ELE 474 PROJECT REPORT

Batuhan Öztürken
Electrical & Electronics Engineering
191201024

I. Introduction

JPEG, short for Joint Photographic Experts Group, is a popular method for compressing digital images, particularly in digital photography. It offers adjustable compression levels, balancing between storage size and image quality. Since its introduction in 1992, JPEG has become the most widely used image format globally, essential for digital imaging and prevalent on the internet. The format, which typically uses "jpg" or "jpeg" file extensions, includes JPEG/Exif and JPEG/JFIF variants, commonly used in digital cameras and online image sharing. Despite the introduction of JPEG 2000 as a potential successor, the original JPEG remains the dominant standard in digital image compression.

The aim of the project is to apply JPEG compression steps to a bitmap extension image given to us, this image is 512x512, and then to obtain the success parameters by retrieving the image.

The operations mentioned are block splitting, DCT, quantization, zig-zag encoding and Huffman encoding. Then, all of the these processes are performed in reverse and the image is obtained back with some loss.

II. Encode Stages

A. Block Splitting

The 8x8 block splitting in JPEG compression is a crucial step in its compression algorithm for several reasons. JPEG compression uses the Discrete Cosine Transform (DCT) to convert the spatial domain of an image (i.e., pixel values) into the frequency domain (i.e., how often certain tones appear). The DCT is most effective on small blocks of data. By dividing the image into 8x8 pixel blocks, the JPEG algorithm can apply the DCT more efficiently. After applying the DCT, each 8x8 block undergoes a quantization process, where less visually important information (typically high-frequency components) is reduced. This process significantly reduces the file size. The 8x8 size is large enough to capture important visual details but small enough to allow for significant data reduction.

The 8x8 size represents a balance between computational efficiency and compression effectiveness. Larger blocks would capture more data but would increase computational complexity and potentially decrease the effectiveness of compression.

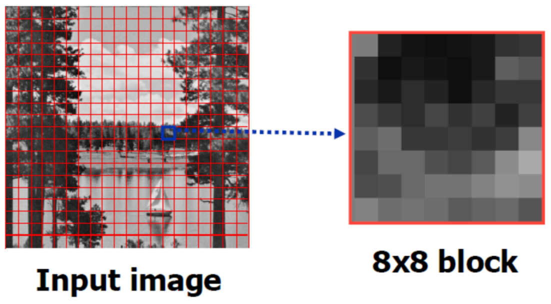
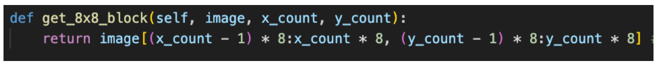


Fig. 1. Block Splitting Figure



```python
def get_8x8_block(self, image, x_count, y_count):
    return image[(x_count - 1) * 8:x_count * 8, (y_count - 1) * 8:y_count * 8]
```

Code 1 : Block Splitting Code

B. Discrete Cosine Transform

Each 8×8 block of each component (Y, Cb, Cr) is converted to a frequency-domain representation, using a normalized, two-dimensional type-II discrete cosine transform (DCT). In JPEG compression, before applying the Discrete Cosine Transform (DCT), pixel values are shifted from the range [0, 255] to [-128, 127] (or a similar range like [-127, 128]). This centering process shifts the mean of the pixel values to zero, which is a prerequisite for the DCT to work effectively. The DCT is designed to work with data that oscillates around zero. By shifting the pixel values to a range centered around zero, the DCT can more effectively separate the image into its frequency components. This process helps in isolating the low-frequency (major visual components) from the high-frequency (fine details and noise) elements of the image.

Then, two-dimensional (2-D) DCT calculation is performed. The formula is given below.

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

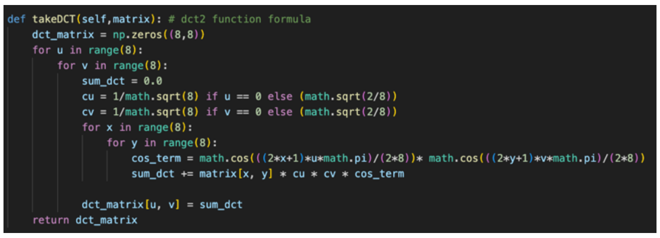
Fig. 2. DCT Formula

Link for the meanings of the terms : DCT Terms

There is a numerically large element in the upper left corner of the matrix. This is the DC coefficient (also called the constant component) that defines the base color of the entire block. The remaining 63 coefficients are AC coefficients (also called alternating components). The advantage of DCT is that it tends to concentrate most of the signal in one corner of the result, as can be seen below. The quantization step to follow highlights this effect while also reducing the overall size of the DCT coefficients, resulting in a signal that is easy to compress efficiently in the entropy stage.

$$G = \overset{\displaystyle \overset{u}{\longrightarrow}}{\begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix}} \Big\downarrow v.$$

Fig. 3. Sample from DCT Result



```
def takeDCT(self,matrix): # dct2 function formula
    dct_matrix = np.zeros((8,8))
    for u in range(8):
        for v in range(8):
            sum_dct = 0.0
            cu = 1/math.sqrt(8) if u == 0 else (math.sqrt(2/8))
            cv = 1/math.sqrt(8) if v == 0 else (math.sqrt(2/8))
            for x in range(8):
                for y in range(8):
                    cos_term = math.cos(((2*x+1)*u*math.pi)/(2*8))* math.cos(((2*y+1)*v*math.pi)/(2*8))
                    sum_dct += matrix[x, y] * cu * cv * cos_term

            dct_matrix[u, v] = sum_dct
    return dct_matrix
```

Code 2: DCT Code

C. *Quantization*

Quantization allows one to greatly reduce the amount of information in the high frequency components. This is done by simply dividing each component in the frequency domain by a constant for that component, and then rounding to the nearest integer. As a result of this, it is typically the case that many of the higher frequency components are rounded to zero, and many of the rest become small positive or negative numbers, which take many fewer bits to represent.

The quantization matrix to be used for our project is given below.

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

Fig. 4. Quantization Matrix

Dividing the G matrix shown in Figure 3 into the Quantization matrix shown in Figure 4 and rounding it to the nearest integer, a matrix as follows is obtained.

