MINI PROJECT

Submitted in partial fulfilment of the requirements for the Award of Degree of

BACHELOR OF TECHNOLOGY

in

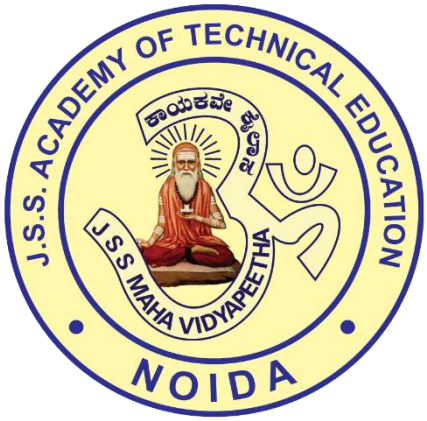
ELECTRONICS AND COMMUNICATION ENGINEERING

on

LOSSY IMAGE COMPRESSION ENCODER BASED ON JPEG STANDARD

SUBMITTED BY:

Amish Verma 2000910310033



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

Prefixed by: Dr. Chhaya Dalela

JSS ACADEMY OF TECHNICAL EDUCATION

C-20/1 SECTOR-62, NOIDA

2023- 2024

DECLARATION

I hereby declare that the mini project work on "Lossy Image Compression Encoder Based on JPEG Standard" submitted to the JSS Academy of Technical Education, Noida is a record of the original work done by me under the guidance of Dr. Chhaya Dalela, my project mentor and the internship work is submitted in the partial fulfilment of the requirement of the award of degree in Bachelor of Technology in Electronics and Communication department of JSS Academy of Technical Education, Noida.

Name: Amish Verma (2000910310033)

Signature:

ACKNOWLEDGEMENT

I am extremely grateful to **Dr. Amarjeet Singh**, Principal JSS Academy of Technical Education, Noida, **Dr. Arun Kumar G.**, Head of Department, Department of Electronics & Communication Engineering, and **Mrs. Rajeshwari Bhat**, Internship Coordinator, for providing all the required resources for the successful completion of this mini project and D**r. Chhaya Dalela**, Associate Professor, Department of Electronics & Communication Engineering, for their valuable suggestions and guidance in the preparation of the mini project report. I express my thanks to all staff members and friends for all the help and coordination extended in bringing out this mini project successfully.

Amish Verma (2000910310033) Dr. Chhaya Dalela (Associate Professor)

ABSTRACT

This project focuses on implementing a lossy image compression encoder based on the JPEG standard. We delve into the intricacies of the JPEG compression algorithm, covering stages like colour space transformation, DCT, quantization, zigzag scanning, and Huffman coding. The Python implementation of the compression encoder is detailed, showcasing code snippets and parameters. Results are evaluated using metrics like compression ratio. The discussion highlights trade-offs and challenges. The concluding chapter summarizes findings, draws conclusions, and suggests future project implementation directions. This project contributes insights into applying JPEG compression techniques using Python programming language which is one of the most widely used languages.

TABLE OF CONTENTS

List Of Figures

CHAPTER 1: INTRODUCTION

Multimedia images play a vital role in preserving life's moments, but their large data size requires compression for practical use. JPEG (Joint Photographic Experts Group) compression, introduced in 1992, addresses this issue effectively. It is the most widely used image compression standard globally and is commonly referred to as JPG. JPEG allows adjustable compression, balancing storage size and image quality. It achieves a typical compression ratio of 10:1 with minimal loss in image quality. The compression is based on the discrete cosine transform (DCT); a method proposed by Nasir Ahmed in 1972[1]. JPEG is extensively used in digital cameras and photographic images, specifically in the JPEG/Exif format. This format is widely known as JPEG. The algorithm is suitable for photographs and paintings with small tone and colour variations. Its file format facilitates easy exchange of bitstreams across various platforms and applications.

1.1 Background

The field of image processing has witnessed significant advancements, with image compression playing a crucial role in reducing storage requirements and facilitating efficient transmission of digital images. Image compression techniques aim to minimize the file size of an image while preserving essential visual information. This is particularly important in various applications such as multimedia, communications, and medical imaging.

1.2 Types of Images

1. Binary Images: Image data are stored as an *m*-by-*n* logical matrix in which values of 0 and 1 are interpreted as black and white, respectively. Some toolbox functions can also interpret an *m*-by-*n* numeric matrix as a binary image, where values of 0 are black and all nonzero values are white.

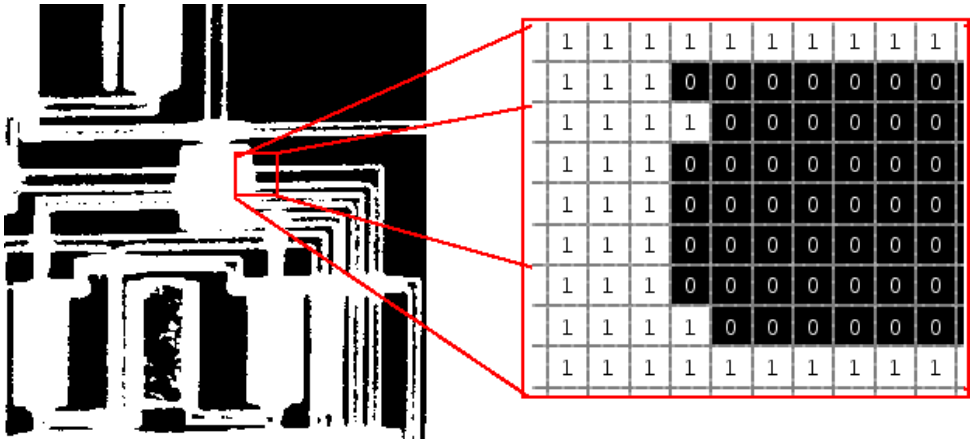


Fig 1.1: Binary Image

2. Indexed Images: Image data are stored as an *m*-by-*n* numeric matrix whose elements are direct indices into a colormap. Each row of the colormap specifies the red, green, and blue components of a single colour.
   - For single or double arrays, integer values range from [1, *p*].
   - For logical, uint8, or uint16 arrays, values range from [0, *p*-1].
   - The colormap is a *c*-by-3 array of data type double with values in the range [0, 1].

