

**Visvesvaraya Technological University
Belagavi-590018, Karnataka**



A Mini Project Report on

“Image Compression using DCT Algorithm”

Submitted in partial fulfillment of the requirement for the
File Structure Laboratory with Mini-Project
[17ISL68]

**Bachelor of Engineering
in
Information Science and Engineering**

Submitted by
GIRISH M [1JT17IS011]

Under the Guidance of
Mr.Vadiraja A
Asst.Professor,Department of ISE



**Department of Information Science and Engineering
Jyothy Institute of Technology
Tataguni, Bengaluru-560082**

Jyothy Institute of Technology
Department of Information Science and Engineering
Tataguni, Bengaluru-560082



CERTIFICATE

Certified that the mini project work entitled “**Image Compression using DCT Algorithm**“ carried out by **GIRISH M [1JT17IS011]** bonafide student of Jyothy Institute of Technology, in partial fulfillment for the award of **Bachelor of Engineering in Information Science and Engineering** department of the **Visvesvaraya Technological University, Belagavi** during the year **2019-2020**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirements in respect of Mini Project work prescribed for the said Degree.

Prof. Vadiraja A

Guide , Asst. Professor

Dept. Of ISE

Dr.Harshvardhan Tiwari

Associate Professor and HOD

Dept. Of ISE

External Viva Examiner

Signature with Date:

- 1.
- 2.

ACKNOWLEDGEMENT

Firstly, we are very grateful to this esteemed institution **Jyothy Institute of Technology** for providing us an opportunity to complete our project.

We express our sincere thanks to our Principal **Dr. Gopalakrishna K** for providing us with adequate facilities to undertake this project.

We would like to thank **Dr. Harshvardhan Tiwari, Associate Professor and Head of Information Science and Engineering** Department for providing for his valuable support.

We would like to thank our guide **Prof. Vadiraja A , Asst. Prof.** for his keen interest and guidance in preparing this work.

Finally, we would thank all our friends who have helped us directly or indirectly in this project.

GIRISH M

1JT17IS011

ABSTRACT

Image compression is the application of data compression on digital images. Image compression can be lossy or lossless. No information is lost in lossless compression. Lossy compression reduces bits by removing unwanted information. In this project it is being attempted to implement basic Image compression using only basic python language. In this project the lossy compression techniques have been used, where data loss cannot affect the image clarity in this area. Image compression addresses the problem of reducing the amount of data required to represent a digital image. It is also used for reducing the redundancy that is nothing but avoiding the duplicate data. It also reduces the storage area to load an image. For this purpose, I've used Discrete Cosine Transform which is also adequate for most compression applications. The discrete cosine transform (DCT) is a mathematical function that transforms digital image data from the spatial domain to the frequency domain.

INDEX

SL No	Description	Page No.
	Chapter 1	
1	INTRODUCTION	
1.1	Introduction to File Structures	1
1.2	Introduction to File Systems	2
1.3	Introduction to Data Compression	3
1.4	Scope & Importance	3
	Chapter 2	
2	REQUIREMENT ANALYSIS AND DESIGN	
2.1	Domain Understanding	4
2.2	Classification of Requirements	4
2.2	System & Time Analysis	4
	Chapter 3	
3	IMPLEMENTATION	
3.1	INTRODUCTION	6
3.2	GUI & Driver Code	6
3.3	Compression	9
3.4	Zig-zag & Inverse zig-zag	13
3.5	Decompression	16
	Chapter 4	
4	RESULTS AND SNAPSHOTS	
4.1	Snapshots & Observation	18
4.2	Conclusion & References	26

CHAPTER 1

INTRODUCTION

CHAPTER 1

INTRODUCTION

1.1 Introduction to File Structures

In simple terms, a file is a collection of data stored on mass storage (e.g. disk or tape). But there is one important distinction that must be made at the outset when discussing file structures. And that is the difference between the logical and physical organization of the data.

On the whole a file structure will specify the logical structure of the data, that is the relationships that will exist between data items independently of the way in which these relationships may actually be realized within any computer. It is this logical aspect that we will concentrate on. The physical organization is much more concerned with optimizing the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of physical store there will be a number of units of the logical structure (probably records) to be stored in it.

For example, if we were to store a tree structure on a magnetic disk, the physical organization would be concerned with the best way of packing the nodes of the tree on the disk given the access characteristics of the disk. Like all subjects in computer science the terminology of file structures has

evolved higgledy-piggledy without much concern for consistency, ambiguity, or whether it was possible to make the kind of distinctions that were important.

It was only much later that the need for a well-defined, unambiguous language to describe file structures became apparent. In particular, there arose a need to communicate ideas about file structures without getting bogged down by hardware considerations.

1.2 Introduction to File System

In computing, a file system or file system controls how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with noway to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each groups of data is called a “file”. The structure and logic rules used to manage the groups of information and their names is called a “file system”. There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more. Some file systems have been designed to be used for specific applications. For example, the ISO 9660 file system is designed specifically for optical discs.

