JPEG IMAGE COMPRESSION

CS663 : Digital Image Processing

Prepared by:

Toshan Achintya Golla(22B2234), Anupam Rawat(22B3982),
Amitesh Shekhar(22B0014)

Date of Submission

24 November 2024

Contents

Chapter 1

Introduction

1.1 Background

Image Compression has always been an important part of Digital Image Processing. Without compression, it is logistically impossible to store and transfer modern day media containers like images and videos. The size of raw videos and images, taken directly from the source, contains lot of redundant data which can be discarded without the loss of any meaningful information. For this project, we have worked on one of the most ubiquitous image compression standards, JPEG(Joint Photographic Experts Group). JPEG can easily achieve BPP (bits per pixel) of 1-2 for colored images without significant reduction in quality. Most webpages use jpeg standard for rendering images efficiently.

1.2 Problem Statement

To build an image compression engine like JPEG and implement it for both grayscale and color images.
Analyse the quality of the compression using RMSE (Root Mean Squared Error) and BPP (Bits Per Pixel) plot.

1.3 Structure of the Report

Our main goal is to implement a simple version of the JPEG algorithm, as was taught in the course. We start with first explaining the main blocks in the JPEG encoder and decoders in section 2.1 and a description of the algorithms used in section 2.2. The code description is given in section 3.1. The results are given in sections 4.1 (dataset used) and 4.2 (resulting images, performance curves etc.). The conclusion and comments are given in section 5.

Chapter 2

Methodology

2.1 Approach

Any general image compression model consists of two distinct functional components: an encoder and a decoder. The encoder performs the compression while the decoder performs the complementary operation of decompression. A block diagram is given below:

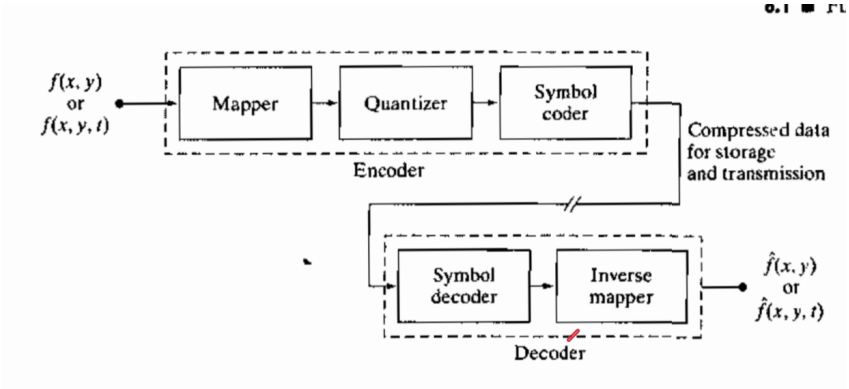

Figure 2.1

In JPEG encoder, we use DCT (Discrete Cosine Transfom) to map the pixel data to its corresponding DCT coefficients. Then a quantizer block quantizes the coefficients (information is lost here, hence JPEG is a lossy compression algorithm). Finally, the quantized coefficients are encoded using run-length and Huffman coding scheme. This data is stored in a suitable format (.bin in our case).
For the decoder block, the same steps are perfomed in the reverse order. (We have followed the scheme followed in class lectures).

A block diagram of JPEG compression model is given below:

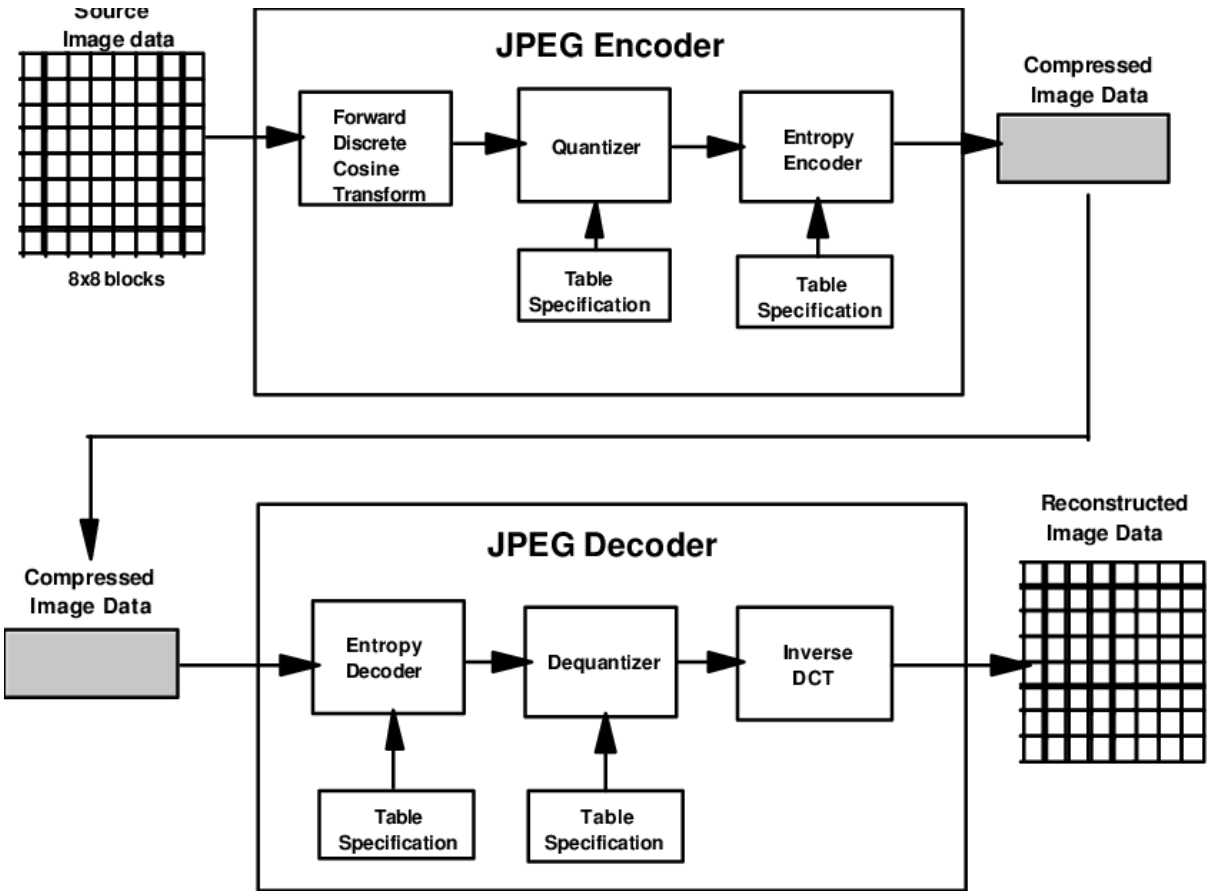

Figure 2.2: JPEG Compression Model (ref : Broko Furht, Research Gate)

2.2 Algorithms Implemented

1. **DCT-2D and IDCT-2D**
   We used the dct-2d() and idct-2d() functions from scipy library to implement discrete cosine transforms on the 8x8 patches. Internally, they use ffts to reduce time complexity.

2. **Huffman Encoding**
   The quantized DCT coefficients were first arranged using **zig-zag** ordering. Now for each 8x8 patch, the Huffman encoding is done using a min-heap implementation.Again, as mentioned in the slides, separate encoding is done for DC and AC coefficients.

3. **Reverse Huffman**
   The encoded coefficients are decoded by traversing the binary tree in reverse direction. The exact implementation is mentioned later.

Chapter 3

Implementation

3.1 JPEG Encoder

1. Divide the image into 8x8 patches

```python
def divide_into_blocks(image):
    h, w = image.shape
    h_blocks = h // 8
    w_blocks = w // 8
    blocks = []
    for i in range(h_blocks):
        for j in range(w_blocks):
            block = image[i*8:(i+1)*8, j*8:(j+1)*8]
            blocks.append(block)
    return blocks
```

Figure 3.1

2. Use inbuilt functions of scipy to calculate dct coeffs of each block
3. Quantize using the Quantization matrix and given Q-value
4. Extract AC-components from each block, store in a list
5. Extract all DC-components from each block, store in a list
6. Perform Huffman encoding on AC-coefficients and get the ac-huffman-table:

```python
# Generate Huffman encoding for the given data.
def huffman_encoding(data):
    # Calculate frequency of each value
    data_non_zero = [x for x in data if x != 0]
    frequency = Counter(data_non_zero)
    heap = [[int(weight), [int(symbol), ""]] for symbol, weight in frequency.items()]
    heapq.heapify(heap)

    # Build the Huffman Tree
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    # Create Huffman dictionary with `int` keys and str values
    huffman_dict = {int(k): str(v) for k, v in dict(heapq.heappop(heap)[1:]).items()}
    return huffman_dict
```