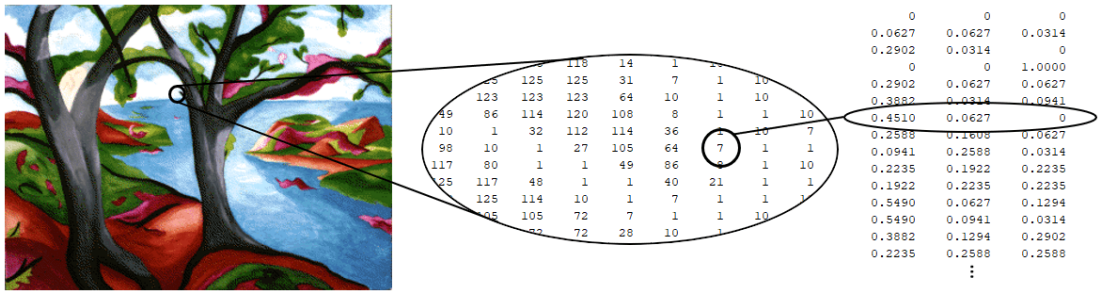


Fig 1.2: Indexed Image

3. Greyscale Images: Image data are stored as an *m*-by-*n* numeric matrix whose elements specify intensity values. The smallest value indicates black, and the largest value indicates white.
   - For single or double arrays, values range from [0, 1].
   - For uint8 arrays, values range from [0, 255].
   - For uint16, values range from [0, 65535].
   - For int16, values range from [-32768, 32767].

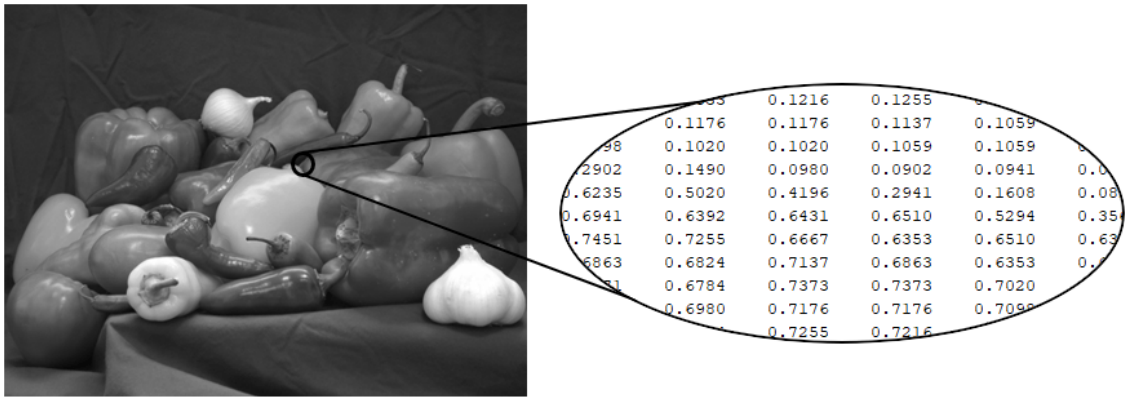


Fig 1.3: Greyscale Image

4. TrueColor Images: Image data are stored as an m-by-n numeric matrix whose elements specify intensity values. The smallest value indicates black and the largest value indicates white. For single or double arrays, values range from [0, 1].
   - For single or double arrays, values range from [0, 1].
   - For uint8 arrays, values range from [0, 255].
   - For uint16, values range from [0, 65535].
   - For int16, values range from [-32768, 32767].

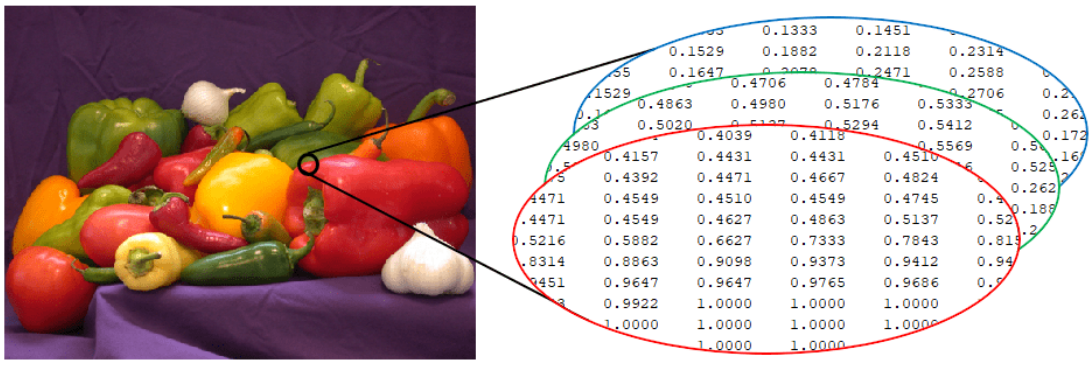


Fig 1.4: TrueColor Image

5. HDR Images: Dynamic range refers to the range of brightness levels. The dynamic range of real-world scenes can be quite high. High dynamic range (HDR) images attempt to capture the whole tonal range of real-world scenes (called scene-referred), using 32-bit floating-point values to store each colour channel. The figure depicts the red, green, and blue channels of a tone mapped HDR image with original pixel values

in the range [0, 3.2813]. Tone mapping is a process that reduces the dynamic range of an HDR image to the range expected by a computer monitor or screen.