File systems can be used on numerous different types of storage devices that use different kinds of media. The most common storage device in use today is a hard disk drive. Other kinds of media that are used include flash memory, magnetic tapes, and optical discs. In some cases, such as with tmpfs, the computer's main memory (random-access memory, RAM) is used to create a temporary file system for short-term use.

Some file systems are used on local data storage devices; others provide file access via a network protocol (for example, NFS, SMB, or 9P clients). Some file systems are “virtual”. Meaning that the supplied “files” (called virtual files) are computed on request (e.g. procfs) or are merely a mapping into a different file system used as a backing store. The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations .

1.3 Data Compression

Data compression is the process of modifying, encoding or converting the bits structure of data in such a way that it consumes less space on disk. It enables reducing the storage size of one or more data instances or elements. Data compression is also known as source coding or bit-rate reduction

A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. The DCT, first proposed by Nasir Ahmed in 1972, is a widely used transformation technique in signal processing and data compression. It is used in most digital media, including digital images (such as JPEG and HEIF, where small high-frequency components can be discarded), digital video (such as MPEG and H.26x), digital audio (such as Dolby Digital, MP3 and AAC), digital television (such as SDTV, HDTV and VOD), digital radio (such as AAC+ and DAB+), and speech coding (such as AAC-LD, Siren and Opus)

1.4 Scope and Importance

When adding images to your website, it's a good idea to compress them. This not only increases your website's loading speed, it also saves on the amount of data your visitors use to visit your site. There are various methods you can use to cut down on your image file sizes, including changing file types, decreasing quality, minimizing image dimensions, and more.

CHAPTER 2
REQUIREMENT ANALYSIS AND
DESIGN

CHAPTER 2

REQUIREMENT ANALYSIS AND DESIGN

2.1 Domain Understanding

The main object is to compress the given image, by converting the image to Grayscale and feeding it to DCT & RLE algorithm using zigzag function. The outcome of this project is to ease the user for compressing images with a friendly, hassle-free, simple yet efficient GUI.

2.2 Classification of Requirements

User Requirements:

1. OS : Windows / Linux/ macOS.
2. Python-3.7.

Software Requirements:

Programming-language:

1. Python.

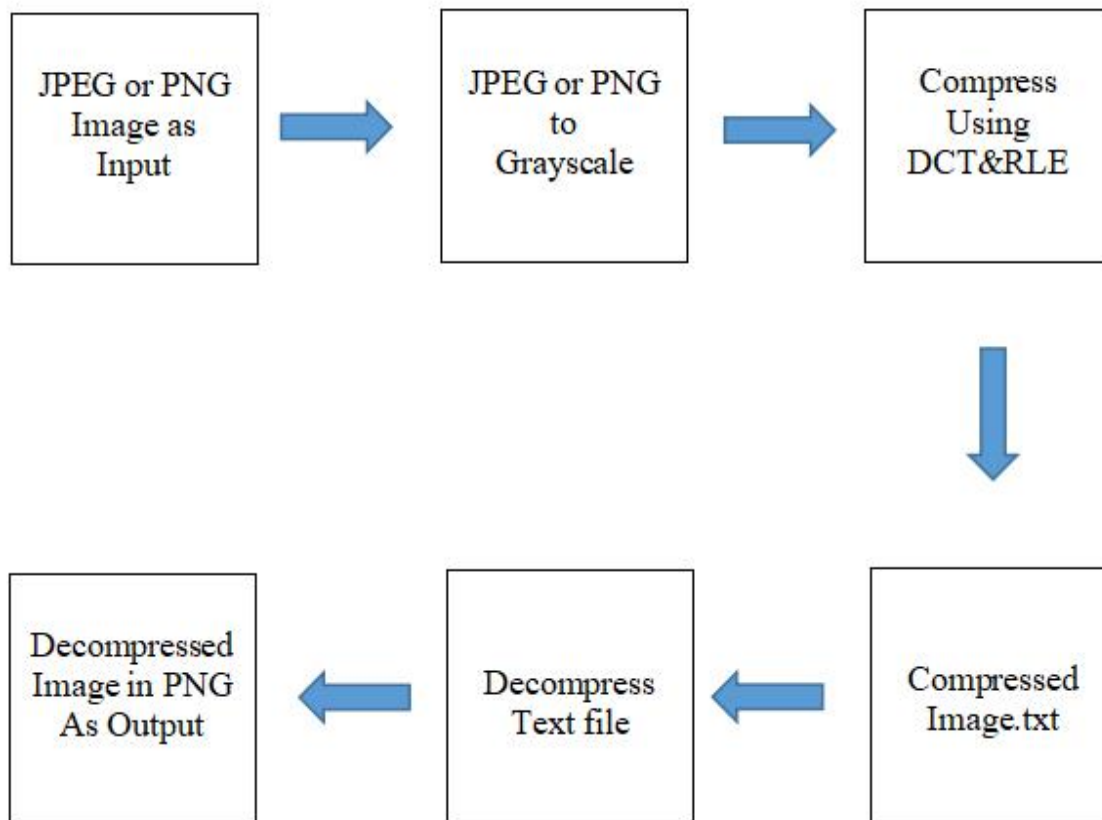
Programming modules:

1. Tkinter-python (GUI Tool Kit).
2. Opencv-python (Computer Vision Library).
3. PIL (Python Imaging Library).
4. OS (Python Library to Interact with Operating System).
5. Time (Python Library to record run-time).