Figure 3.2

7. Similarly perform huffman encoding for DC-coefficients (first item is untouched, rest are encoded using consecutive differences) .Finally get the dc-huffman-table

8. Use run-length encoding to encode the coeffs into triplets and get ac and dc encoded values. Code for AC part given below:

```python
# Encode AC coefficients using run-length and Huffman encoding.
def encode_ac_coefficients(ac_coefficients, ac_huffman_table):
    encoded_ac = []
    run_length = 0
    for coefficient in ac_coefficients:
        coefficient = int(coefficient)  # Ensure Python int
        if coefficient == 0:
            run_length += 1
        else:
            huffman_code = ac_huffman_table[coefficient]  # Python int lookup
            size = len(huffman_code)
            encoded_ac.append((int(run_length), size, str(huffman_code)))
            run_length = 0

    # Ensure we append EOB (End of Block) symbol if necessary
    if run_length > 0:
        encoded_ac.append((0, 0, '0'))  # EOB symbol

    return encoded_ac
```

Figure 3.3

9. Create a header (in the form of dictionary) and store the final encoded blocks, dc-huffman-table, ac-huffman-table, Quality Factor and image shape in a .bin file.

```python
# Save the JPEG-encoded data to a binary file
def save_jpeg_encoded_file(encoded_blocks, dc_huffman_table, ac_huffman_table, Quality_factor, image_shape, color_mode, output_file):
    # Prepare the data
    jpeg_data = {
        "image_shape": image_shape,
        "color_mode": color_mode,
        "quality_factor": Quality_factor,
        "dc_huffman_table"
        "ac_huffman_table"  (parameter) encoded_blocks: Any
        "encoded_blocks": encoded_blocks
    }

    # Serialize and save
    with open(output_file, "wb") as f:
        pickle.dump(jpeg_data, f, protocol=pickle.HIGHEST_PROTOCOL)
```

Figure 3.4

3.2 JPEG Decoder

The same steps are perfomed, albeit in a reversed order. The main steps and important code snippets are given below:

1. Read the compressed bin file
2. Extract the parameters of the image like image shape, quality factor, ac and dc huffman tables as well as the main encoded coefficients from the bin file.

```python
# Read the JPEG-encoded data from a binary file
def read_jpeg_encoded_file(input_file):
    with open(input_file, "rb") as f:
        jpeg_data = pickle.load(f)
    return (jpeg_data["image_shape"],
            jpeg_data["quality_factor"],
            jpeg_data["dc_huffman_table"],
            jpeg_data["ac_huffman_table"],
            jpeg_data["encoded_blocks"])
```

Figure 3.5

3. Decode the AC and DC coefficients using ac and dc huffman table and run-length decoding (as mentioned in Gonzalez and Woods)

```python
def decode_ac_coefficients(encoded_ac, ac_huffman_table): # Decode the AC coefficients
    # Reverse the AC Huffman table for decoding
    reverse_ac_huffman_table = {v: k for k, v in ac_huffman_table.items()}

    # Initialize the AC coefficients array
    ac_coefficients = []

    for run_length, size, huffman_code in encoded_ac:
        if run_length == 0 and size == 0:
            # End of Block (EOB)
            break

        # Decode the coefficient value
        coefficient = reverse_ac_huffman_table[huffman_code]

        # Add zeros for the run length
        ac_coefficients.extend([0] * run_length)

        # Add the actual coefficient
        ac_coefficients.append(coefficient)

    # Pad with zeros to make 63 coefficients
    while len(ac_coefficients) < 63:
        ac_coefficients.append(0)

    return ac_coefficients
```