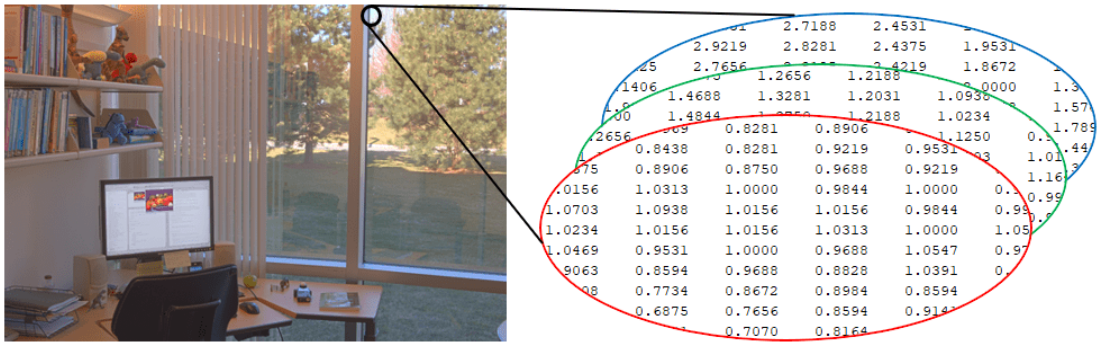


Fig 1.5: HDR Image

1.3 Objective

The primary objective of this project is to implement the JPEG (Joint Photographic Experts Group) image compression encoder. JPEG is a widely used and standardized image compression method known for its ability to achieve high compression ratios with acceptable loss in image quality. The programming language used in this project is Python. Python was chosen for its easily understandable syntax, simplicity, and ease of running the code on almost any platform.

1.4 Significance

The significance of image compression lies in its capacity to reduce the data size of images, making them more manageable for storage and transmission. In applications like web browsing, where bandwidth is a critical factor, and in storage-limited environments, effective image compression becomes indispensable. JPEG compression strikes a balance between compression efficiency and perceptual image quality.

1.5 Scope and Limitations

The scope of this project encompasses the implementation of JPEG compression encoder, with a focus on understanding the key components of the algorithm. The limitation of this project is

the absence of a decoder implementation and a module to efficiently save the compressed images to the disk.

CHAPTER 2: IMAGE COMPRESSION

Image compression serves as a fundamental process in the realm of digital content, strategically employed to diminish the storage footprint of images while safeguarding their core information. This transformation yields a compressed image of lower quality compared to the pristine original, yet this compromise is pivotal for rendering images wieldier for storage, transmission, and download purposes. The inherent balancing act between diminished quality and augmented practicality positions compressed images as pragmatic assets across a spectrum of applications, where optimizing resource utilization remains a paramount concern.

2.1 Lossless Compression

Lossless compression is a sophisticated data compression technique that employs algorithms to transform a compressed image's data back into its original form without any loss of information. Unlike lossy compression, which sacrifices some data to achieve higher compression ratios, lossless compression maintains the integrity of every bit of data in the original file.

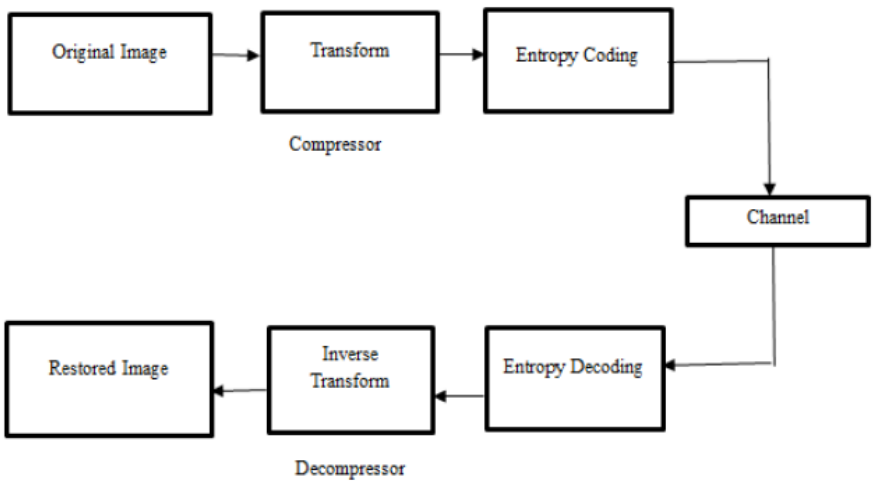


Fig 2.1: Block diagram of Lossless compression method

The key characteristics are:

1. Data Prevention: The primary characteristic of lossless compression is its ability to precisely reconstruct the original data, ensuring that no noticeable loss occurs during

compression and decompression. This fidelity to the original data is crucial in applications where preserving every detail is paramount, such as medical imaging, text documents, and certain types of graphics.

2. Reversibility: One distinctive feature of lossless compression is its reversible nature, meaning that the compression process can be undone, and the file can be perfectly restored to its original state. This reversibility is particularly valuable in scenarios where data accuracy is non-negotiable, such as archiving, data transmission, and digital preservation.

3. File format retention: In lossless compression, the file format is retained in its entirety, and the compressed data holds the complete information required to reconstruct the original image. This characteristic is especially advantageous in situations where maintaining the original file format is crucial for compatibility with specific applications or systems.

While lossless compression ensures data preservation, it typically achieves lower compression ratios compared to lossy compression. Consequently, the compressed files may not be as compact as their lossy counterparts, making lossless compression more suitable for scenarios where file size is not the primary concern, and data accuracy and integrity take precedence.