2.3 System and Time Analysis

When the program is executed, it loads a simple GUI which primarily asks the user to enter the name of the image . After entering image name user can compress the image by pressing compress button When user press Compress button, the program loads the image and it starts to compress the image. After the completion of compressing, the user is given information about compression such as time taken, original image size, compressed file size, etc.

If user wants to Decompress the compressed image in txt format . He can simply press Decompress button which decompresses the image and the user is given information about compression such as time taken, original image size, compressed file size, etc



Based On the test and trials its been observed that execution and compression time increase linearly with increase in size of the image. Where as Decompression time stays almost constant. Another observation made is that PNG images tend to have better compression rates than that of JPEG and compression rate depends on no of different colors present in image. Graphs supporting this observation is attached below in results chapter.

CHAPTER 3

IMPLEMENTATION

CHAPTER 3

IMPLEMENTATION

3.1 Introduction

The Whole project can be split into 3 parts based on the work done by those, namely they are:

- 1.GUI (Graphical User Interface)&Driver Code
- 2.Compression
- 3.Decompression
- 4.zig-zag

3.2 GUI (Graphical User Interface) & Driver Code

Python has a lot of GUI frameworks, but Tkinter is the only framework that's built into the Python standard library. Tkinter has several strengths. It's cross-platform, so the same code works on Windows, macOS, and Linux. Visual elements are rendered using native operating system elements, so applications built with Tkinter look like they belong on the platform where they're run.

Although Tkinter is considered the de-facto Python GUI framework, it's not without criticism. One notable criticism is that GUIs built with Tkinter look outdated. If you want a shiny, modern interface, then Tkinter may not be what you're looking for.

However, Tkinter is lightweight and relatively painless to use compared to other frameworks. This makes it a compelling choice for building GUI applications in Python, especially for applications where a modern sheen is unnecessary, and the top priority is to build something that's functional and cross-platform quickly.

I've used this interface to build a simple interface for this project. which has text field, two buttons, and few labels to display some essential information.. It's easy to implement and work with as it uses native python libraries, below is snapshot from this project used to render the GUI.

Import all the required modules namely:

- | | |
|------------|------------------|
| 1. Tkinter | 4. Compression |
| 2. PIL | 5. Decompression |
| 3. CV2 | 6. OS |

```

import PIL.Image
import cv2
import tkinter as tk
from tkinter import *
import os
import compression as pia
import decompression as pib
import time

def convert(seconds):
    seconds = seconds % (24 * 3600)
    seconds %= 3600
    minutes = seconds // 60
    seconds %= 60
    return "%02d:%02d" % (minutes, seconds) #formatting

def dcompress():
    t1=time.time()
    pib.abc()
    t2=time.time()
    t=str(convert(t2-t1))
    tk.Label(master,text="Decompression done and saved as ").grid(row=34,column=6)
    tk.Label(master,text="uncompressed_image.bmp with execution time ").grid(row=36,column=6)
    tk.Label(master,text=t+"sec").grid(row=38,column=6)
    file_stats2=os.stat("uncompressed_image.png")
    size=file_stats2.st_size/ (1024 * 1024)
    tk.Label(master,text="Decompressed image size: "+str(round(size,2))+ "MB").grid(row=40,column=6)

def compress():
    img=e1.get()
    if img!="":
        image=cv2.imread(img,cv2.IMREAD_GRAYSCALE )
        #cv2.imshow("original image in grayscale",image)
        t1= time.time()
        pia.abc(img)
        t2=time.time()
        t=str(convert(t2-t1))
        tk.Label(master,text="Compression done and saved as img.txt \n with execution time"+ t +"sec").grid(row=16,column=6)
        file_stats = os.stat(img)

        file_stats1 = os.stat("image.txt")
        size=file_stats1.st_size/ (1024 * 1024)
        size1=file_stats1.st_size/ (1024 * 1024)
        tk.Label(master,text="original image size: "+str(round(size,2))+ "MB").grid(row=18,column=6)
        tk.Label(master,text="compressed txt file size:"+str(round(size1,2))+ "MB").grid(row=20,column=6)

master = tk.Tk()
#master.geometry("400x400")
master.title("Image Compression using DCT")
#for img input
label1=tk.Label(master,text="Img Name").grid(row=1,column=5)

tk.Label(master,text="Note image will be converted to grayscale ").grid(row=4,column=6)
tk.Label(master,text="and then compressed and decompressed").grid(row=6,column=6)
tk.Label(master,text="resulting in grayscale img as output").grid(row=8,column=6)
tk.Label(master,text="even if original image is in RGB").grid(row=10,column=6)
e1 = tk.Entry(master, width=20)
e1.grid(row=1, column=6)

tk.Button(master,text='Quit', command=master.quit).grid(row=14,column=12,sticky=tk.W,pady=4)
tk.Button(master,text='Compress', command=compress).grid(row=14,column=5,sticky=tk.W,pady=4)
tk.Button(master,text='Decompress', command=dcompress).grid(row=14,column=6,sticky=tk.W,pady=4)

tk.mainloop()