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Fig. 5. Matrix G Divided by Q's Resultant

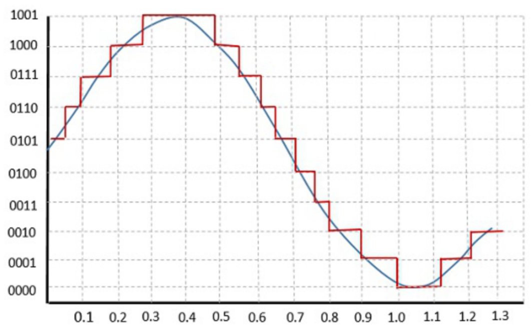
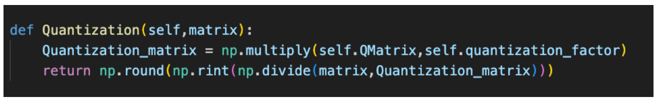


Fig. 6. Quantization Example



```
def Quantization(self,matrix):
    Quantization_matrix = np.multiply(self.QMatrix,self.quantization_factor)
    return np.round(np.rint(np.divide(matrix,Quantization_matrix)))
```

Code 3: Quantization Code (Quantization factor = 1)

D. *Entropy Coding - ZigZag Coding*

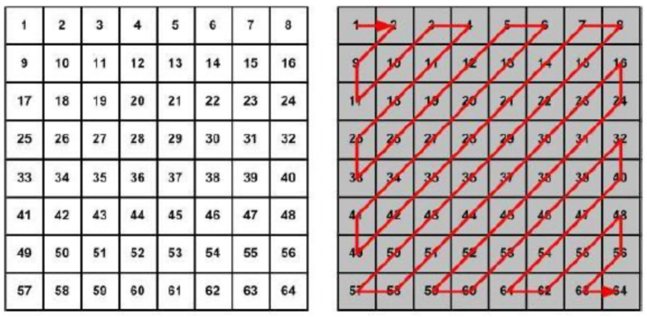
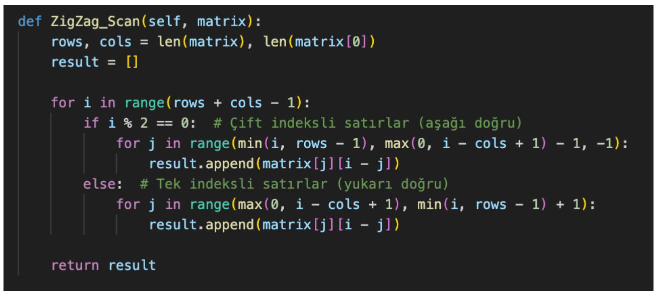


Fig. 7. Zig-Zag Scan

Zig-Zag Scan moves along the path shown in Figure 7, transforming the 8x8 matrix into a 1x64 element vector with the row of elements in the direction of the arrow. Entropy coding is a special form of lossless data compression. It involves arranging the image components in a "zigzag" order employing run-length encoding (RLE) algorithm that groups similar frequencies together, inserting length coding zeros, and then using Huffman coding on what is left. The DCT coefficients in the upper-left corner of the matrix represent these lower frequency components. Zigzag ordering starts from this corner, ensuring that these visually significant components are handled first.



```python
def ZigZag_Scan(self, matrix):
    rows, cols = len(matrix), len(matrix[0])
    result = []

    for i in range(rows + cols - 1):
        if i % 2 == 0:  # Çift indeksli satırlar (aşağı doğru)
            for j in range(min(i, rows - 1), max(0, i - cols + 1) - 1, -1):
                result.append(matrix[j][i - j])
        else:  # Tek indeksli satırlar (yukarı doğru)
            for j in range(max(0, i - cols + 1), min(i, rows - 1) + 1):
                result.append(matrix[j][i - j])

    return result
```

Code 3: Zig-Zag Scan

An example Zigzag Scan output provided by the code is:

```
[15.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

E. *Huffman Encode*

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Huffman Coding steps:

- Any array be as follows:

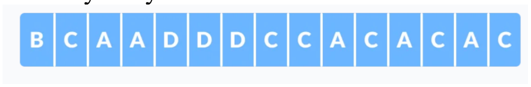


Fig. 8. Initial String

Using the Huffman Coding technique, we can compress the string to a smaller size. Huffman coding first creates a tree using the frequencies of the character and then generates code for each character. Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

- Calculate the frequency of each character in the string.

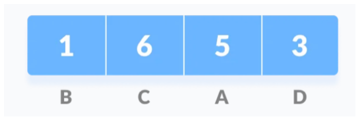


Fig. 9. Frequency of String

- Sort the characters in increasing order of the frequency. These are stored in a priority queue and Make each unique character as a leaf node.

- Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

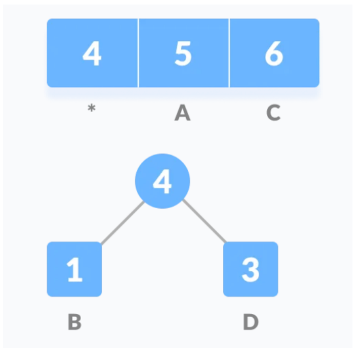


Fig. 10. Getting the Sum of the Least Numbers

- Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above) and insert node z into the tree.

- Repeat

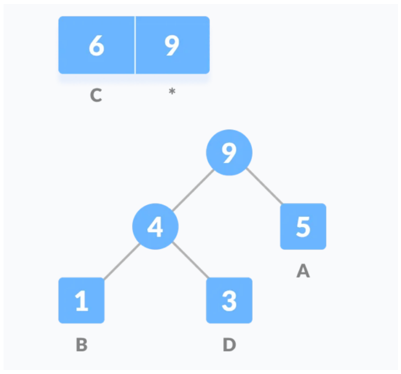


Fig. 11. Repeat Steps

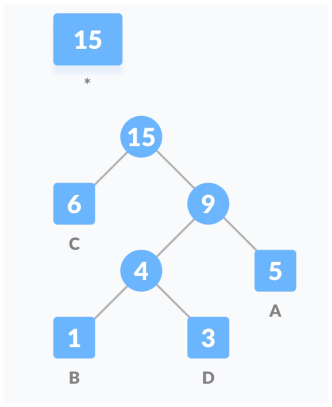


Fig. 12. Repeat Steps

- For each non-leaf node, assign 0 to the left edge and 1 to the right edge

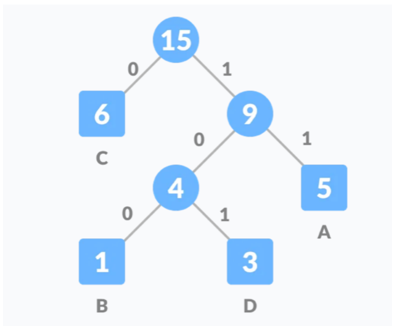
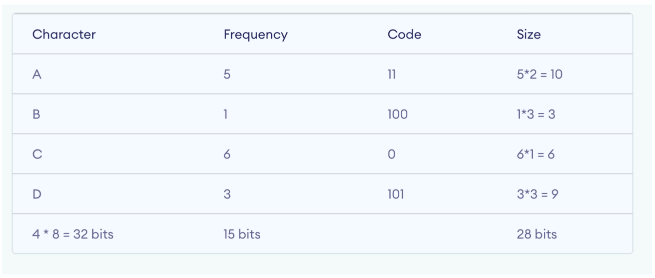


Fig. 13. Assign the Binary Numbers

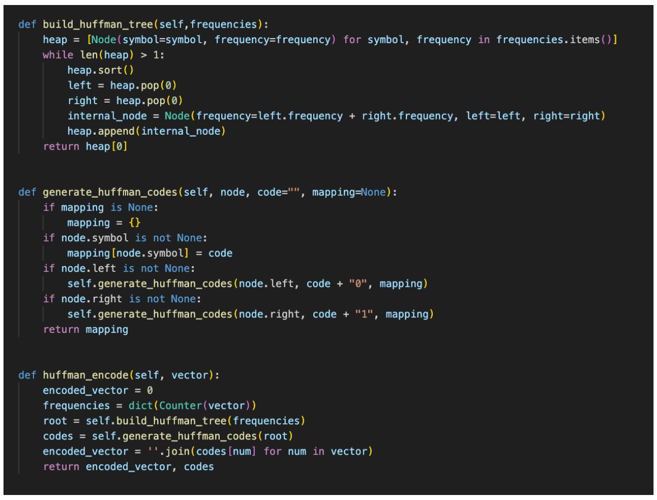
For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.



Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

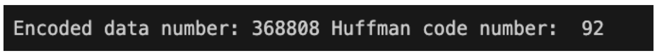
Table 1: Total Table

Huffman codes are optimal prefix codes, meaning no code is a prefix of any other code. This property ensures that each code is uniquely decodable, preventing ambiguity in the decompression process. Huffman has played a significant role in the storage and transmission of digital media, making it more efficient and accessible.



