplifying the quality and, consequently, elevating the PSNR. Hence, there's a direct correlation between (m) and PSNR, where boosting (m) leads to an upscale in PSNR.

Visual Comparison

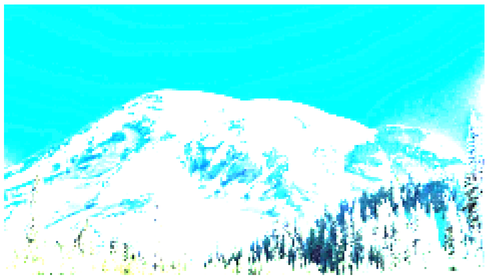

Figure 3 compressed image for m = 1

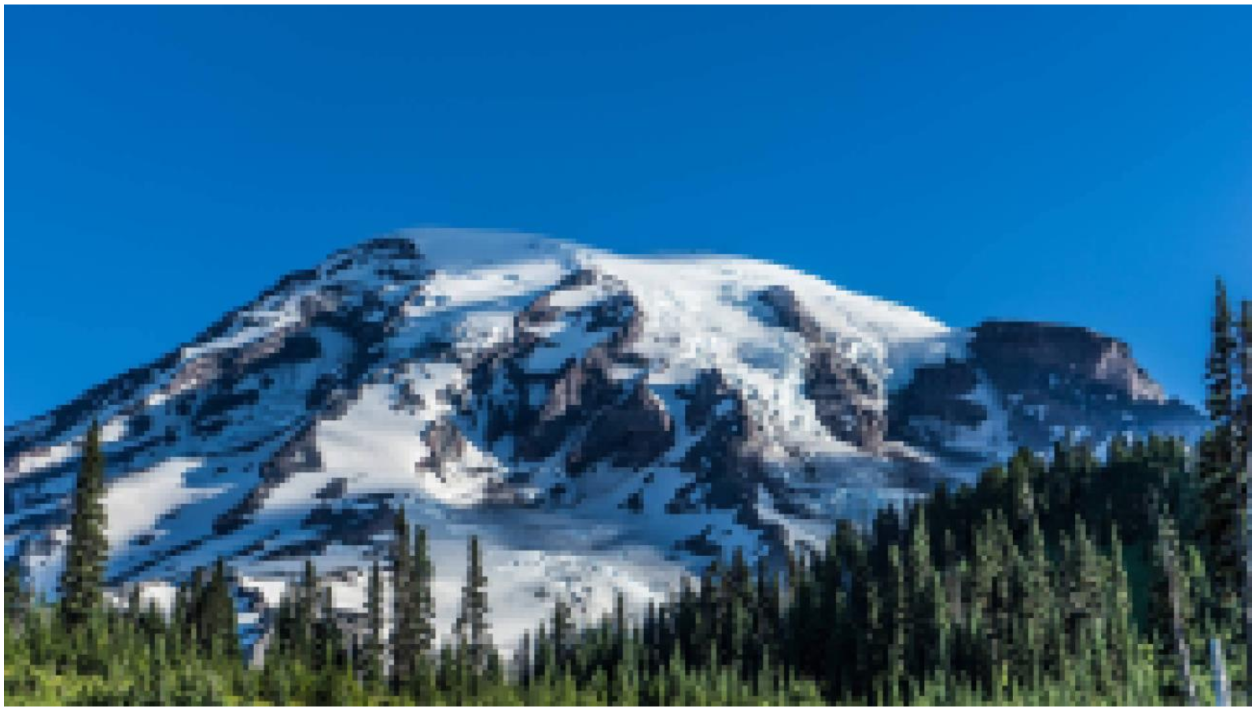

Figure 4 Decompressed image for m = 1

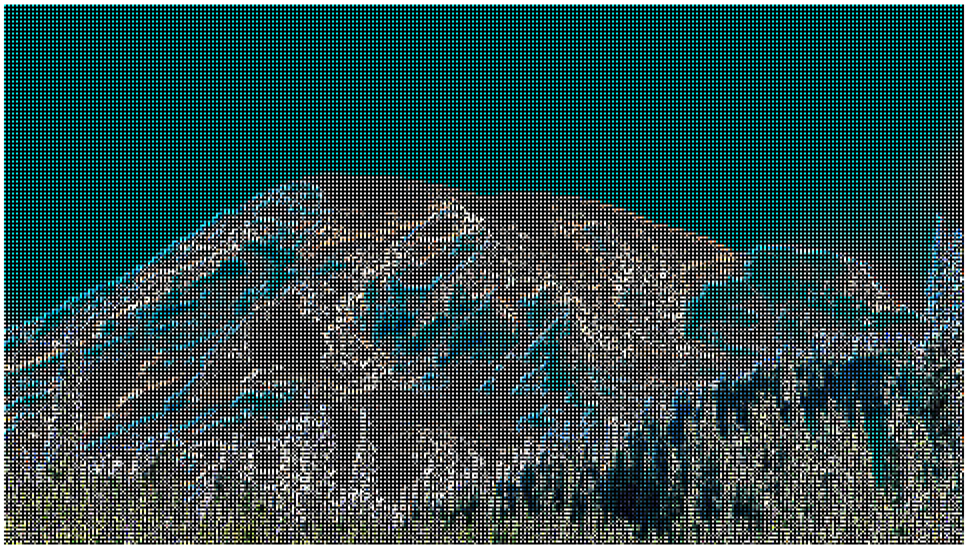

Figure 5 compressed image for m = 2