```

Fig 3.2.1

```

def compress():
    img=e1.get()
    if img!="":
        image=cv2.imread(img,cv2.IMREAD_GRAYSCALE )
        #cv2.imshow("original image in grayscale",image)
        t1= time.time()
        pia.abc(img)
        t2=time.time()
        t=str(convert(t2-t1))
        tk.Label(master,text="Compression done and saved as img.txt \n with execution time"+ t +"sec").grid(row=16,column=6)
        file_stats = os.stat(img)

        file_stats1 = os.stat("image.txt")
        size=file_stats1.st_size/ (1024 * 1024)
        size1=file_stats1.st_size/ (1024 * 1024)
        tk.Label(master,text="original image size: "+str(round(size,2))+ "MB").grid(row=18,column=6)
        tk.Label(master,text="compressed txt file size:"+str(round(size1,2))+ "MB").grid(row=20,column=6)

master = tk.Tk()
#master.geometry("400x400")
master.title("Image Compression using DCT")
#for img input
label1=tk.Label(master,text="Img Name").grid(row=1,column=5)

tk.Label(master,text="Note image will be converted to grayscale ").grid(row=4,column=6)
tk.Label(master,text="and then compressed and decompressed").grid(row=6,column=6)
tk.Label(master,text="resulting in grayscale img as output").grid(row=8,column=6)
tk.Label(master,text="even if original image is in RGB").grid(row=10,column=6)
e1 = tk.Entry(master, width=20)
e1.grid(row=1, column=6)

tk.Button(master,text='Quit', command=master.quit).grid(row=14,column=12,sticky=tk.W,pady=4)
tk.Button(master,text='Compress', command=compress).grid(row=14,column=5,sticky=tk.W,pady=4)
tk.Button(master,text='Decompress', command=dcompress).grid(row=14,column=6,sticky=tk.W,pady=4)

tk.mainloop()

```

Fig 3.2.2

In above figs, I've have imported all the modules and we have few functions namely convert, compress, decompress. Where convert takes one parameter seconds in Int and returns minutes:seconds form. Compress reads the image name from text field e1,den call's the abc function from compression to compress the image and then displays all necessary info on the window and opens original image read in grayscale in separate window. The dcompress call's the abc function from Decompression to to decompress the image by reading image.text created by

compress and generates PNG image and displays necessary information on this window and image in another window.

3.3 Compression

This project consists of three algorithms first the image is quantized using Quantization, then compressed using DCT and in the encoded to text format and compressed using RLE we will go through all three algorithms

Quantization:

Quantization, involved in image processing, is a lossy compression technique achieved by compressing a range of values to a single quantum value. When the number of discrete symbols in a given stream is reduced, the stream becomes more compressible. For example, reducing the number of colors required to represent a digital image makes it possible to reduce its file size.

The human eye is fairly good at seeing small differences in brightness over a relatively large area, but not so good at distinguishing the exact strength of a high frequency (rapidly varying) brightness variation. This fact allows one to reduce the amount of information required by ignoring 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. This is the main lossy operation in the whole process. 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.

This is an example of DCT coefficient matrix:

$$\begin{bmatrix} -415 & -33 & -58 & 35 & 58 & -51 & -15 & -12 \\ 5 & -34 & 49 & 18 & 27 & 1 & -5 & 3 \\ -46 & 14 & 80 & -35 & -50 & 19 & 7 & -18 \\ -53 & 21 & 34 & -20 & 2 & 34 & 36 & 12 \\ 9 & -2 & 9 & -5 & -32 & -15 & 45 & 37 \\ -8 & 15 & -16 & 7 & -8 & 11 & 4 & 7 \\ 19 & -28 & -2 & -26 & -2 & 7 & -44 & -21 \\ 18 & 25 & -12 & -44 & 35 & 48 & -37 & -3 \end{bmatrix}$$

A common quantization matrix is:

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

Dividing the DCT coefficient matrix element-wise with this quantization matrix, and rounding to integers results in:

-26	-3	-6	2	2	-1	0	0
0	-3	4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-4	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

For example, using -415 (the DC coefficient) and rounding to the nearest integer

$$\text{round}\left(\frac{-415}{16}\right) = \text{round}(-25.9375) = -26$$

Typically this process will result in matrices with values primarily in the upper left (low frequency) corner. By using a zig-zag ordering to group the non-zero entries and run length encoding, the quantized matrix can be much more efficiently stored than the non-quantized version.