```python
def build_huffman_tree(self,frequencies):
    heap = [Node(symbol=symbol, frequency=frequency) for symbol, frequency in frequencies.items()]
    while len(heap) > 1:
        heap.sort()
        left = heap.pop(0)
        right = heap.pop(0)
        internal_node = Node(frequency=left.frequency + right.frequency, left=left, right=right)
        heap.append(internal_node)
    return heap[0]


def generate_huffman_codes(self, node, code="", mapping=None):
    if mapping is None:
        mapping = {}
    if node.symbol is not None:
        mapping[node.symbol] = code
    if node.left is not None:
        self.generate_huffman_codes(node.left, code + "0", mapping)
    if node.right is not None:
        self.generate_huffman_codes(node.right, code + "1", mapping)
    return mapping


def huffman_encode(self, vector):
    encoded_vector = 0
    frequencies = dict(Counter(vector))
    root = self.build_huffman_tree(frequencies)
    codes = self.generate_huffman_codes(root)
    encoded_vector = ''.join(codes[num] for num in vector)
    return encoded_vector, codes
```

Code 4: Huffman Encode code, the code contains Huffman trees and Huffman codes.



```
Encoded data number: 368808 Huffman code number:  92
```

Fig. 14. Encoded Data and Huffman Code Number for First Part.

III. Decode Stages

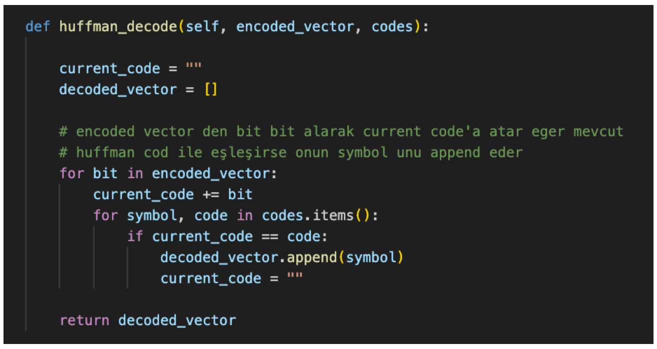
Decoding to display the image consists of doing all the second header in reverse. Start the decoding process by decoding the Huffman codes and trees that we have successfully completed the jpeg compression process.

A. Huffman Decode

For decoding the code, we can take the code and traverse through the tree to find the character. The results will give us the Zig-Zag Scan output again.

From our previous example;

- Start from the root and do the following until a leaf is found.
- If the current bit is 0, we move to the left node of the tree.
- If the bit is 1, we move to right node of the tree.
- If during the traversal, we encounter a leaf node, we print the character of that particular leaf node and then again continue the iteration of the encoded data starting from step 1.



```python
def huffman_decode(self, encoded_vector, codes):

    current_code = ""
    decoded_vector = []

    # encoded vector den bit bit alarak current code'a atar eger mevcut
    # huffman cod ile eşleşirse onun symbol unu append eder
    for bit in encoded_vector:
        current_code += bit
        for symbol, code in codes.items():
            if current_code == code:
                decoded_vector.append(symbol)
                current_code = ""

    return decoded_vector
```

Code 5: Huffman Decode, Match the Encoded Vector.

B. Inverse Zig-Zag

Explained in section II-D the process of converting the [8 8] matrix into [1 64] vector to give the vector suitable for Huffman encode process. Now we will obtain a matrix from the vector by doing the reverse process.

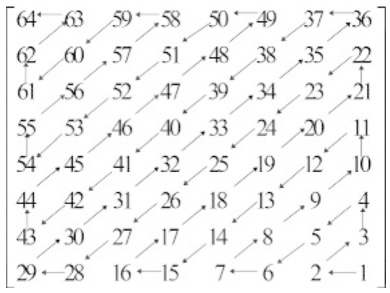
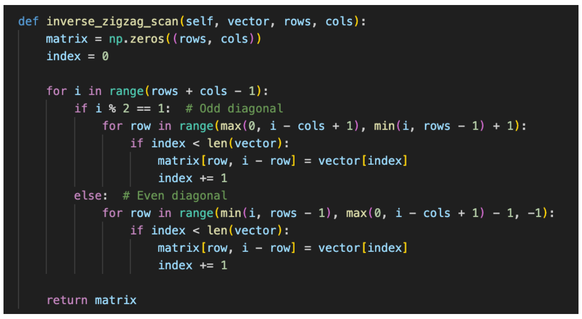


Fig. 15. Inverse Zig-Zag Scanning

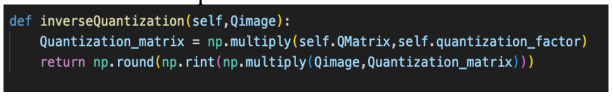