2.2 Lossy Compression

Lossy compression is a data compression technique designed to reduce the size of a file by selectively removing certain data elements. Unlike lossless compression, lossy compression involves a trade-off between file size reduction and a decrease in image or sound quality. This method is extensively used in multimedia applications such as images, audio, and video.

The key characteristics are:

1. File size reduction: The primary objective of lossy compression is to significantly reduce the size of a file by discarding redundant or less critical information. This reduction in file size makes it more feasible for storage, transmission, and efficient use in various applications.

2. Quality Degradation: While the compression process results in a smaller file size, it is accompanied by a decrease in the quality of the multimedia content. This trade-off between file size and quality is carefully managed to ensure that the loss in quality is acceptable for the intended purpose, such as web streaming or storage optimization.

3. Irreversibility: Unlike lossless compression, lossy compression is irreversible. Once the compression is applied, the original file cannot be precisely reconstructed. This characteristic is especially important to consider when the integrity of the original data is critical, and irreversible alterations are acceptable.

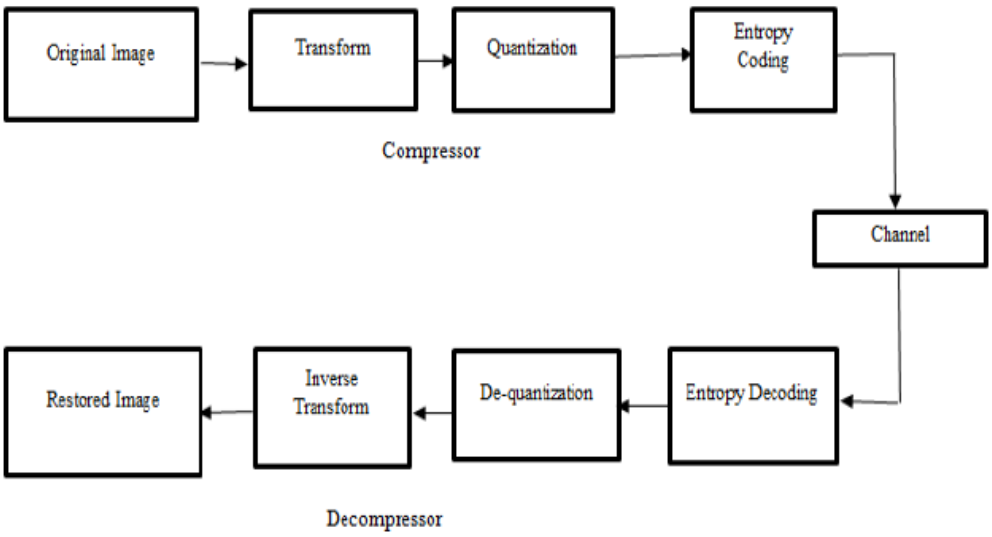


Fig 2.2: Block diagram of Lossy compression method

Lossy compression finds widespread use in multimedia applications, including images, audio, and video. It is employed in scenarios where a certain degree of quality loss is deemed acceptable, and the priority lies in achieving a significant reduction in file size.

2.3 Image Compression

The compression process initiates with the conversion of the RGB image into YIQ if necessary. The resulting image undergoes a series of transformations, including DCT transformation and quantization, to achieve efficient compression.

2.3.1 DCT Transformation

A crucial step in image compression, the Discrete Cosine Transform (DCT) is applied to the transformed image. DCT analyses frequency components, capturing essential information for compression purposes.

2.3.2 Quantization

In the quantization stage, unnecessary data is eliminated to reduce both the size and quality of the image. This step is vital in achieving compression while maintaining a balance between image size and perceptual quality.

2.3.3 Encoding for protection

To secure the compressed image, encoding is implemented. This involves changing the names of values in the quantized image, adding a layer of security. The image is passed through a channel encoder during this process.

2.3.4 Decompression Process Overview

Decompression involves reversing the steps taken during compression to restore the image to its original form. The image goes through decoding, inverse quantization, inverse transformation, and retrieval of lost data.

2.3.5 Image decoding

The compressed image is decoded by passing it through the channel decoder, followed by the entropy decoder. This step retrieves the encoded information for further processing.

2.3.6 Inverse Quantization

Inverse quantization is applied to restore the lost data in the quantization stage, ensuring that the decompressed image retains essential details.

2.3.7 Inverse Transformation

The image is subjected to an inverse transformation, reversing the DCT process, to reconstruct the original image from the compressed representation.

2.3.8 Redundancy Techniques in Image Compression

Image compression aims to reduce the amount of data required to represent a digital image by eliminating redundancies. Several techniques address different types of redundancies in the image.

1. Spatial Redundancy: Spatial redundancy refers to the correlation between neighbouring pixel values. In image compression, techniques are employed to exploit and reduce this redundancy for more efficient representation.
2. Spectral Redundancy: Spectral redundancy involves the correlation between different spectral bands and colour planes in an image. Image compression techniques target spectral redundancies to enhance compression efficiency.
3. Psycho-visual Redundancy: Psycho-visual redundancy accounts for the removal of unimportant information perceived by the human visual system. Image compression techniques leverage this redundancy to optimize compression without compromising perceived image quality.

CHAPTER 3: DISCRETE COSINE TRANSFORM

3.1 Introduction

The Discrete Cosine Transform (DCT) stands as a pivotal algorithm, originating in 1972 through the pioneering work of Nasir Ahmed, aimed at the rapid computation of the Fourier transform. Its application extends prominently into digital signal processing, enabling the extraction of patterns and facilitating the implementation of Wiener filtering techniques. Much like its counterpart, the Discrete Fourier Transform (DFT), the DCT serves to transition an image from the spatial domain to the frequency domain. Unlike some other transforms, the DCT does not inherently reduce the number of bits required for each block. However, its adoption in international standards is primarily attributable to its commendable performance-to-computational cost ratio. The DCT boasts a real transform with notable advantages in energy compaction, allowing for an efficient representation of image information. While various variants of the DCT exist, the foundational principles remain paramount in understanding its applications. Noteworthy among its features is its swiftness in implementation, outpacing the speed of the Discrete Fourier Transform. The DCT, owing to its cosine basis functions, exhibits shift variance, with a significant exception—convolution implementation with the DCT has not been conclusively proven feasible. The continuous exploration and refinement of the DCT, guided by its inherent strengths and applications, continue to underscore its significance in various domains, particularly in image processing and compression. The evolution of the DCT paradigm, with its intriguing properties and practical advantages, continues to contribute to the landscape of digital signal processing and multimedia applications.