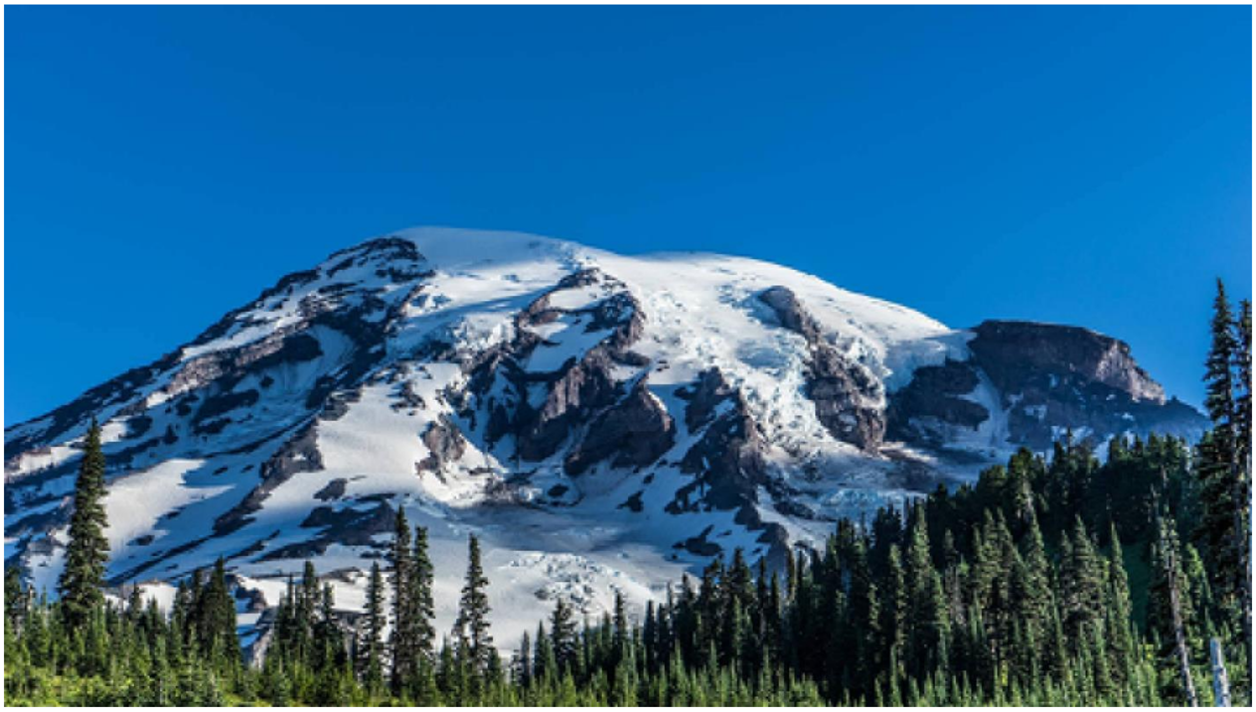

Figure 6 Decompressed image for m = 2

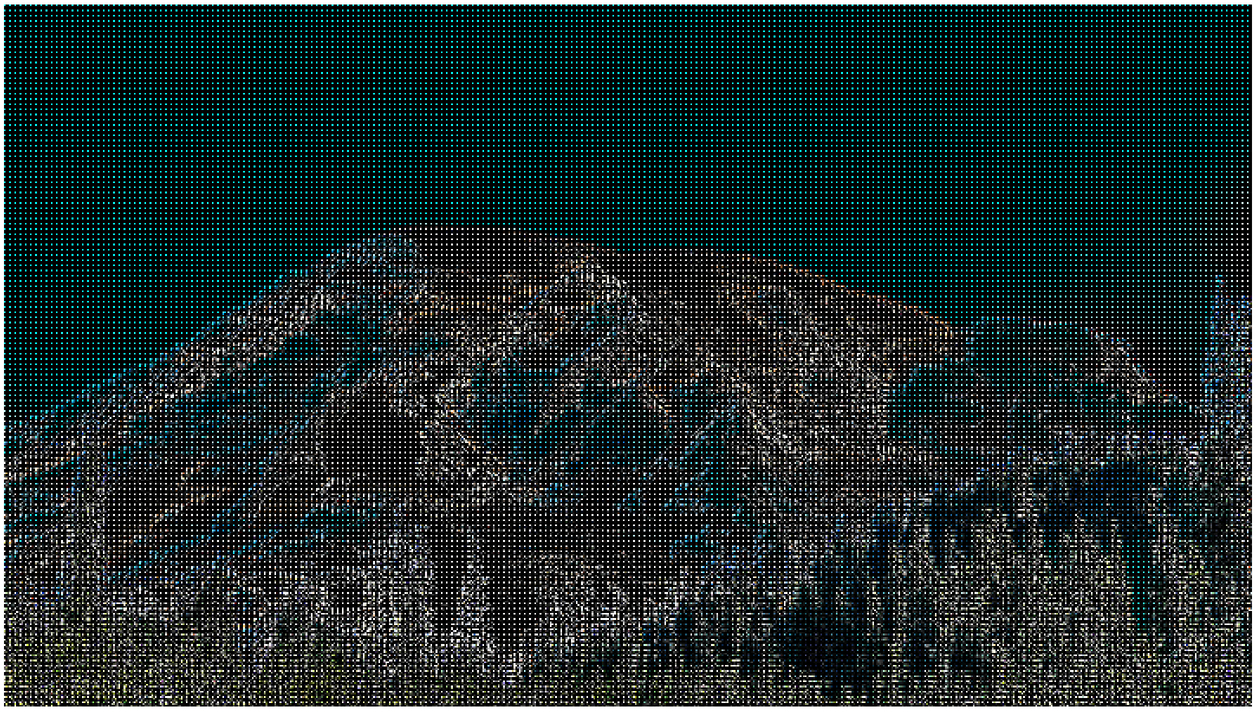

Figure 7 compressed image for m = 3

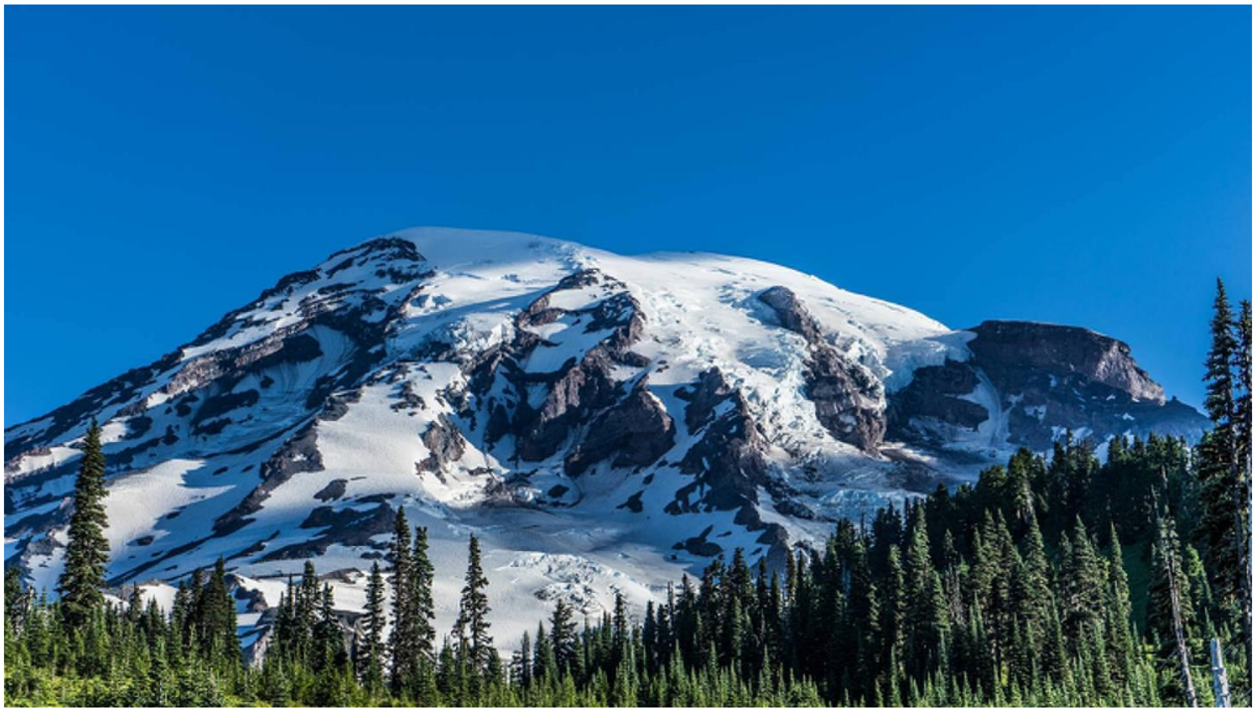

Figure 8 Decompressed image for m = 3

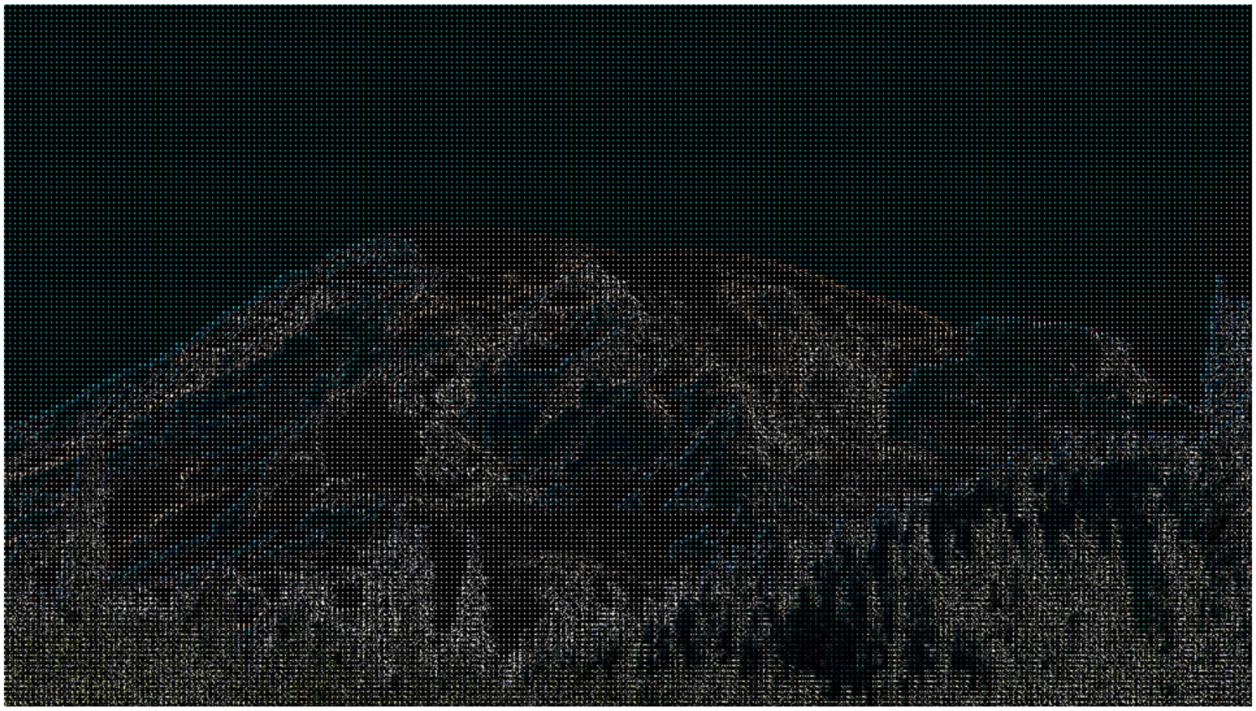

Figure 9 compressed image for m = 4

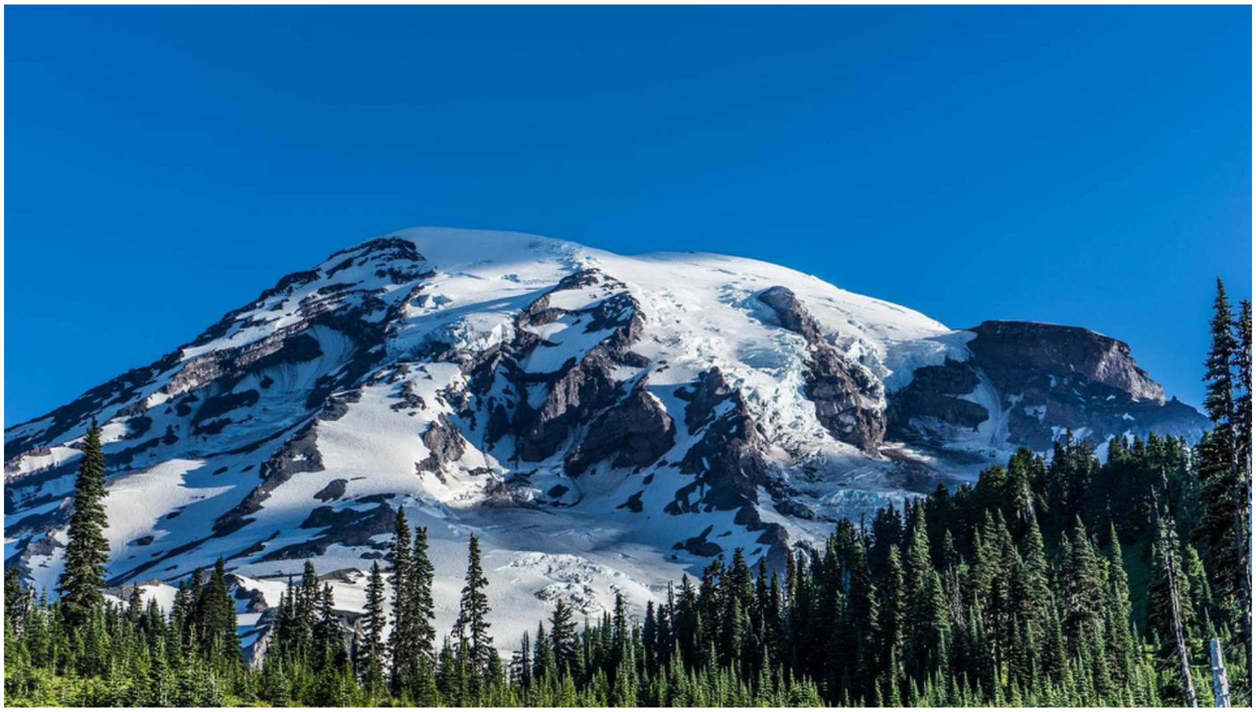

Figure 10 Decompressed image for m = 4

Discussion

Interpretation of Results

The results of the experiments provide valuable insights into the performance of the image compression algorithm. Size comparisons reveal the significant reduction in image size achieved through compression, with higher values of m resulting in larger compression ratios. However, this reduction in size comes at the cost of image quality, as evidenced by the corresponding PSNR values. Visual comparisons further illustrate the trade-off between compression ratio and image fidelity, with higher values of m leading to more noticeable artifacts and loss of detail in the compressed images.

Trade-offs Between Compression Ratio and Image Quality

The trade-offs involved in image compression stem from the inherent tension between achieving high compression ratios and preserving image quality. As the number of retained DCT coefficients (m) decreases, the compression ratio increases, leading to more efficient use of storage space. However, this also results in greater loss of image detail and perceptual fidelity, as evidenced by lower PSNR values and degraded visual quality. Balancing these trade-offs requires careful consideration of the specific requirements and constraints of the application, as higher compression ratios may be acceptable in scenarios where preserving fine details is less critical.