```python
def inverse_zigzag_scan(self, vector, rows, cols):
    matrix = np.zeros((rows, cols))
    index = 0

    for i in range(rows + cols - 1):
        if i % 2 == 1:  # Odd diagonal
            for row in range(max(0, i - cols + 1), min(i, rows - 1) + 1):
                if index < len(vector):
                    matrix[row, i - row] = vector[index]
                    index += 1
        else:  # Even diagonal
            for row in range(min(i, rows - 1), max(0, i - cols + 1) - 1, -1):
                if index < len(vector):
                    matrix[row, i - row] = vector[index]
                    index += 1

    return matrix
```

Code 6: Inverse Zig-Zag Scan 1x64 vector turn to 8x8 matrix.

C. *Inverse Quantization*

Quantization was done by dividing the 8x8 matrix, which was previously the DCT output, by the quantization matrix given in figure x. The purposes of quantization are described in section II-C. Now we will try to obtain the DCT output data again by dequantization. There will be a loss here because the round operation has been done before.



```python
def inverseQuantization(self,Qimage):
    Quantization_matrix = np.multiply(self.QMatrix,self.quantization_factor)
    return np.round(np.rint(np.multiply(Qimage,Quantization_matrix)))
```

Code 7: Inverse Quantization

8x8 data coming from Inverse Zigzag is multiplied by Quantization matrix.

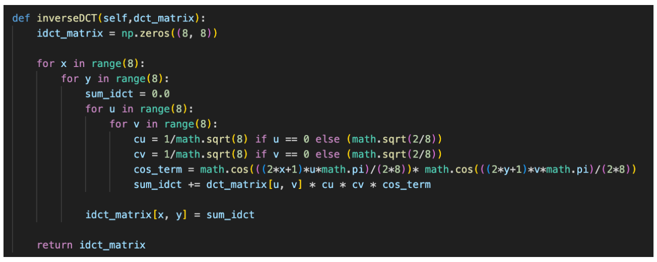
D. *Inverse Discrete Cosine Transform*

DCT input matrix was obtained by inverting the DCT process defined in Section II-B. While doing this process, the following mathematical expression was used.

$$f_{x,y} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\alpha(u)\alpha(v)F_{u,v}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

Fig. 16. Inverse DCT

Then, we reverse the zero centering process we did before (128 subtraction) and add 128 to each pixel value.



```python
def inverseDCT(self,dct_matrix):
    idct_matrix = np.zeros((8, 8))

    for x in range(8):
        for y in range(8):
            sum_idct = 0.0
            for u in range(8):
                for v in range(8):
                    cu = 1/math.sqrt(8) if u == 0 else (math.sqrt(2/8))
                    cv = 1/math.sqrt(8) if v == 0 else (math.sqrt(2/8))
                    cos_term = math.cos(((2*x+1)*u*math.pi)/(2*8))* math.cos(((2*y+1)*v*math.pi)/(2*8))
                    sum_idct += dct_matrix[u, v] * cu * cv * cos_term

            idct_matrix[x, y] = sum_idct

    return idct_matrix
```

Code 8: Inverse DCT

E. *Merge Blocks*

In the first phase of the project, the image was divided into 8x8 blocks. Although the output matrix of Inverse DCT was a lossy matrix, it produced matrices very similar to the 8x8 matrices in the first phase of the project. We obtain our image by combining all inverse DCT matrices into a 512x512 matrix.

The image decode process is completed successfully.

F. *PSNR and Image Output*

The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images. This ratio is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

PSNR value is 36.88815762229764 dB.

The resulting image:

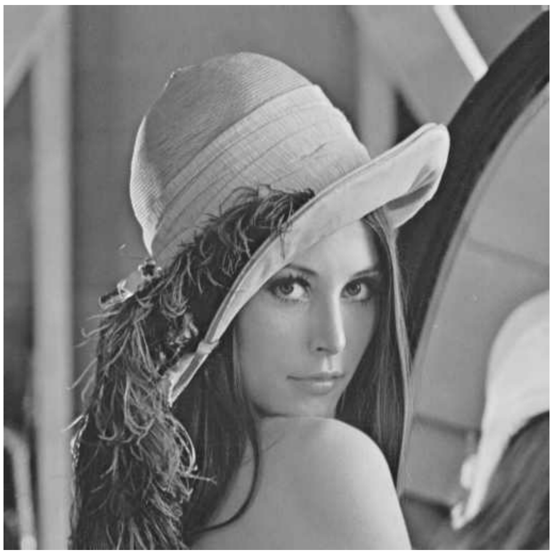


Fig. 17. The image recovered

There are distortions in this image due to rounding operations performed in quantization. But it is almost impossible to see this with the eye.

G. *Results of the First Phase of the Project*

Execution time	49.604254961013794 seconds
PSNR value	36.88815762229764 dB
Encoded data number	368808
Huffman code number	92
File size	45.02 Kb
Bit Per Pixel	1.41

Table 2: Result Table

IV. Rate Distortion Optimization

A. *Adaptive Quantization Matrix*

Rate-distortion optimization (RDO) is a method of improving image quality in image compression. The name refers to the optimization of the amount of distortion (loss of image quality) against the amount of data required to encode the image, the rate. While it is primarily used by image encoders, rate-distortion optimization can be used to improve quality in any encoding situation (image, video, audio, or otherwise) where decisions have to be made that affect both file size and quality simultaneously.

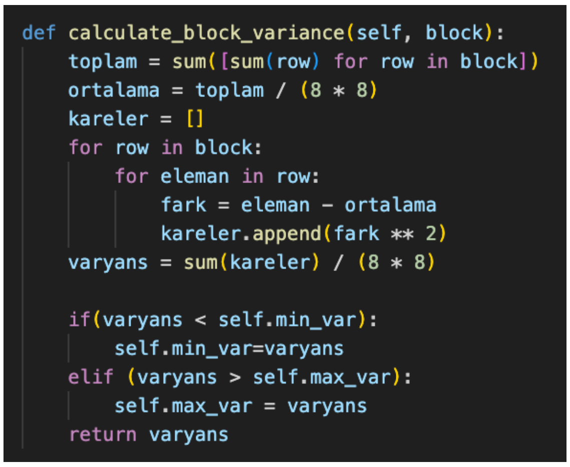
The purpose here is to find different quantization coefficients for each 8x8 matrix at the DCT output, multiply the matrices at the DCT output with different quantization matrices, and thus obtain better compression.

Adaptive quantization is a technique used in signal processing, particularly in data compression. It's a method of quantization, which is the process of mapping a large set of input values to a smaller set – a crucial step in digital signal processing. By adjusting the quantization process to the specific characteristics of the signal, it can achieve higher efficiency. This means better quality at lower bitrates, which is especially valuable in audio and video compression. It's also effective in handling signals with a wide dynamic range, as it can allocate more bits to represent more detailed parts of the signal, while using fewer bits for simpler parts.

When creating an Adaptive Quantization matrix, the variance of the 8x8 matrix to be quantized is taken into consideration. Dealing with adaptive quantization, especially in the context of matrix data (like images or multidimensional signal data), examining the variances of matrices is crucial for a few key reasons:

- Variance in a matrix can reveal important characteristics about the data. High variance areas in an image, for instance, might indicate edges, textures, or other important details. These areas might require finer quantization to preserve detail, as opposed to low variance areas which might represent smooth, uniform regions where coarser quantization can be applied without significant loss of perceived quality.
- By setting different threshold values for high and low variance blocks, we adjust the quantization factor according to these thresholds. For example, for blocks with very high variance we apply less compression (lower quantization factor), and for blocks with low variance we apply more compression.

The variance of each block is calculated as follows.