Figure 3.6

```python
def decode_dc_coefficients(encoded_dc, dc_huffman_table): # Decode the DC coefficients
    # Reverse the DC Huffman table for decoding
    reverse_dc_huffman_table = {v: k for k, v in dc_huffman_table.items()}

    # Decode DC differences
    dc_differences = [encoded_dc[0]]  # The first DC coefficient is explicitly stored
    for code in encoded_dc[1:]:
        diff = int(reverse_dc_huffman_table[code])
        dc_differences.append(diff)

    # Reconstruct absolute DC coefficients
    dc_coefficients = [dc_differences[0]]
    for i in range(1, len(dc_differences)):
        dc_coefficients.append(dc_coefficients[i-1] + dc_differences[i])

    return dc_coefficients
```

Figure 3.7

4. Decode each block using the previously obtained coefficients and reverse zig-zag
5. Dequantize the coefficients (still error would be there due to rounding off during encoding)

6. Inverse DCT

7. Stitch the patches together to create the final decompressed image

```python
def combine_patches(image_patches, image_shape):
    h, w = image_shape
    h_blocks = h // 8
    w_blocks = w // 8

    # Initialize an empty array for the reconstructed image
    reconstructed_image = np.zeros((h, w), dtype=np.uint8)

    # Place each patch into its correct position
    for i in range(h_blocks):
        for j in range(w_blocks):
            patch_idx = i * w_blocks + j
            reconstructed_image[i*8:(i+1)*8, j*8:(j+1)*8] = image_patches[patch_idx]

    return reconstructed_image
```

Figure 3.8

3.3 JPEG for colored images

This **color JPEG compression** implementation is almost similar, with the following key differences from grayscale JPEG compression:

1. **Color Space Conversion**: The input color image (in BGR format) is converted to the **YCbCr color space**, separating luminance (Y) and chrominance (Cb, Cr) components (using an inbuilt function of opencv $.COLOR_BGR2YCrCb$) which internally uses similar formula as the one mentioned in the slides :

$$\begin{aligned} Y &= 16 + (65.481R + 128.553G + 24.966B) \\ C_B &= 128 + (-37.79R - 74.203G + 112B) \\ C_R &= 128 + (112R - 93.786G - 18.214B) \end{aligned}$$

2. **Chroma Subsampling**: The chrominance channels (Cb and Cr) are **downsampled** (halved) to reduce data size while retaining visual quality. This step is not required for grayscale images.

3. **Separate Compression**: Each channel (Y, downsampled Cb, and downsampled Cr) undergoes compression using a **grayscale JPEG compression** method independently.

4. **Upsampling**: The chrominance channels are **upsampled** back to their original dimensions after processing to reconstruct the image.

5. **Final Reconstruction**: The compressed YCbCr image is merged and converted back to the BGR format for saving as the final compressed image.

By handling chrominance data separately and using subsampling, color JPEG achieves greater **compression efficiency** compared to grayscale JPEG, while maintaining visual fidelity.

Chapter 4

Results and Analysis

4.1 JPEG on grayscale images

Original Image: (note that we have first converted it to grayscale using this and then used it for our jpeg compressor since the inbuilt function of opencv for grayscale conversion was introducting some white dots in the compressed images.)

Figure 4.1: Original Image, Size = 42KB, 637x637 image (credits : Professor Ajit Rajwade, IIT Bombay)

JPEG Compression for increasing Q (quality factor):

(a) Q = 10, 19KB (b) Q = 20, 22KB (c) Q = 30, 26KB

(d) Q = 40, 29KB (e) Q = 50, 32KB (f) Q = 60, 36KB

(g) Q = 70, 37KB (h) Q = 80, 40KB (i) Q = 90, 41KB

(j) Q = 100, 41KB (k) Q = 300, 51KB (l) Q = 500, 51KB

Figure 4.2: Compressed Images for increasing Q (10 to 100 in steps of 10 from top to bottom, left to right)

4.1.1 Dataset Used for the RMSE vs BPP plot

We have used the Kodak Lossless True Color Image Suite dataset. They are lossless, true color (24 bits per pixel, aka "full color") images in png format (since it a lossless standard). Each image is 768x512 or 512x768 in size and we have used 24 images from this suite.