Performance Comparison for Different m Values

Comparing the performance of the compression algorithm for different values of m highlights the nuanced relationship between compression ratio and image quality. Lower values of m yield higher compression ratios but poorer image quality, while higher values of m provide better image fidelity at the expense of reduced compression efficiency. By evaluating the trade-offs between compression ratio and image quality for each m value, it becomes possible to identify optimal compression settings tailored to specific use cases. For instance, applications prioritizing storage efficiency may opt for higher compression ratios, while those emphasizing visual quality may opt for lower compression ratios with minimal loss of detail.

Observations and Insights

Key observations from the experiments include:

- The significant impact of m on both compression ratio and image quality, highlighting the importance of selecting an appropriate value based on the application requirements.

- The diminishing returns associated with increasing m beyond a certain threshold, where further reductions in compression ratio yield minimal improvements in image quality.

- The potential for further optimization or improvement of the compression algorithm through techniques such as adaptive compression, where the value of m is dynamically adjusted based on image content or perceptual criteria.

Conclusion

In conclusion, the study underscores the importance of image compression in various applications and provides valuable insights into the trade-offs involved in achieving optimal compression performance. By balancing compression ratio and image quality, it becomes possible to tailor compression settings to specific requirements, maximizing storage efficiency while minimizing perceptual distortion. Moving forward, continued research and development in image compression algorithms offer promising avenues for enhancing efficiency and quality in practical compression scenarios.