```python
def calculate_block_variance(self, block):
    toplam = sum([sum(row) for row in block])
    ortalama = toplam / (8 * 8)
    kareler = []
    for row in block:
        for eleman in row:
            fark = eleman - ortalama
            kareler.append(fark ** 2)
    varyans = sum(kareler) / (8 * 8)

    if(varyans < self.min_var):
        self.min_var=varyans
    elif (varyans > self.max_var):
        self.max_var = varyans
    return varyans
```

Code 9: Variance

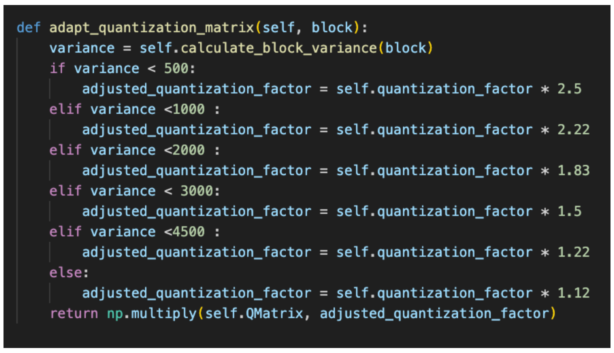
The variance range is [2.85 9125.22], and my quantization coefficients are set as follows according to the resulting variance ranges.

Quantization Factor = 1

0 < Variance < 500	= Quantization Factor * 2.50
500 < Variance < 1000	= Quantization Factor * 2.22
1000 < Variance < 2000	= Quantization Factor * 1.83
2000 < Variance < 3000	= Quantization Factor * 1.50
3000 < Variance < 4500	= Quantization Factor * 1.22
4500 < Variance	= Quantization Factor * 1.12

By means of these coefficients, we compress blocks with high variance more, while we compress blocks with low variance less. In this way, our number of code words decreases.

By multiplying the coefficients determined according to the variance with the quantization matrix shown in Figure 4 in Section II-C. Our adaptive quantization matrices are formed. All steps in the first phase of the project are repeated from the beginning. The only difference is using different quantization matrices for each block.



