|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Theory & Coding Techniques** | **Semester:** | **V** |
| **Date of Performance:** | **10 / 09 / 2024** | **Batch:** | **B - 1** |
| **Faculty Name:** | **Prof. Makarand Kulkarni** | **Roll Number:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade / Marks:** | **\_\_ / 25** |

**Experiment No.: 4**

**Title:** **JPEG Compression**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| * To implement JPEG image compression using MATLAB/Python. |

|  |
| --- |
| **COs to be achieved:** |
| **CO1**: Learn the basic mathematics and concepts of data compression techniques.  **CO4**: Illustrate the concepts of algorithms and standards for audio, image and video compression. |

|  |
| --- |
| **Theory:** |
| [JPEG](https://www.geeksforgeeks.org/image-formats/) stands for Joint Photographic Experts Group. We perform such type of compression to reduce the size of the file without damaging its quality. By reducing the size we can store it in a huge amount which was not possible earlier. Reducing the size of images will also improve the efficiency of the system as it will give less load on it.  Process Of JPEG Compression:  Firstly, we convert the R, G, B color format to Y, Cb, Cr format. Some colors are more sensitive to human eyes and thus are high-frequency colors. Some colors of chromium compounds like Cb and Cr are less sensitive to human eyes thus can be ignored. Then we reduce the size of pixels in down sampling. We divide our image into 8\*8 pixels and perform forward DCT (Direct Cosine Transformation). Then we perform quantization using quantum tables and we compress our data using various encoding methods like run-length encoding and Huffman encoding.  In the second stage, we decompress our data, it involves decoding where we decode our data, and we again de-quantize our data by referring to the quantization table.  Then we perform Inverse DCT and up sampling to convert it into original pixels and finally, color transformation takes place to convert the image into its original color format. |

|  |
| --- |
| **Stepwise – Procedure:** |
| **Steps of algorithm –**   1. **Splitting** – We split our image into the blocks of 8\*8 blocks. It forms 64 blocks in which each block is referred to as 1 pixel. 2. **Color Space Transform** – In this phase, we convert R, G, B to Y, Cb, Cr model. Here Y is for brightness, Cb is color blueness and Cr stands for Color redness. We transform it into chromium colors as these are less sensitive to human eyes thus can be removed. 3. **Apply DCT** – We apply Direct cosine transform on each block. The discrete cosine transforms. (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. 4. **Quantization** – In the Quantization process, we quantize our data using the quantization table. 5. Serialization – In serialization, we perform the zig-zag scanning pattern to exploit redundancy. 6. **Vectoring** – We apply DPCM (differential pulse code modeling) on DC elements. DC elements are used to define the strength of colors. 7. **Encoding** – In the last stage, we apply to encode either run-length encoding or Huffman encoding. The main aim is to convert the image into text and by applying any encoding we convert it into binary form (0, 1) to compress the data.   **Write a program for JPEG compression.** |

|  |
| --- |
| **Observations:** |
| **JPEG Compression Code:**  import numpy as np  from PIL import Image  import matplotlib.pyplot as plt  import warnings  # Suppress the DecompressionBombWarning  warnings.simplefilter('ignore', Image.DecompressionBombWarning)  # Increase the maximum allowed pixels to prevent the warning  Image.MAX\_IMAGE\_PIXELS = None  # Or set it to a higher value if you want to limit it  # Quantization table for JPEG compression (for simplicity we use a standard table)  quantization\_table = 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]])  # Step 1: Split image into 8x8 blocks  def split\_into\_blocks(image\_matrix):      h, w = image\_matrix.shape      blocks = []      for i in range(0, h, 8):          for j in range(0, w, 8):              block = image\_matrix[i:i+8, j:j+8]              if block.shape == (8, 8):  # Ensuring we only take complete blocks                  blocks.append(block)      return blocks  # Step 2: Convert RGB to YCbCr Color Space  def rgb\_to\_ycbcr(image):      return image.convert('YCbCr')  # Step 3: Apply Discrete Cosine Transform (DCT)  def dct\_2d(block):      max\_value = np.max(block)      max\_value = np.clip(max\_value, a\_min=1, a\_max=None)  # Prevent division by zero      return np.round(np.fft.fft2(block) \* 255 / max\_value)  # Step 4: Quantization of the blocks  def quantize\_block(block, quantization\_table):      return np.round(block / quantization\_table)  # Step 5: Serialization - Zigzag scanning (simplified in our case)  def zigzag\_scan(block):      zigzag\_pattern = []      for i in range(0, 8):          if i % 2 == 0:              zigzag\_pattern.extend(block[i][:i+1])          else:              zigzag\_pattern.extend(np.flip(block[i][:i+1]))      return zigzag\_pattern  # Step 6: Vectoring using DPCM on DC components (simplified in this case)  def dpcm\_encode(blocks):      dc\_elements = [block[0, 0] for block in blocks]      diffs = np.diff(dc\_elements)      return diffs  # Step 7: Run-Length Encoding (RLE) - Simple implementation  def rle\_encode(data):      encoding = []      i = 0      while i < len(data):          count = 1          while i + 1 < len(data) and data[i] == data[i + 1]:              count += 1              i += 1          encoding.append((data[i], count))          i += 1      return encoding  # Main JPEG Compression Flow  def jpeg\_compression(image\_path):      # Load image and convert to grayscale for simplicity      image = Image.open(image\_path)      image\_ycbcr = rgb\_to\_ycbcr(image)        # Convert the image into a numpy array (Y channel only for simplicity)      image\_matrix = np.array(image\_ycbcr)[:, :, 0]  # Only Y channel (luminance)      # Calculate original image size in bytes      original\_size = image.size[0] \* image.size[1]  # Size in pixels, not bytes      # Step 1: Split into 8x8 blocks      blocks = split\_into\_blocks(image\_matrix)        # Step 3: Apply DCT to each block      dct\_blocks = [dct\_2d(block) for block in blocks]        # Step 4: Quantize each block      quantized\_blocks = [quantize\_block(block, quantization\_table) for block in dct\_blocks]        # Step 5: Zigzag Scan each quantized block      zigzag\_scanned = [zigzag\_scan(block) for block in quantized\_blocks]        # Step 6: Vectoring using DPCM on DC components      dpcm\_encoded = dpcm\_encode(quantized\_blocks)        # Step 7: Run-Length Encoding on AC components (skipping DC components)      rle\_encoded = [rle\_encode(block[1:]) for block in zigzag\_scanned]        # Calculate compressed size (simplified)      compressed\_size = sum(len(block) for block in rle\_encoded) + len(dpcm\_encoded)        # Calculate the compression factor      compression\_factor = original\_size / compressed\_size if compressed\_size != 0 else float('inf')        # Print sizes and compression factor      print(f"Original Image Size (in pixels): {original\_size} pixels")      print(f"Compressed Image Size (approx.): {compressed\_size} bytes")      print(f"Compression Factor: {compression\_factor:.2f}")      return rle\_encoded, dpcm\_encoded  # Run the JPEG compression process  image\_path = "img2.jpg"  jpeg\_compression(image\_path)  **Code Output:**    *Image to compress* |

|  |
| --- |
| **Post Lab Subjective Type Questions:** |
|  |

|  |
| --- |
| **Conclusion:** |
| The experiment successfully applied JPEG compression to an image, significantly reducing its size while maintaining acceptable visual quality. The compression factor was calculated, demonstrating the efficiency of the process in minimizing storage requirements. |

**Signature of faculty in-charge with date:**