4.1.2 RMSE vs BPP plot

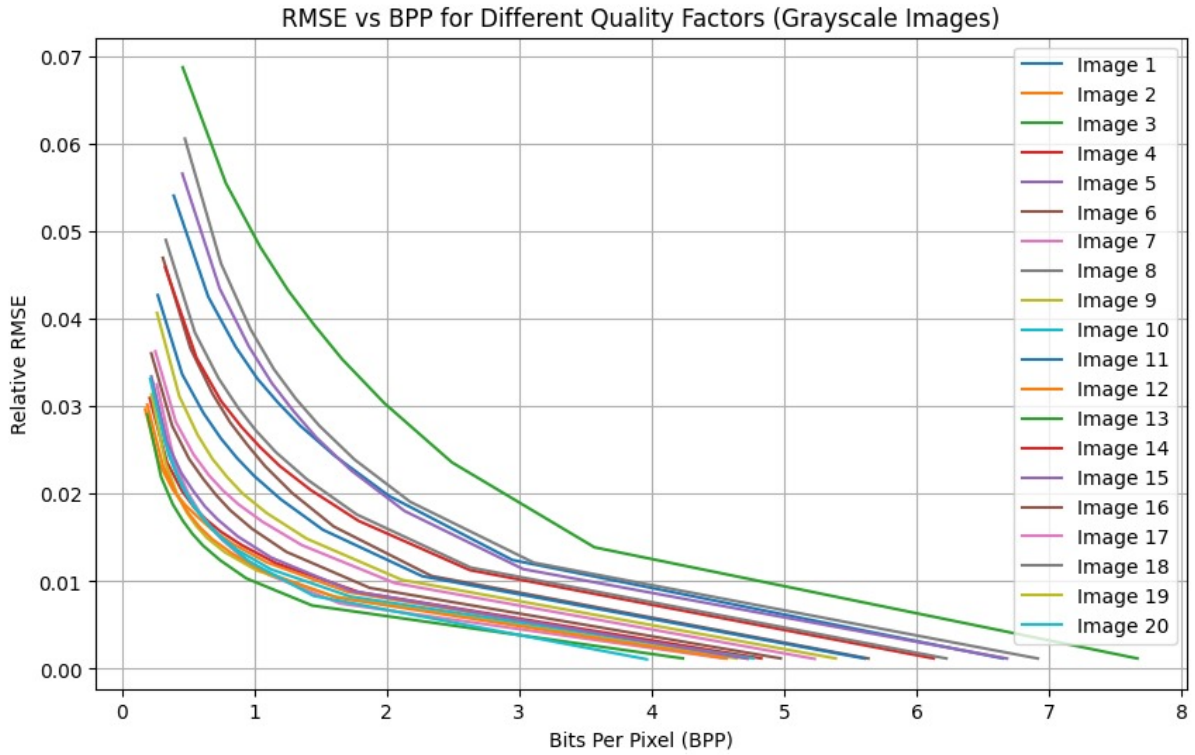

Figure 4.3: RMSE vs BPP plot for Grayscale images from the kodak dataset

4.2 JPEG on color images

Original Image:

Figure 4.4: Original Image, Size = 623KB, 512x768 image

JPEG Compression for increasing Q (quality factor):

(a) Q = 10, 38KB (b) Q = 30, 72KB (c) Q = 50, 90KB

(d) Q = 70, 102KB (e) Q = 90, 109KB (f) Q = 110, 118KB

Figure 4.5: Compressed Images for increasing Q (10 to 110 in steps of 20 from top to bottom, left to right)

4.2.1 Dataset Used for the RMSE vs BPP plot

We have used the same Kodak Lossless True Color Image Suite dataset. They are lossless, true color (24 bits per pixel, aka "full color") images in png format (since it a lossless standard). Each image is 768x512 or 512x768 in size and we have used 24 images from this suite.