The Discrete Cosine Transform (DCT) serves as a mathematical tool that represents a finite sequence of data points as a sum of cosine functions oscillating at various frequencies. Its widespread application is notably prominent in the realm of digital media, encompassing digital audio, videos, radios, and speech coding. The versatile nature of the DCT renders it invaluable in an array of science and engineering applications, underscoring its significance in the analysis and processing of digital signals and multimedia content.

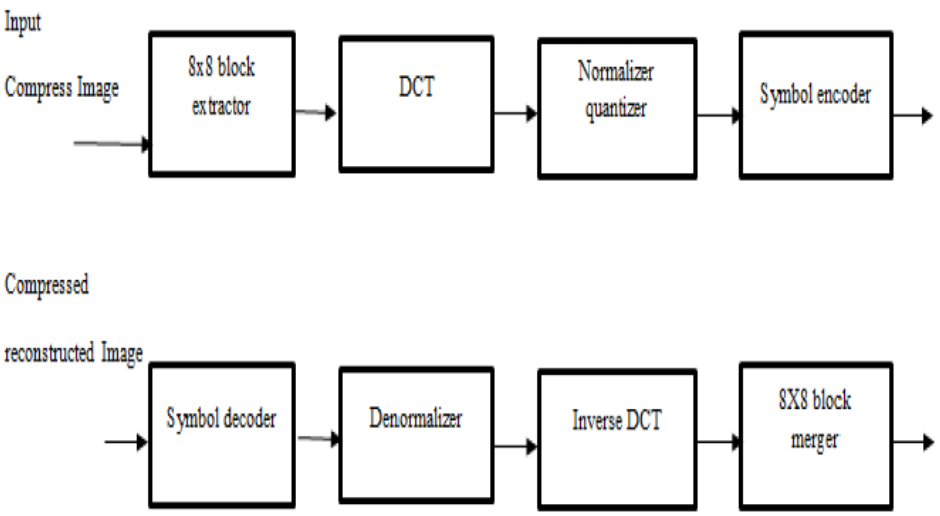


Fig 3.1: DCT formed by Image compression.

3.2 JPEG Process

1. The first step in image compression is that to break the image into 8^8 blocks.
2. While the breaking is done, we must apply the DCT to each image.
3. Therefore, by quantization each block is get compressed.
4. The array of compressed blocks that constitute the image gets stored by drastically reduced amount of space.

3.3 Principles and applications of DCT

The compression of images initiates with the transformation of the original RGB-coloured image into either YIQ or YUV colour spaces. In YIQ, the 'Y' component represents brightness, while 'IQ' signifies chrominance. Similarly, in YUV, 'Y' denotes brightness, and 'UV' encapsulates the colour information of the image. The colour-transformed image then undergoes sampling through an 8x8 block extractor, resulting in a sampled image containing values *f(u,v)*, where 'u' and 'v' represent different blocks of the image.

Subsequently, the values are subjected to the Discrete Cosine Transform (DCT), a critical step in the compression process. This transform helps express the finite sequence of data points as a sum of cosine functions oscillating at different frequencies. The unwanted

values are then removed through quantization, a process that reduces both the size and quality of the image.

In the encoding process, zero and non-zero values are arranged in a zigzag form to represent *f(u,v)* values as dc1 and ac1, respectively. The values lost throughout this process culminate in the creation of the compressed image. The DCT, akin to the Discrete Fourier Transform, is implemented using only real numbers and is primarily related to Fourier series coefficients of symmetrically and periodically sequenced data.

Utilizing cosine and sine functions is crucial for compression, as fewer cosine functions are needed to approximate a typical signal. The cosine functions express the discrete Fourier transform, with the DCT specifically related to Fourier series coefficients of symmetrically and periodically sequenced data.

There exist eight standard DCT variants, with the type-II DCT being the most widely used. Often referred to as block compression, the DCT compresses data into discrete DCT blocks, with various block sizes, such as 8x8 pixels for standard DCT and 4x4 and 32x32 pixels for varied DCT. The DCT possesses a robust energy compaction property, enabling the attainment of high-quality images at substantial data compression ratios. However, it's worth noting that heavy DCT compression may lead to blocky compression artifacts. In conclusion, the DCT, with its versatile applications and energy compaction capabilities, finds widespread use in numerous domains, contributing significantly to image compression and related fields.

3.4 Advantages and Disadvantages

3.4.1: Advantages:

- This Image format has been using since long time and is extremely portable.
- The format is very easy to read so they can be understood easily by printers to print.
- JPEG format is used to store high resolution images where some of them are blur shows small in size format.
- These format images are easy to share from devices to devices.

- The size in JPEG images can be reduced and compressed. In which the file format is suitable for transferring images from internet to devices.

3.4.2: Disadvantages:

- Compression in JPEG format loses certain actual contents of the image.
- Quality of the image is reduced after compression owing to the loss of actual content of the image.
- This compression is not suitable for images which having sharp line sand edges.
- This type of format is not capable for handling animated graphic images.
- In JPEG format only 8bit images are supported. But modern high-resolution digital cameras support 10, 12, 14or16 bit images. The images that are stored in JPEG format, in which the extra information is disabled, results to decreasing in image quality.