DCT (Discrete Cosine Transform):

Discrete Cosine Transform is used in lossy image compression because it has very strong energy compaction, i.e., its large amount of information is stored in very low frequency component of a signal and rest other frequency having very small data which can be stored by using very less number of bits (usually, at most 2 or 3 bit).

To perform DCT Transformation on an image, first we have to fetch image file information (pixel value in term of integer having range 0 – 255) which we divide in block of 8 X 8 matrix and then we apply discrete cosine transform on that block of data.

After applying discrete cosine transform, we will see that its more than 90% data will be in lower frequency component. For simplicity, I've took a matrix of size 8 X 8 having all value as 255 (considering image to be completely white) and we are going to perform 2-D discrete cosine transform on that to observe the output.

Let we are having a 2-D variable named matrix of dimension 8 X 8 which contains image information and a 2-D variable named DCT of same dimension which contain the information after applying discrete cosine transform. So, we have the formula.

$$dct[i][j] = c_i * c_j \left(\sum_{k=0}^{m-1} \sum_{l=0}^{n-1} matrix[k][l] * \cos((2*k+1)*i*pi/2*m) * \cos((2*l+1)*j*pi/2*n) \right)$$

where :

If i=0:

$$c_i = 1/\sqrt{m},$$

Else

$$c_i = \sqrt{2}/\sqrt{m}$$

Similarly,

If j=0:

$$c_j = 1/\sqrt{n},$$

Else

$$c_j = \sqrt{2}/\sqrt{n}$$

and we have to apply this formula to all the value, i.e., from i=0 to m-1 and j=0 to n-1. In this project both m=n=8.

RLE (Run-Length Encoding):

RLE is probably the easiest compression algorithm .It replaces sequences of the same data values. Suppose the following string of data (17 bytes) has to be compressed:

ABBBBBBBBBBCDEEEF

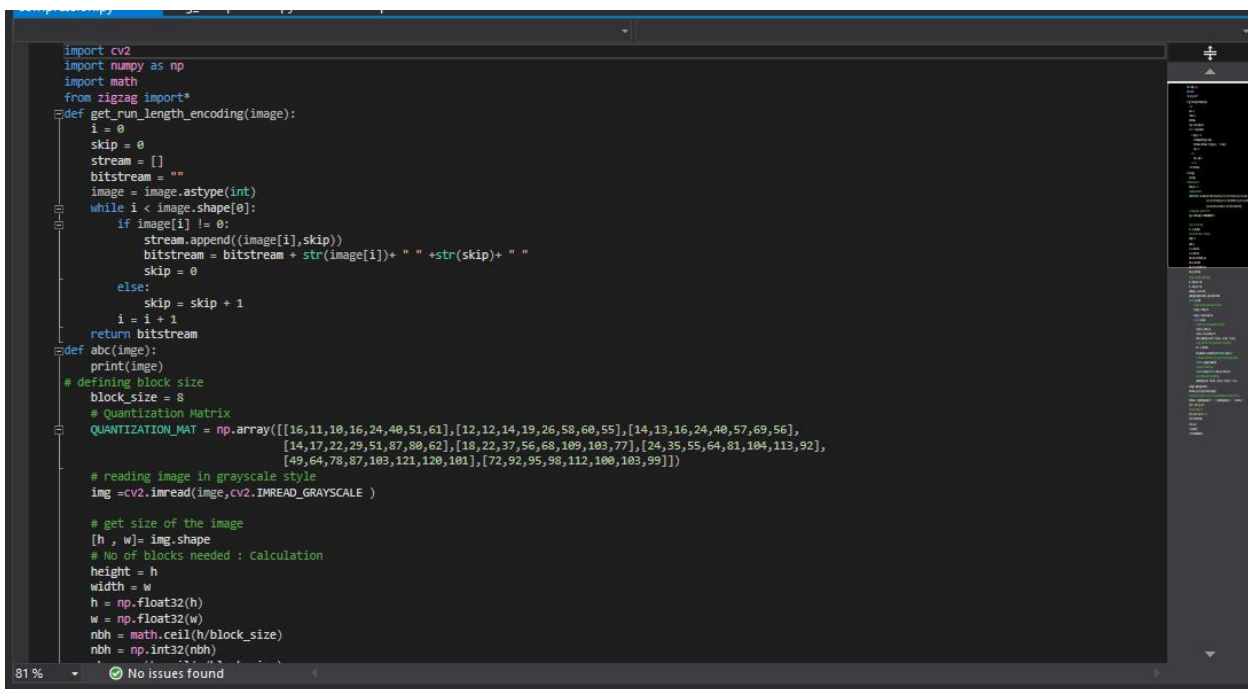
Using RLE compression, the compressed file takes 10 bytes and could look like this:

A*8BCD*4EF

As you can see, repetitive strings of data are replaced by a control character (*) followed by number of repeated characters and repetitive character itself. The control character is not fixed, it can differ from implementation to implementation.