```python
def adapt_quantization_matrix(self, block):
    variance = self.calculate_block_variance(block)
    if variance < 500:
        adjusted_quantization_factor = self.quantization_factor * 2.5
    elif variance <1000 :
        adjusted_quantization_factor = self.quantization_factor * 2.22
    elif variance <2000 :
        adjusted_quantization_factor = self.quantization_factor * 1.83
    elif variance < 3000:
        adjusted_quantization_factor = self.quantization_factor * 1.5
    elif variance <4500 :
        adjusted_quantization_factor = self.quantization_factor * 1.22
    else:
        adjusted_quantization_factor = self.quantization_factor * 1.12
    return np.multiply(self.QMatrix, adjusted_quantization_factor)
```

Code 9: Adaptive Quantization Matrix Coeff Creator

B. *Results of the Second Phase of the Project*

We expect a higher compression and a lower PSNR value compared to the first part. In addition, visual distortions should now be visible.

Execution time	49.604254961013794 seconds
PSNR value	29.41262409442379 dB
Encoded data number	338620
Huffman code number	83
File size	41.33 Kb
Bit Per Pixel	1.29

Table 3: Result Table

The resulting image:

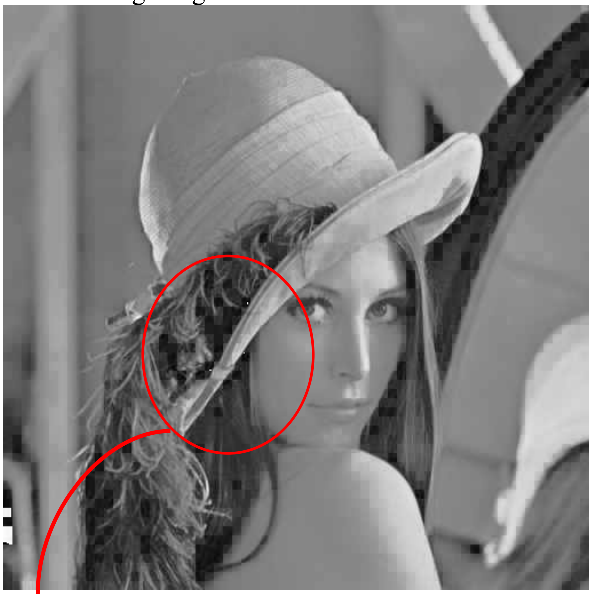
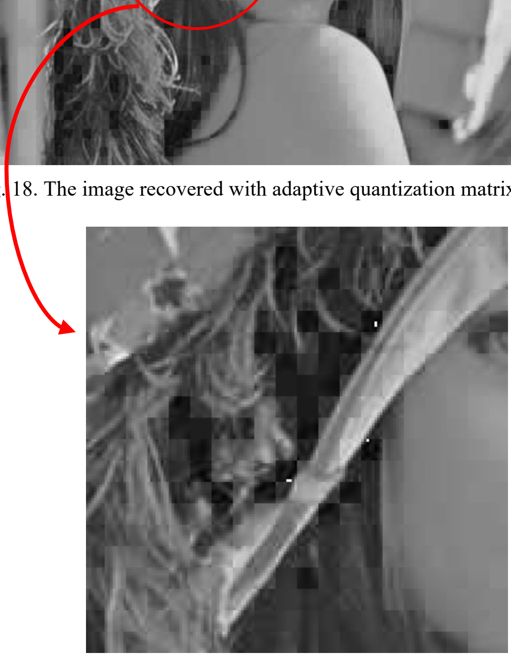


Fig. 18. The image recovered with adaptive quantization matrix



At this stage, distortions in the image can be easily seen with our eyes. This is an expected result since PSNR has a lower value.

V. Final Solution and conclusion

In conclusion, this project successfully demonstrates the application and effectiveness of JPEG compression on a 512x512 bitmap image. The process involved block splitting, Discrete Cosine Transform (DCT), quantization, zig-zag encoding, Huffman encoding, and their corresponding decoding techniques. The project highlighted the balance between compression efficiency and image quality, showcasing significant file size reduction while maintaining acceptable visual fidelity. The results, including execution time, PSNR values, and file sizes, underline the practicality of JPEG compression in image processing. This work not only provides a comprehensive understanding of JPEG compression mechanics but also opens avenues for further optimization and application in various digital imaging scenarios.

Resultant Table:

Execution time	49.604254961013794 seconds
FIRST PART DATAS	VALUES
PSNR value	36.88815762229764 dB
Encoded data number	368808
Huffman code number	92
File size	45.02 Kb
Bit Per Pixel	1.41
SECOND PART DATAS	VALUES
PSNR value	29.41262409442379 dB
Encoded data number	338620
Huffman code number	83
File size	41.33 Kb
Bit Per Pixel	1.29

VI. Application

In the code coded with Python, i coded a GUI by converting the project to .exe format so that the user does not have to deal with packages and can run the project directly. Although it is not very aesthetically pleasing in appearance, we have completed the code with an interface that is simple to use.

You can access the working principle and demo of our application from this link : https://youtu.be/3_Bpp2kSZps

Project File: Download Link

REFERENCES

[1] https://www.mathworks.com/help/vision/ref/psnr.html
[2] https://en.wikipedia.org/wiki/Rate–distortion_optimization
[3] https://en.wikipedia.org/wiki/JPEG
[4] https://www.geeksforgeeks.org/huffman-decoding/
[5] https://www.programiz.com/dsa/huffman-coding

PROJECT CODE:

```
from PIL import Image
import cv2 as cv
import numpy as np
import math
import matplotlib.pyplot as plt
from collections import Counter
import time
from tqdm import tqdm

import heapq
from collections import defaultdict

class Node:
    def __init__(self, symbol=None,
frequency=0, left=None, right=None):
        self.symbol = symbol
        self.frequency = frequency
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency <
other.frequency

class ImageCodec :
    def
__init__(self,input_image_path,output_image_
path,output_adapted_image_path,quantization_
factor):
        self.input_image_path =
input_image_path
        self.output_image_path =
output_image_path
        self.output_adapted_image_path =
output_adapted_image_path
        self.QMatrix= np.zeros((8,8))
        self.QMatrix =
[[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]]

        self.quantization_factor =
quantization_factor
        self.min_var = 10000
        self.max_var = 0


    def run(self):
        image =
self.getImage(self.input_image_path)
        height, width= image.shape
        print("\nInput Image size:",
height,"x",width,"\n")

        encoded_image,
huffman_codes,adapted_encoded_image,
adapted_huffman_codes =
self.encode_image(image)

        decoded_image, adapted_decoded_image
=
self.decode_image(encoded_image,huffman_code
s,adapted_encoded_image,
adapted_huffman_codes,image)

        height, width= decoded_image.shape
        adapted_height, adapted_width=
adapted_decoded_image.shape

        print("\nDecoded image shape:
",height, "x",width,"\n")
        print("Original Image:",image,"\n")
        print("JPEG
Image:",decoded_image,"\n")

        print("\nAdapted Decoded image
shape: ",adapted_height,
"x",adapted_width,"\n")
        print("Adapted JPEG
Image:",adapted_decoded_image,"\n")
```

```
        self.toImage(decoded_image)

self.toImageAdapted(adapted_decoded_image)


    def getImage(self,path):
        image = cv.imread(path,
cv.IMREAD_GRAYSCALE)

        return image


    def toImage(self,raw_data):
        image =
Image.fromarray(raw_data.astype(np.uint8))
        image.save(self.output_image_path)
        image.show()

    def toImageAdapted(self,raw_data):
        image =
Image.fromarray(raw_data.astype(np.uint8))

image.save(self.output_adapted_image_path)
        image.show()


    def get_8x8_block(self, image, x_count,
y_count):
        return image[(x_count - 1) *
8:x_count * 8, (y_count - 1) * 8:y_count *
8] # divide 8x8 blocks


    def takeDCT(self,matrix): # dct2
function formula
        dct_matrix = np.zeros((8,8))
        for u in range(8):
            for v in range(8):
                sum_dct = 0.0
                cu = 1/math.sqrt(8) if u ==
0 else (math.sqrt(2/8))
                cv = 1/math.sqrt(8) if v ==
0 else (math.sqrt(2/8))
                for x in range(8):
                    for y in range(8):
                        cos_term =
math.cos(((2*x+1)*u*math.pi)/(2*8))*
math.cos(((2*y+1)*v*math.pi)/(2*8))
                        sum_dct += matrix[x,
y] * cu * cv * cos_term

                dct_matrix[u, v] = sum_dct
        return dct_matrix
```

```

    def Quantization(self,matrix):
        Quantization_matrix =
np.multiply(self.QMatrix,self.quantization_f
actor)
        return
np.round(np.rint(np.divide(matrix,Quantizati
on_matrix)))