CHAPTER 4: ALGORITHM, OBSERVATIONS AND RESULTS

4.1 Python Code

```
from math import ceil
from collections import Counter

import cv2
import numpy as np

# define quantization tables
QTY = np.array([[16, 11, 10, 16, 24, 40, 51, 61],  # luminance
quantization table
                [12, 12, 14, 19, 26, 48, 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]])

QTC = np.array([[17, 18, 24, 47, 99, 99, 99, 99],  # chrominance
quantization table
                [18, 21, 26, 66, 99, 99, 99, 99],
                [24, 26, 56, 99, 99, 99, 99, 99],
                [47, 66, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99]])
# define window size
windowSize = len(QTY)

def zigzag(matrix: np.ndarray) -> np.ndarray:
    """
    computes the zigzag of a quantized block
    :param numpy.ndarray matrix: quantized matrix
    :returns: zigzag vectors in an array
    """
    # initializing the variables
    h = 0
    v = 0
    v_min = 0
    h_min = 0
    v_max = matrix.shape[0]
    h_max = matrix.shape[1]
    i = 0
    output = np.zeros((v_max * h_max))

    while (v < v_max) and (h < h_max):
```

```
        if ((h + v) % 2) == 0:  # going up
            if v == v_min:
                output[i] = matrix[v, h]  # first line
                if h == h_max:
                    v = v + 1
                else:
                    h = h + 1
                i = i + 1
            elif (h == h_max - 1) and (v < v_max):  # last column
                output[i] = matrix[v, h]
                v = v + 1
                i = i + 1
            elif (v > v_min) and (h < h_max - 1):  # all other cases
                output[i] = matrix[v, h]
                v = v - 1
                h = h + 1
                i = i + 1
        else:  # going down
            if (v == v_max - 1) and (h <= h_max - 1):  # last line
                output[i] = matrix[v, h]
                h = h + 1
                i = i + 1
            elif h == h_min:  # first column
                output[i] = matrix[v, h]
                if v == v_max - 1:
                    h = h + 1
                else:
                    v = v + 1
                i = i + 1
            elif (v < v_max - 1) and (h > h_min):  # all other cases
                output[i] = matrix[v, h]
                v = v + 1
                h = h - 1
                i = i + 1
        if (v == v_max - 1) and (h == h_max - 1):  # bottom right element
            output[i] = matrix[v, h]
            break
    return output


def trim(array: np.ndarray) -> np.ndarray:
    """
    in case the trim_zeros function returns an empty array, add a zero to
the array to use as the DC component
    :param numpy.ndarray array: array to be trimmed
    :return numpy.ndarray:
    """
    trimmed = np.trim_zeros(array, 'b')
    if len(trimmed) == 0:
        trimmed = np.zeros(1)
    return trimmed


def run_length_encoding(array: np.ndarray) -> list:
```

```
    """
    finds the intermediary stream representing the zigzags
    format for DC components is <size><amplitude>
    format for AC components is <run_length, size> <Amplitude of non-zero>
    :param numpy.ndarray array: zigzag vectors in array
    :returns: run length encoded values as an array of tuples
    """
    encoded = list()
    run_length = 0
    eob = ("EOB",)

    for i in range(len(array)):
        for j in range(len(array[i])):
            trimmed = trim(array[i])
            if j == len(trimmed):
                encoded.append(eob)  # EOB
                break
            if i == 0 and j == 0:  # for the first DC component
                encoded.append((int(trimmed[j]).bit_length(), trimmed[j]))
            elif j == 0:  # to compute the difference between DC
components
                diff = int(array[i][j] - array[i - 1][j])
                if diff != 0:
                    encoded.append((diff.bit_length(), diff))
                else:
                    encoded.append((1, diff))
                run_length = 0
            elif trimmed[j] == 0:  # increment run_length by one in case
of a zero
                run_length += 1
            else:  # intermediary steam representation of the AC
components
                encoded.append((run_length, int(trimmed[j]).bit_length(),
trimmed[j]))
                run_length = 0
            # send EOB
        if not (encoded[len(encoded) - 1] == eob):
            encoded.append(eob)
    return encoded


def get_freq_dict(array: list) -> dict:
    """
    returns a dict where the keys are the values of the array, and the
values are their frequencies
    :param numpy.ndarray array: intermediary stream as array
    :return: frequency table
    """
    #
    data = Counter(array)
    result = {k: d / len(array) for k, d in data.items()}
    return result
```

```
def find_huffman(p: dict) -> dict:
    """
    returns a Huffman code for an ensemble with distribution p
    :param dict p: frequency table
    :returns: huffman code for each symbol
    """
    # Base case of only two symbols, assign 0 or 1 arbitrarily; frequency
does not matter
    if len(p) == 2:
        return dict(zip(p.keys(), ['0', '1']))

    # Create a new distribution by merging lowest probable pair
    p_prime = p.copy()
    a1, a2 = lowest_prob_pair(p)
    p1, p2 = p_prime.pop(a1), p_prime.pop(a2)
    p_prime[a1 + a2] = p1 + p2

    # Recurse and construct code on new distribution
    c = find_huffman(p_prime)
    ca1a2 = c.pop(a1 + a2)
    c[a1], c[a2] = ca1a2 + '0', ca1a2 + '1'

    return c


def lowest_prob_pair(p):
    # Return pair of symbols from distribution p with lowest probabilities
    sorted_p = sorted(p.items(), key=lambda x: x[1])
    return sorted_p[0][0], sorted_p[1][0]

# read image
imgOriginal = cv2.imread('marbles.bmp', cv2.IMREAD_COLOR)
# convert BGR to YCrCb
img = cv2.cvtColor(imgOriginal, cv2.COLOR_BGR2YCR_CB)
width = len(img[0])
height = len(img)
y = np.zeros((height, width), np.float32) + img[:, :, 0]
cr = np.zeros((height, width), np.float32) + img[:, :, 1]
cb = np.zeros((height, width), np.float32) + img[:, :, 2]
# size of the image in bits before compression
totalNumberOfBitsWithoutCompression = len(y) * len(y[0]) * 8 + len(cb) *