If the control character itself appears in the file then one extra character is coded. Also you can see, RLE encoding is only effective if there are sequences of 4 or more repeating characters because three characters are used to conduct RLE so coding two repeating characters would even lead to an increase in file size. It is important to know that there are many different run-length encoding schemes. The above example has just been used to demonstrate the basic principle of RLE encoding. Sometimes the implementation of RLE is adapted to the type of data that is being compressed.

Below snapshots shows the implementation of all three Algorithms to efficiently compress image.



```
import cv2
import numpy as np
import math
import zigzag
from zigzag import*

def get_run_length_encoding(image):
    i = 0
    skip = 0
    stream = []
    bitstream = ""
    image = image.astype(int)
    while i < image.shape[0]:
        if image[i] != 0:
            stream.append((image[i],skip))
            bitstream = bitstream + str(image[i]) + " " + str(skip) + " "
            skip = 0
        else:
            skip = skip + 1
            i = i + 1
    return bitstream

def abc(image):
    print(image)
    # defining block size
    block_size = 8
    # Quantization Matrix
    QUANTIZATION_MAT = np.array([[16,11,10,16,24,40,51,61],[12,12,14,19,26,58,60,55],[14,13,16,24,40,57,69,56],
    [14,17,22,29,51,87,80,62],[18,22,37,56,68,109,103,77],[24,35,55,64,81,104,113,92],
    [49,64,78,87,103,121,120,101],[72,92,95,98,112,100,103,99]])

    # reading image in grayscale style
    img =cv2.imread(image,cv2.IMREAD_GRAYSCALE )

    # get size of the image
    [h , w]= img.shape
    # No of blocks needed : calculation
    height = h
    width = w
    h = np.float32(h)
    w = np.float32(w)
    nbh = math.ceil(h/block_size)
    nbh = np.int32(nbh)
```

Fig 3.3.1

```

w = np.float32(w)
nbh = math.ceil(h/block_size)
nbh = np.int32(nbh)
nbw = math.ceil(w/block_size)
nbw = np.int32(nbw)
# height and width of padded image
H = block_size * nbh
W = block_size * nbw
padded_img = np.zeros((H,W))
padded_img[0:height,0:width] = img[0:height,0:width]
for i in range(nbh):
    # Compute start and end row index of the block
    row_ind_1 = i*block_size
    row_ind_2 = row_ind_1+block_size
    for j in range(nbw):
        # Compute start & end column index of the block
        col_ind_1 = j*block_size
        col_ind_2 = col_ind_1+block_size
        block = padded_img[row_ind_1 : row_ind_2, col_ind_1 : col_ind_2]
        # apply 2D discrete cosine transform to the selected block
        DCT = cv2.dct(block)
        DCT_normalized = np.divide(DCT,QUANTIZATION_MAT).astype(int)
        # reorder DCT coefficients in zig zag order by calling zigzag function
        reordered = zigzag(DCT_normalized)
        # reshape the reordered array
        reshaped = np.reshape(reordered, (block_size, block_size))
        # copy reshaped matrix into padded_img
        padded_img[row_ind_1 : row_ind_2, col_ind_1 : col_ind_2] = reshaped
arranged = padded_img.flatten()
bitstream = get_run_length_encoding(arranged)
# Two terms are assigned for size as well, semicolon denotes end of image to receiver
bitstream = str(padded_img.shape[0]) + " " + str(padded_img.shape[1]) + " " + bitstream + ";"
print(" compressing done")
# Written to image.txt
file1 = open("image.txt","w+")
file1.write(bitstream)
file1.close()
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Fig 3.3.2

In the above fig, I've imported few modules namely cv2, numpy, math, zigzag. There are two functions in compression .py namely

1. get_run_length_encoding(image)
2. abc(image)

Both accept parameter as image. In abc image is broken into small blocks of 8*8 matrix using quantization then the image is padded into those blocks by determining no of blocks required and compute start and end row index of each block and apply 2D DCT on each block one by one which results in a DCT normalized matrix which is then passed to zigzag function to to reorder DCT coefficient . this image matrix is passed to get_run_length_encoding(image) encoding which returns the image as compressed image in text form and saved ad image.txt in current directory.

3.4 Zig-zag & Inverse zig-zag

Zig-zag provides functions for identifying the peaks and valleys of a time series. Additionally, it provides a function for computing the maximum draw down.After the quantization step of

algorithm, the nature of the quantized DCT coefficients and the random occurrence of zeros in the high frequency coefficients lead to the further need of the compression with the use of the lossless encoding. DC and AC coefficients are treated in different manner during the zigzag step. Due to the strong spatial correlation between the DC coefficients of consecutive 8x8 blocks, they are encoded as the difference between them and their counterparts in the previous blocks. This step takes advantage from the fact that DC coefficients represent a significant fraction of total image energy.

All 63 AC coefficients are scanned by traversing the quantized 8x8 array in a zigzag fashion as shown in the figure below. This zigzag scan traverse through the DCT coefficients in the increasing order of their spatial frequency. Scanning in this order results in a bit stream that contains a large number of trailing zeros as there is a high probability of the high frequency AC coefficients being zeros after quantization. Thus, the resulting bit stream will have large numbers of consecutive zeros or “Runs of Zeros” which makes it suitable for Run Length Encoding. Since more than 60% of the DCT coefficients are zero after quantization for many real life images, Run Length Encoding gives an opportunity of achieving very high compression ratio.

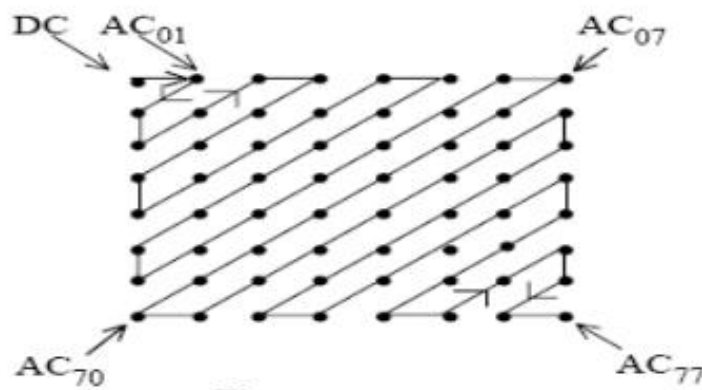


Fig 3.3.1