4.2.2 RMSE vs BPP plot

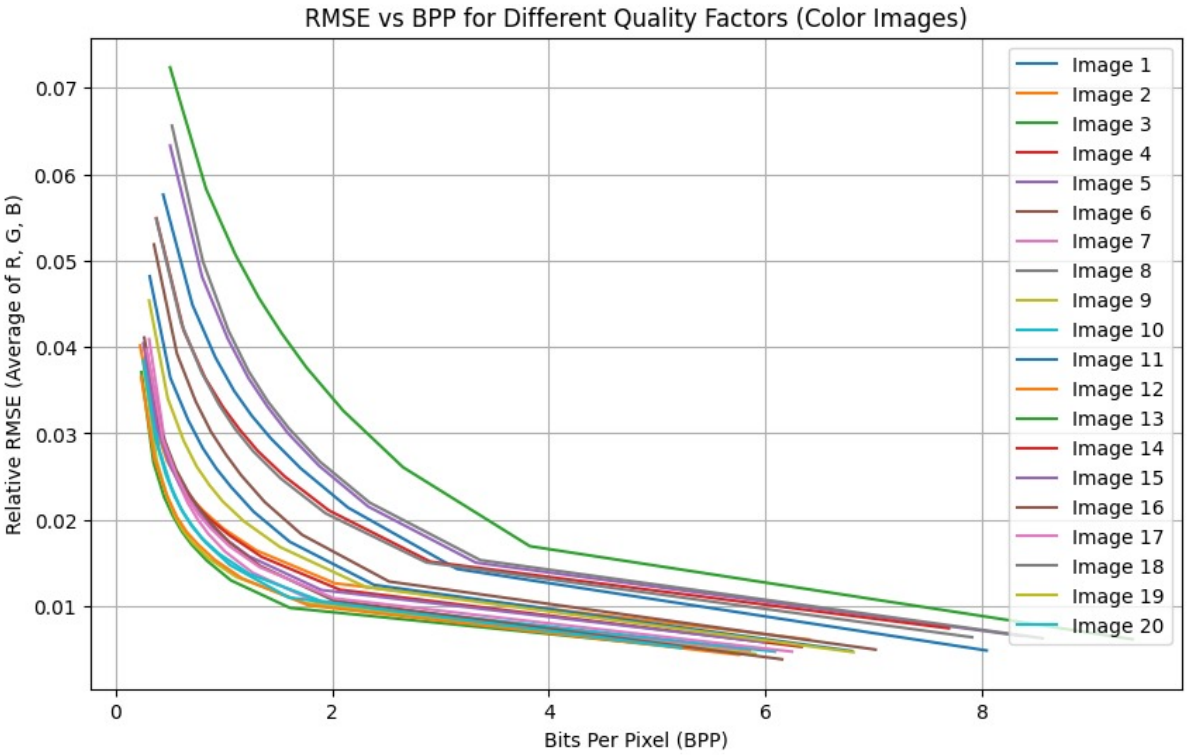

Figure 4.6: RMSE vs BPP plot for Color images from the kodak dataset

Chapter 5

Conclusion

We successfully implemented the JPEG algorithm, achieving 0.65 BPP for grayscale image at Q=50 and 1.8 BPP for color image at Q=50. This is a tremendous level of compression without much loss in information. Some of the good and bad aspects of our implementation are as follows:

Good Aspects

1. Implementation in python as opposed to cpp or matlab. Makes the code easier to follow and understand. Matrix manipulation is taken care by numpy. Also faster due to parallel computation (not relevant for our tiny workload though).

2. Using heaps to implement Huffman encoding. Since heaps also store the data as a min-max binary tree, it is exactly similar to the huffman tree. Implemeting heap is also made easier by python's inbuilt library and its compatibility with multi-dimensional lists.

3. Code is modular, script can be directly imported as a module

4. Made a interactive GUI using tkinter library, makes the demo more presentable and elegant.

5. Ringing and seam artifacts are minimized at Q above 30 (according to our limited testing)

Bad Aspects

1. We are cropping the image to make the height and width multiples of 8. This leads to loss of information (though unavoidable in our model). Padding could have been a better approach.

2. Upon converting RGB to grayscale images using the inbuilt function of opencv (or averaging the R,G, B values) was leading to small scattered dots in the output image (But doing so using online converter tools avoided this issue). There may exist a better way to convert from RGB colorspace to grayscale which prevents this.

3. Implementing the code in good old C would have lead to a more efficient and faster code.

Chapter 6

Contributions

1. Toshan Achintya Golla (22b2234) : Wrote the code for encoder, saving the compressed file in bin format (earlier decided on JSON) and decoder, implemented color compression

2. Anupam Rawat (22b3982) : Reconstruction and Combination of the image patches, Made the interactive GUI, result analysis

3. Amitesh Shekhar (22b0014) : Huffman encoding-decoding and run-length encoding, Report, Read the research paper : *Using partial differential equations (PDEs) for image compression: M. Mainberger and J.Weickert, "Edge-Based Image Compression with Homogeneous Diffusion", CAIP 2009*