len(cb[0]) * 8 + len(cr) * len(cr[0]) * 8
# channel values should be normalized, hence subtract 128
y = y - 128
cr = cr - 128
cb = cb - 128
# 4: 2: 2 subsampling is used # another subsampling scheme can be used
# thus chrominance channels should be sub-sampled
# define subsampling factors in both horizontal and vertical directions
SSH, SSV = 2, 2
# filter the chrominance channels using a 2x2 averaging filter # another
type of filter can be used
crf = cv2.boxFilter(cr, ddepth=-1, ksize=(2, 2))
cbf = cv2.boxFilter(cb, ddepth=-1, ksize=(2, 2))
```

```
crSub = crf[::SSV, ::SSH]
cbSub = cbf[::SSV, ::SSH]

# check if padding is needed,
# if yes define empty arrays to pad each channel DCT with zeros if
necessary
yWidth, yLength = ceil(len(y[0]) / windowSize) * windowSize, ceil(len(y) /
windowSize) * windowSize
if (len(y[0]) % windowSize == 0) and (len(y) % windowSize == 0):
    yPadded = y.copy()
else:
    yPadded = np.zeros((yLength, yWidth))
    for i in range(len(y)):
        for j in range(len(y[0])):
            yPadded[i, j] += y[i, j]

# chrominance channels have the same dimensions, meaning both can be
padded in one loop
cWidth, cLength = ceil(len(cbSub[0]) / windowSize) * windowSize,
ceil(len(cbSub) / windowSize) * windowSize
if (len(cbSub[0]) % windowSize == 0) and (len(cbSub) % windowSize == 0):
    crPadded = crSub.copy()
    cbPadded = cbSub.copy()
# since chrominance channels have the same dimensions, one loop is enough
else:
    crPadded = np.zeros((cLength, cWidth))
    cbPadded = np.zeros((cLength, cWidth))
    for i in range(len(crSub)):
        for j in range(len(crSub[0])):
            crPadded[i, j] += crSub[i, j]
            cbPadded[i, j] += cbSub[i, j]

# get DCT of each channel
# define three empty matrices
yDct, crDct, cbDct = np.zeros((yLength, yWidth)), np.zeros((cLength,
cWidth)), np.zeros((cLength, cWidth))

# number of iteration on x axis and y axis to calculate the luminance
cosine transform values
hBlocksForY = int(len(yDct[0]) / windowSize)  # number of blocks in the
horizontal direction for luminance
vBlocksForY = int(len(yDct) / windowSize)  # number of blocks in the
vertical direction for luminance
# number of iteration on x axis and y axis to calculate the chrominance
channels cosine transforms values
hBlocksForC = int(len(crDct[0]) / windowSize)  # number of blocks in the
horizontal direction for chrominance
vBlocksForC = int(len(crDct) / windowSize)  # number of blocks in the
vertical direction for chrominance

# define 3 empty matrices to store the quantized values
yq, crq, cbq = np.zeros((yLength, yWidth)), np.zeros((cLength, cWidth)),
np.zeros((cLength, cWidth))
# and another 3 for the zigzags
```

```
yZigzag = np.zeros(((vBlocksForY * hBlocksForY), windowSize * windowSize))
crZigzag = np.zeros(((vBlocksForC * hBlocksForC), windowSize *
windowSize))
cbZigzag = np.zeros(((vBlocksForC * hBlocksForC), windowSize *
windowSize))

yCounter = 0
for i in range(vBlocksForY):
    for j in range(hBlocksForY):
        yDct[i * windowSize: i * windowSize + windowSize, j * windowSize:
j * windowSize + windowSize] = cv2.dct(
            yPadded[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        yq[i * windowSize: i * windowSize + windowSize, j * windowSize: j
* windowSize + windowSize] = np.ceil(
            yDct[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize] / QTY)
        yZigzag[yCounter] += zigzag(
            yq[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        yCounter += 1
yZigzag = yZigzag.astype(np.int16)

# either crq or cbq can be used to compute the number of blocks
cCounter = 0
for i in range(vBlocksForC):
    for j in range(hBlocksForC):
        crDct[i * windowSize: i * windowSize + windowSize, j * windowSize:
j * windowSize + windowSize] = cv2.dct(
            crPadded[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        crq[i * windowSize: i * windowSize + windowSize, j * windowSize: j
* windowSize + windowSize] = np.ceil(
            crDct[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize] / QTC)
        crZigzag[cCounter] += zigzag(
            crq[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        cbDct[i * windowSize: i * windowSize + windowSize, j * windowSize:
j * windowSize + windowSize] = cv2.dct(
            cbPadded[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        cbq[i * windowSize: i * windowSize + windowSize, j * windowSize: j
* windowSize + windowSize] = np.ceil(
            cbDct[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize] / QTC)
        cbZigzag[cCounter] += zigzag(
            cbq[i * windowSize: i * windowSize + windowSize, j *
windowSize: j * windowSize + windowSize])
        cCounter += 1
crZigzag = crZigzag.astype(np.int16)
cbZigzag = cbZigzag.astype(np.int16)

# find the run length encoding for each channel
```

```
# then get the frequency of each component in order to form a Huffman
dictionary
yEncoded = run_length_encoding(yZigzag)
yFrequencyTable = get_freq_dict(yEncoded)
yHuffman = find_huffman(yFrequencyTable)

crEncoded = run_length_encoding(crZigzag)
crFrequencyTable = get_freq_dict(crEncoded)
crHuffman = find_huffman(crFrequencyTable)

cbEncoded = run_length_encoding(cbZigzag)
cbFrequencyTable = get_freq_dict(cbEncoded)
cbHuffman = find_huffman(cbFrequencyTable)

# calculate the number of bits to transmit for each channel
# and write them to an output file
file = open("CompressedImage.asfh", "w")
yBitsToTransmit = str()
for value in yEncoded:
    yBitsToTransmit += yHuffman[value]

crBitsToTransmit = str()
for value in crEncoded:
    crBitsToTransmit += crHuffman[value]

cbBitsToTransmit = str()
for value in cbEncoded:
    cbBitsToTransmit += cbHuffman[value]

if file.writable():
    file.write(yBitsToTransmit + "\n" + crBitsToTransmit + "\n" +
cbBitsToTransmit)
file.close()

totalNumberOfBitsAfterCompression = len(yBitsToTransmit) +
len(crBitsToTransmit) + len(cbBitsToTransmit)
print(
    "Compression Ratio is " + str(
        np.round(totalNumberOfBitsWithoutCompression /
totalNumberOfBitsAfterCompression, 1)))
```