    # matrix element wise division

#and round to nearest integer
    def
QuantizationAdapted(self,matrix,Qadaptedmatr
ix):
        return
np.round(np.rint(np.divide(matrix,Qadaptedma
trix)))

    def ZigZag_Scan(self, matrix):
        rows, cols = len(matrix),
len(matrix[0])
        result = []

        for i in range(rows + cols - 1):
            if i % 2 == 0:  # Çift indeksli
satırlar (aşağı doğru)
                for j in range(min(i, rows -
1), max(0, i - cols + 1) - 1, -1):

result.append(matrix[j][i - j])
            else:  # Tek indeksli satırlar
(yukarı doğru)
                for j in range(max(0, i -
cols + 1), min(i, rows - 1) + 1):

result.append(matrix[j][i - j])

        return result


    def
build_huffman_tree(self,frequencies):
        heap = [Node(symbol=symbol,
frequency=frequency) for symbol, frequency
in frequencies.items()]
        while len(heap) > 1:
            heap.sort()
            left = heap.pop(0)
            right = heap.pop(0)
            internal_node =
Node(frequency=left.frequency +
right.frequency, left=left, right=right)
            heap.append(internal_node)
        return heap[0]


    def generate_huffman_codes(self, node,
code="", mapping=None):
        if mapping is None:
            mapping = {}
        if node.symbol is not None:
            mapping[node.symbol] = code
        if node.left is not None:

self.generate_huffman_codes(node.left, code
+ "0", mapping)
        if node.right is not None:

self.generate_huffman_codes(node.right, code
+ "1", mapping)
        return mapping


    def huffman_encode(self, vector):
        encoded_vector = 0
        frequencies = dict(Counter(vector))
        root =
self.build_huffman_tree(frequencies)
        codes =
self.generate_huffman_codes(root)
        encoded_vector = ''.join(codes[num]
for num in vector)
        return encoded_vector, codes


    def encode_image(self, image):
        print("Starting Image Encoding\n")
        print("--------------------------
"*4)

        image = image.astype(np.float16)
        average_num = 128.0
        x_count, y_count = 1, 1
        finish = 0
        progress_bar = tqdm(total=64*64,
desc="Encoding Image")

        image = image - average_num
        total_zigzag_scan = []
        total_Q_adapted_zigzag_scan = []

        #Dikkat eksenler farklı
```

```
        while finish != 1:
            matrix =
self.get_8x8_block(image, x_count, y_count)
            dct_matrix =
self.takeDCT(matrix)

            Qimage =
self.Quantization(dct_matrix)

            Q_adapted_matrix =
self.adapt_quantization_matrix(dct_matrix)
            Q_adapted_image =
self.QuantizationAdapted(dct_matrix,Q_adapte
d_matrix)

            zigzag_scan =
self.ZigZag_Scan(Qimage)
            zigzag_scan = list(map(int,
zigzag_scan))

            Q_adapted_zigzag_scan =
self.ZigZag_Scan(Q_adapted_image)
            Q_adapted_zigzag_scan =
list(map(int, Q_adapted_zigzag_scan))


total_zigzag_scan.extend(zigzag_scan)

total_Q_adapted_zigzag_scan.extend(Q_adapted
_zigzag_scan)

            progress_bar.update(1)

            if y_count == 64 and x_count !=
64:
                y_count = 1
                x_count += 1
            elif y_count == 64 and x_count
== 64:
                finish = 1
            else:
                y_count += 1

        encoded_image, huffman_codes =
self.huffman_encode(total_zigzag_scan)
        adapted_encoded_image,
adapted_huffman_codes =
self.huffman_encode(total_Q_adapted_zigzag_s
can)
        print(self.max_var," ",self.min_var)

        progress_bar.close()
        print("Image Encoding Finished\n")
        print("Encoded data
number:",len(encoded_image),"Huffman code
number: ",len(huffman_codes),"\n")
        print("Adapted Encoded data
number:",len(adapted_encoded_image),"Huffman
code number:
",len(adapted_huffman_codes),"\n")
        return encoded_image, huffman_codes
, adapted_encoded_image,
adapted_huffman_codes


    def huffman_decode(self, encoded_vector,
codes):

        current_code = ""
        decoded_vector = []

        # encoded vector den bit bit alarak
current code'a atar eger mevcut
        # huffman cod ile eşleşirse onun
symbol unu append eder
        for bit in encoded_vector:
            current_code += bit
            for symbol, code in
codes.items():
                if current_code == code:

decoded_vector.append(symbol)
                    current_code = ""

        return decoded_vector


    def inverse_zigzag_scan(self, vector,
rows, cols):
        matrix = np.zeros((rows, cols))
        index = 0

        for i in range(rows + cols - 1):
            if i % 2 == 1:  # Odd diagonal
                for row in range(max(0, i -
cols + 1), min(i, rows - 1) + 1):
                    if index < len(vector):
                        matrix[row, i - row]
= vector[index]
                        index += 1
            else:  # Even diagonal
                for row in range(min(i, rows
- 1), max(0, i - cols + 1) - 1, -1):
                    if index < len(vector):
                        matrix[row, i - row]
= vector[index]
```

```
                        index += 1

        return matrix


    def inverseQuantization(self,Qimage):
        Quantization_matrix =
np.multiply(self.QMatrix,self.quantization_f
actor)
        return
np.round(np.rint(np.multiply(Qimage,Quantiza
tion_matrix)))


    def
inverse_adapt_quantization_matrix(self,
Qimage, block):
        """Inverse of the
adapt_quantization_matrix function."""
        # Bloğun varyansını hesapla
        variance =
self.calculate_block_variance(block)
        if variance < 500:
        # Düşük varyans: Daha yüksek
sıkıştırma
            adjusted_quantization_factor =
self.quantization_factor * 2.5
        elif variance <1000 :
            adjusted_quantization_factor =
self.quantization_factor * 2.22
        elif variance <2000 :
            adjusted_quantization_factor =
self.quantization_factor * 1.83
        elif variance < 3000:
            # Orta varyans: Orta düzey
sıkıştırma
            adjusted_quantization_factor =
self.quantization_factor * 1.5
        elif variance <4500 :
            adjusted_quantization_factor =
self.quantization_factor * 1.22
        else:
            # Yüksek varyans: Daha az
sıkıştırma
            adjusted_quantization_factor =
self.quantization_factor * 1.12
        adapted_Q_matrix =
np.multiply(self.QMatrix,
adjusted_quantization_factor)
        # Kuantize edilmiş matrisi orijinal
DCT katsayılarına dönüştür
```