```

import numpy as np

def zigzag(input):
    #initializing the variables
    h = 0
    v = 0
    vmin = 0
    hmin = 0
    vmax = input.shape[0]
    hmax = input.shape[1]
    i = 0
    output = np.zeros((vmax * hmax))
    while ((v < vmax) and (h < hmax)):
        if ((h + v) % 2) == 0:
            # going up
            if (v == vmin):
                output[i] = input[v, h]
                if (h == hmax):
                    v = v + 1
                else:
                    h = h + 1
                i = i + 1
            elif ((h == hmax - 1) and (v < vmax)):
                # if we got to the last column
                output[i] = input[v, h]
                v = v + 1
                i = i + 1
            elif ((v > vmin) and (h < hmax - 1)):
                # all other cases
                output[i] = input[v, h]
                v = v - 1
                h = h + 1
                i = i + 1
            else:
                # going down
                if ((v == vmax - 1) and (h < hmax - 1)):
                    # if we got to the last line
                    output[i] = input[v, h]
                    h = h + 1
                    i = i + 1
                elif (h == hmin):
                    # if we got to the first column
                    output[i] = input[v, h]
                    if (v == vmax - 1):
                        v = v + 1
                    else:
                        v = v - 1
                    i = i + 1
                else:
                    output[i] = input[v, h]
                    v = v + 1
                    i = i + 1
        else:
            # going down
            if ((v == vmax - 1) and (h < hmax - 1)):
                # if we got to the last line
                output[i] = input[v, h]
                h = h + 1
                i = i + 1
            elif (h == hmin):
                # if we got to the first column
                output[i] = input[v, h]
                if (v == vmax - 1):
                    v = v + 1
                else:
                    v = v - 1
                i = i + 1
            else:
                output[i] = input[v, h]
                v = v + 1
                i = i + 1
        if (v == vmax - 1 and h == hmax - 1):
            break
    return output

```

Fig 3.3.2

```

    if (v == vmax - 1):
        h = h + 1
    else:
        v = v + 1
        i = i + 1
    elif ((v < vmax - 1) and (h > hmin)):
        # all other cases
        output[i] = input[v, h]
        v = v + 1
        h = h - 1
        i = i + 1
    if ((v == vmax - 1) and (h == hmax - 1)):
        # bottom right element
        output[i] = input[v, h]
        break
    return output