4.2 Results

We choose an image saved in bitmap image format which is an uncompressed, lossless image file format. The image is called “marbles.bmp” and is of 4,165 KB approximately in size.

After we run our compression algorithm, we see a 10.9 :1 reduction in the size of the image. What that means is that we were able to reduce the size of our image by approximately eleven times. That’s on-par with the standard JPEG compression.

See the images below to see the difference in size before and after we run our compression algorithm.

```
print("Size of the image before compression:", bits_to_something(totalNumberOfBitsWithoutCompression, "mb"), "MB")
```

```
Size of the image before compression: 4.06 MB
```

Fig 4.1: Code snippet showing size of the image before compression

```
print("Size of the image after chroma subsampling:", bits_to_something(totalBits, "mb"), "MB")
```

```
Size of the image after chroma subsampling: 2.03 MB
```

Fig 4.2: Code snippet showing size of the image after chroma subsampling

```
print("Size of the image after compression:", bits_to_something(totalNumberOfBitsAfterCompression, "mb"), "MB")
```

```
Size of the image after compression: 0.37 MB
```

Fig 4.3: Code snippet showing size of the image after compression

```
[69]: _, ax = plt.subplots(1, 2, figsize=(10, 10))
      ax[0].imshow(imgOriginal)
      ax[0].axis('off')
      ax[0].set_title('Image Before Compression')
      ax[1].imshow(rgb_image)
      ax[1].axis('off')
      ax[1].set_title('Image After Compression')

[69]: Text(0.5, 1.0, 'Image After Compression')
```

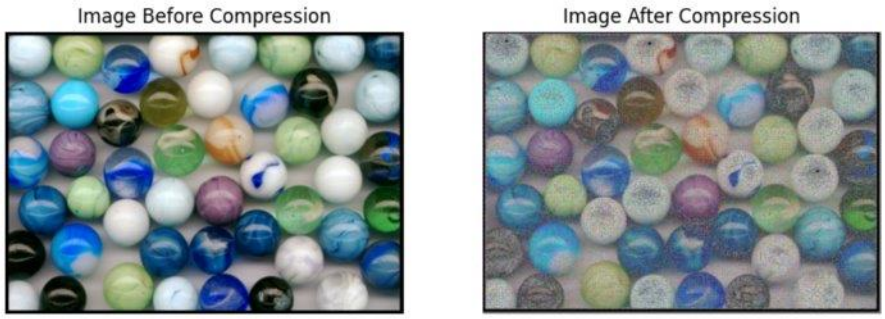


Fig 4.4: marbles.bmp before and after compression

4.3 Conclusion

JPEG image compression standard is effective in compressing images with a compression ratio of 10:1 without visible loss in quality. Most of the JPEG standard's lossy compression is based on experimental results. Also, baseline JPEG standard is rarely implemented in production-grade code. Implementing a production-grade JPEG encoder and decoder is a complicated task which might require months of coding to properly implement. What I have done in this project is implement a stripped-out version of the JPEG standard to study the algorithms that are used in one of the most widely used compression algorithms in the world. While our encoder was able to achieve a 10.9:1 theoretical compression ratio, it left some compression artifacts in the final decoded image.

Further work can be done in this project to implement a JPEG image decoder. Moreover, using other quantization tables can also give us better results in the encoder we have implemented.

REFERENCES

[1] K. Sharma and K. Gupta, "Lossless data compression techniques and their performance," 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 2017, pp. 256-261, doi: 10.1109/CCAA.2017.8229810.

[2] G. K. Wallace, "The JPEG still picture compression standard," in IEEE Transactions on Consumer Electronics, vol. 38, no. 1, pp. xviii-xxxiv, Feb. 1992, doi: 10.1109/30.125072.

[3] G. Hudson, A. Léger, B. Niss and I. Sebestyén, "JPEG at 25: Still Going Strong," in IEEE MultiMedia, vol. 24, no. 2, pp. 96-103, Apr.-June 2017, doi: 10.1109/MMUL.2017.38.

[4] Image Types: https://www.mathworks.com/help/images/image-types-in-the-toolbox.html, Accessed on 01/12/23.

[5] Understanding and Decoding a JPEG Image using Python: https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/, Accessed on 01/12/23.

[6] Lei Wang, Jiaji Wu, Licheng Jiao, Li Zhang and Guangming Shi, "Lossy to lossless image compression based on reversible integer DCT," 2008 15th IEEE International Conference on Image Processing, San Diego, CA, 2008, pp. 1037-1040, doi: 10.1109/ICIP.2008.4711935.