```
        return
np.round(np.rint(np.multiply(Qimage,
adapted_Q_matrix)))


    def inverseDCT(self,dct_matrix):
        idct_matrix = np.zeros((8, 8))

        for x in range(8):
            for y in range(8):
                sum_idct = 0.0
                for u in range(8):
                    for v in range(8):
                        cu = 1/math.sqrt(8)
if u == 0 else (math.sqrt(2/8))
                        cv = 1/math.sqrt(8)
if v == 0 else (math.sqrt(2/8))
                        cos_term =
math.cos(((2*x+1)*u*math.pi)/(2*8))*
math.cos(((2*y+1)*v*math.pi)/(2*8))
                        sum_idct +=
dct_matrix[u, v] * cu * cv * cos_term

                idct_matrix[x, y] = sum_idct

        return idct_matrix


    def
decode_image(self,encoded_image,huffman_code
s, adapted_encoded_image,
adapted_huffman_codes,image):
        print("Starting Image Decoding\n")
        print("--------------------------
"*4)
        x_count=  1
        y_count = 1
        average_num = 128.0
        progress_bar = tqdm(total=64*64*2,
desc="Decoding Image")

        decoded_image = np.empty((512,512))
        adapted_decoded_image =
np.empty((512,512))

        decoded_vector =
self.huffman_decode(encoded_image,huffman_co
des)
        adapted_decoded_vector =
self.huffman_decode(adapted_encoded_image,
adapted_huffman_codes)

        block_size = 64
```

```

        for i in range(4096):
            start_index = i * block_size
            end_index = start_index +
block_size
            block_vector =
decoded_vector[start_index:end_index]

            inverse_zigzag_scan =
self.inverse_zigzag_scan(block_vector,8,8)
            unquantized_matrix =
self.inverseQuantization(inverse_zigzag_scan
)
            idct_matrix =
self.inverseDCT(unquantized_matrix)

            progress_bar.update(1)

            decoded_image[(x_count - 1) * 8:
x_count * 8, (y_count - 1) * 8:y_count * 8]
= idct_matrix

            if y_count == 64 and x_count !=
64:
                y_count = 1
                x_count += 1
            else:
                y_count += 1

        decoded_image = decoded_image +
average_num
        decoded_image =
decoded_image.astype(np.uint8)

        x_count= 1
        y_count = 1
        for i in range(4096):
            start_index = i * block_size
            end_index = start_index +
block_size
            block_vector =
adapted_decoded_vector[start_index:end_index
]

            inverse_zigzag_scan =
self.inverse_zigzag_scan(block_vector,8,8)

            block =
self.get_8x8_block(image,x_count,y_count)
            dct_matrix = self.takeDCT(block)
            unquantized_matrix =
self.inverse_adapt_quantization_matrix(inver
se_zigzag_scan, dct_matrix)
```

```
            idct_matrix =
self.inverseDCT(unquantized_matrix)

            progress_bar.update(1)

            adapted_decoded_image[(x_count -
1) * 8: x_count * 8, (y_count - 1) *
8:y_count * 8] = idct_matrix


            if y_count == 64 and x_count !=
64:
                y_count = 1
                x_count += 1
            else:
                y_count += 1

        adapted_decoded_image =
adapted_decoded_image + average_num
        adapted_decoded_image =
adapted_decoded_image.astype(np.uint8)

        progress_bar.close()
        print("Image Decoding Finished")

        return decoded_image,
adapted_decoded_image

    def calculate_PSNR(self, original,
compressed):
        mse = np.mean(np.power((original -
compressed), 2))
        max_pixel = 255.0
        psnr = 20 * math.log10(max_pixel /
math.sqrt(mse))
        return psnr




    def calculate_block_variance(self,
block):
        toplam = sum([sum(row) for row in
block])
        ortalama = toplam / (8 * 8)
        kareler = []
        for row in block:
            for eleman in row:
                fark = eleman - ortalama
                kareler.append(fark ** 2)
        varyans = sum(kareler) / (8 * 8)
```

```
        if(varyans < self.min_var):
            self.min_var=varyans
        elif (varyans > self.max_var):
            self.max_var = varyans
        return varyans

    def adapt_quantization_matrix(self,
block):
        variance =
self.calculate_block_variance(block)
        if variance < 500:
            adjusted_quantization_factor =
self.quantization_factor * 2.5
        elif variance <1000 :
            adjusted_quantization_factor =
self.quantization_factor * 2.22
        elif variance <2000 :
            adjusted_quantization_factor =
self.quantization_factor * 1.83
        elif variance < 3000:
            adjusted_quantization_factor =
self.quantization_factor * 1.5
        elif variance <4500 :
            adjusted_quantization_factor =
self.quantization_factor * 1.22
        else:
            adjusted_quantization_factor =
self.quantization_factor * 1.12
        return np.multiply(self.QMatrix,
adjusted_quantization_factor)




if __name__ == "__main__":

    # get the start time
    st = time.time()

    input_image_path = "lena_gray.bmp"
    output_image_path="jpeg_lena.bmp"

output_adapted_image_path="jpeg_lena_adapted
.bmp"
    codec_instance =
ImageCodec(input_image_path,output_image_pat
h,output_adapted_image_path,quantization_fac
tor = 1)
    codec_instance.run()

    # get the end time
    et = time.time()
    # get the execution time
    elapsed_time = et - st
    print('Execution time:', elapsed_time,
'seconds')

    original =
codec_instance.getImage(input_image_path)
    compressed =
codec_instance.getImage(output_image_path)
    adapted =
codec_instance.getImage(output_adapted_image
_path
                                            )
    psnr_value =
codec_instance.calculate_PSNR(original,
compressed)
    psnr_adapted_value =
codec_instance.calculate_PSNR(original,adapt
ed)

    print(f"PSNR value is {psnr_value} dB")
    print(f"Adapted PSNR value is
{psnr_adapted_value} dB")
```