Code

```
import os
import cv2
import sys
import numpy as np
from scipy.fftpack import dct, idct
from math import log10
import matplotlib.pyplot as plt
class ImageCompressor:
    @staticmethod
    def dct_2d(processed_block):
        """
        Computes the 2D Discrete Cosine Transform (DCT) of a given block.

        Parameters:
            processed_block (numpy.ndarray): Input block (8x8).

        Returns:
            numpy.ndarray: 2D DCT of the input block.
        """
        return dct(dct(processed_block.T, norm='ortho').T, norm='ortho')

    @staticmethod
    def idct_2d(processed_block):
        """
        Computes the inverse 2D Discrete Cosine Transform (IDCT) of a given
block.

        Parameters:
            processed_block (numpy.ndarray): Input block (8x8).

        Returns:
            numpy.ndarray: Inverse 2D DCT of the input block.
        """
        return idct(idct(processed_block.T, norm='ortho').T, norm='ortho')

    @staticmethod
    def compress(image, m):
        """
        Compresses an image by retaining only the top-left m x m DCT coefficients
of each 8x8 block.

        Parameters:
            image (numpy.ndarray): Input image (height x width x channels).
            m (int): Number of coefficients to retain.

        Returns:
            numpy.ndarray: Compressed image with retained DCT coefficients.
        """
        blocks_shape = (image.shape[0] // 8, image.shape[1] // 8)
        compressed_image = np.zeros((m * blocks_shape[0], m * blocks_shape[1],
3))
        for channel in range(3):
            for row in range(blocks_shape[0]):
                for col in range(blocks_shape[1]):
                    processed_block = image[8 * row:8 * (row + 1), 8 * col:8 *
(col + 1), channel]
                    dct_block = ImageCompressor.dct_2d(processed_block)
                    dct_retained_block = dct_block[:m, :m]
                    compressed_image[m * row:m * (row + 1), m * col:m * (col +
1), channel] = dct_retained_block
        return compressed_image

    @staticmethod
    def decompress(compressed_image, m):
        """
        Decompresses an image by performing inverse DCT and reconstructing
blocks.

        Parameters:
            compressed_image (numpy.ndarray): Compressed image with retained DCT
coefficients.
            m (int): Number of coefficients retained during compression.

        Returns:
            numpy.ndarray: Decompressed image.
        """
        blocks_shape = (compressed_image.shape[0] // m, compressed_image.shape[1]
// m)
        decompressed_image = np.zeros((8 * blocks_shape[0], 8 * blocks_shape[1],
3))
        for channel in range(3):
            for row in range(blocks_shape[0]):
                for col in range(blocks_shape[1]):
                    processed_block = np.zeros((8, 8))
                    processed_block[:m, :m] = compressed_image[m * row:m * (row +
1), m * col:m * (col + 1), channel]
                    idct_block = ImageCompressor.idct_2d(processed_block)
                    decompressed_image[8 * row:8 * (row + 1), 8 * col:8 * (col +
1), channel] = idct_block
        return decompressed_image

    @staticmethod
    def psnr(input_image, decompressed_image):
        """
        Calculates the Peak Signal-to-Noise Ratio (PSNR) between two images.

        Parameters:
            input_image (numpy.ndarray): Original input image.
            decompressed_image (numpy.ndarray): Decompressed image.

        Returns:
            float: PSNR value.
        """
        SE = (input_image - decompressed_image) ** 2
        MSE = np.mean(SE)
        peak = 255
        PSNR = 10 * log10((peak ** 2) / MSE)
        return PSNR

    @staticmethod
    def visualize_components(input_image):
        """
        Visualizes and saves the color components of an image.

        Parameters:
            input_image (numpy.ndarray): Input image.

        Returns:
            None
        """
        os.makedirs("Image Components", exist_ok=True)
        colors = ["Reds", "Greens", "Blues"]
        for i in range(3):
            plt.figure(figsize=(14, 6))
            plt.axis("on")
            plt.imshow(input_image[:, :, 2 - i], cmap=colors[i]);
            plt.axis("off")
            plt.savefig(f"Image Components/{colors[i][:-1].lower()}
channel.png");

    @staticmethod
    def run(image_path, m):
        """
        Executes the project tasks including compression, decompression, PSNR
calculation, and plotting.

        Parameters:
            image_path (str): Path to the input image.
            m (int): Number of coefficients to retain.

        Returns:
            None
        """
        # Open a file to write results
        with open('sizes.txt', 'w') as file:
            # Read the input image
            input_image_array = cv2.imread(image_path)

            # Visualize the components of the input image
            ImageCompressor.visualize_components(input_image_array)

            # Create directories for storing compressed and decompressed images
if they don't exist
            os.makedirs("Decompressed Images", exist_ok=True)
            os.makedirs("Compressed Images", exist_ok=True)

            # Write the original size of the image to the file
            file.write(f"Original Image Size : {os.path.getsize(image_path) /
(1024 ** 2):.2f} MB\n")

            # Initialize an array to store PSNR values
            PSNRS = np.zeros(m)

            # Compress and decompress the image for different compression levels
            for i in range(1, m + 1):
                # Compress the image
                compressed_image = ImageCompressor.compress(input_image_array, i)

                cv2.imwrite(f'Compressed Images/compressedImageWithM{i}.png',
compressed_image)

                # Decompress the image
                decompressed_image = ImageCompressor.decompress(compressed_image,
i)

                # Write image information and calculate PSNR
                PSNR = ImageCompressor.write_image_info(file,
os.path.getsize(f'Compressed Images/compressedImageWithM{i}.png') / (1024 ** 2),
                                                       input_image_array,
decompressed_image, i)
                # Store PSNR value
                PSNRS[i - 1] = PSNR

                # Save the decompressed image
                cv2.imwrite(f"Decompressed Images/decompressedImageWithM{i}.png",
decompressed_image)


            # Plot PSNR values
            ImageCompressor.plot_PSNRS(m, PSNRS)

    @staticmethod
    def write_image_info(file, size_in_MB, input_image, decompressed_image, m):
        """
        Writes image compression information and PSNR value to a file.

        Parameters:
            file (file): File object to write information to.
            size_in_MB (float): Size of the compressed image in MB.
            input_image (numpy.ndarray): Original input image.
            decompressed_image (numpy.ndarray): Decompressed image.
            m (int): Number of coefficients retained during compression.

        Returns:
            float: PSNR value.
        """
        file.write(f"m = {m} :\n")
        file.write(f"Compressed Image Size: {size_in_MB:.2f} MB\n")
        PSNR = ImageCompressor.psnr(input_image, decompressed_image)
        file.write(f"PSNR: {PSNR:.2f} dB\n")
        file.write("\n")
        return PSNR

    @staticmethod
    def plot_PSNRS(m, PSNRS):
        """
        Plots PSNR values against m values and saves the plot as an image.

        Parameters:
            m (int): Number of coefficients to retain.
            PSNRS (numpy.ndarray): Array of PSNR values.
        Returns:
            None
        """
        all_ms = np.linspace(1, m, m, dtype=np.uint8)
        plt.figure(figsize=(14, 6))
        plt.plot(all_ms, PSNRS, marker='o')
        plt.xlabel("m")
        plt.ylabel("PSNR")
        plt.title("PSNR vs m")
        plt.grid(True)
        plt.savefig("PSNRGraph.png")

ImageCompressor.run("./image1.png", 4)
```