def inverse_zigzag(input, vmax, hmax):
    #initializing the variables
    h = 0
    v = 0
    vmin = 0
    hmin = 0
    output = np.zeros((vmax, hmax))
    i = 0
    while ((v < vmax) and (h < hmax)):
        print('v:', v, 'h:', h, 'i:', i)
        if ((h + v) % 2) == 0:
            # going up
            if (v == vmin):
                output[v, h] = input[i]
                if (h == hmax):
                    v = v + 1
                else:
                    h = h + 1
                i = i + 1
            elif ((h == hmax - 1) and (v < vmax)):
                # if we got to the last column
                output[v, h] = input[i]
                v = v + 1
                i = i + 1
            elif ((v > vmin) and (h < hmax - 1)):
                # all other cases
                output[v, h] = input[i]
                v = v - 1
                h = h + 1
                i = i + 1
            else:
                # going down
                if ((v == vmax - 1) and (h < hmax - 1)):
                    # if we got to the last line
                    output[v, h] = input[i]
                    h = h + 1
                    i = i + 1
                elif (h == hmin):
                    # if we got to the first column
                    output[v, h] = input[i]
                    if (v == vmax - 1):
                        v = v + 1
                    else:
                        v = v - 1
                    i = i + 1
                else:
                    output[v, h] = input[i]
                    v = v + 1
                    i = i + 1
        else:
            # going down
            if ((v == vmax - 1) and (h < hmax - 1)):
                # if we got to the last line
                output[v, h] = input[i]
                h = h + 1
                i = i + 1
            elif (h == hmin):
                # if we got to the first column
                output[v, h] = input[i]
                if (v == vmax - 1):
                    v = v + 1
                else:
                    v = v - 1
                i = i + 1
            else:
                output[v, h] = input[i]
                v = v + 1
                i = i + 1
        if (v == vmax - 1 and h == hmax - 1):
            break
    return output

```

Fig 3.3.3

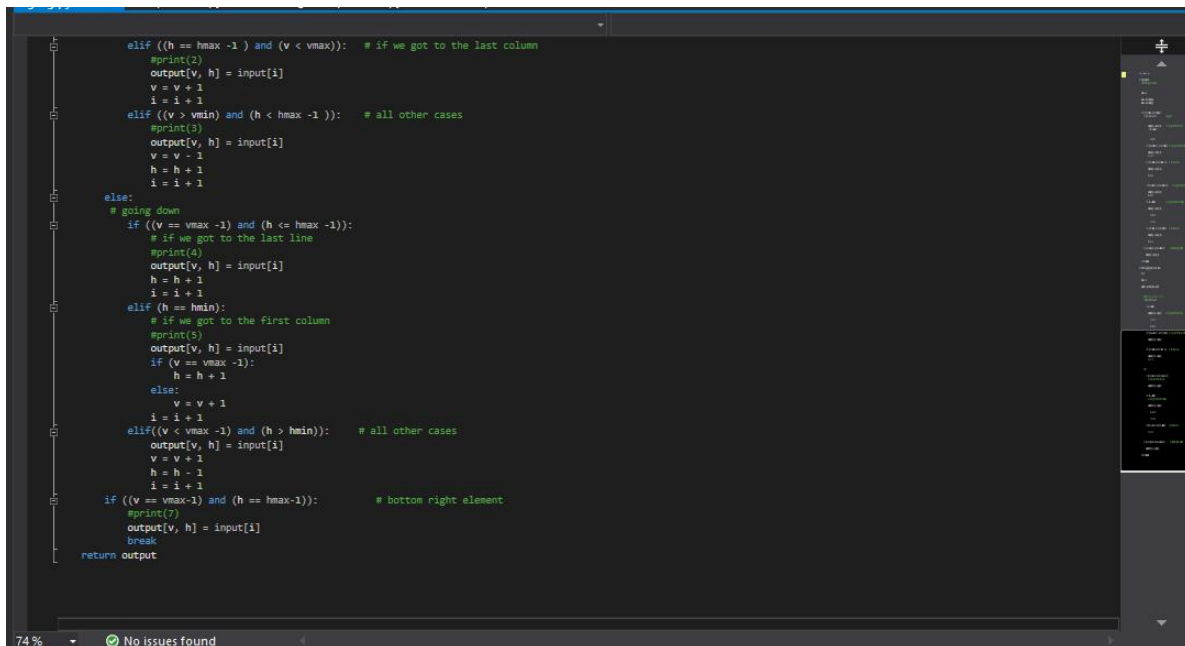


Fig 3.3.4\

In the above Fig's, I've implemented both zig-zag and Inverse zig-zag as functions in a single python program which is being called in this project to identify the peaks and valleys of a time series. Zig-zag function basically, convert image matrix into 1d array bitstream which is encoded and compressed using RLE. Inverse zig-zag to obtain image matrix back from 1d array bitstream

3.5 Decompression

This part of Project include 3 major steps RLE decoding, Inverse zig-zag, Dequantization and Inverse DCT

RLE decoding:

Decoding an RLE-encoded stream of data is actually even easier than encoding it. Like before, you iterate through the data stream one character at a time. If you see a numeric character then you add it to your count, and if you see a non-numeric character then you add count of those characters to your decoding, which is returned to the caller once you iterate through all of the input data.

Inverse zig-zag:

Inverse zig-zag is applied to get back image matrix from bitstream returned by RLE decoding, refer 3.4 for detailed explanation.

Dequantization:

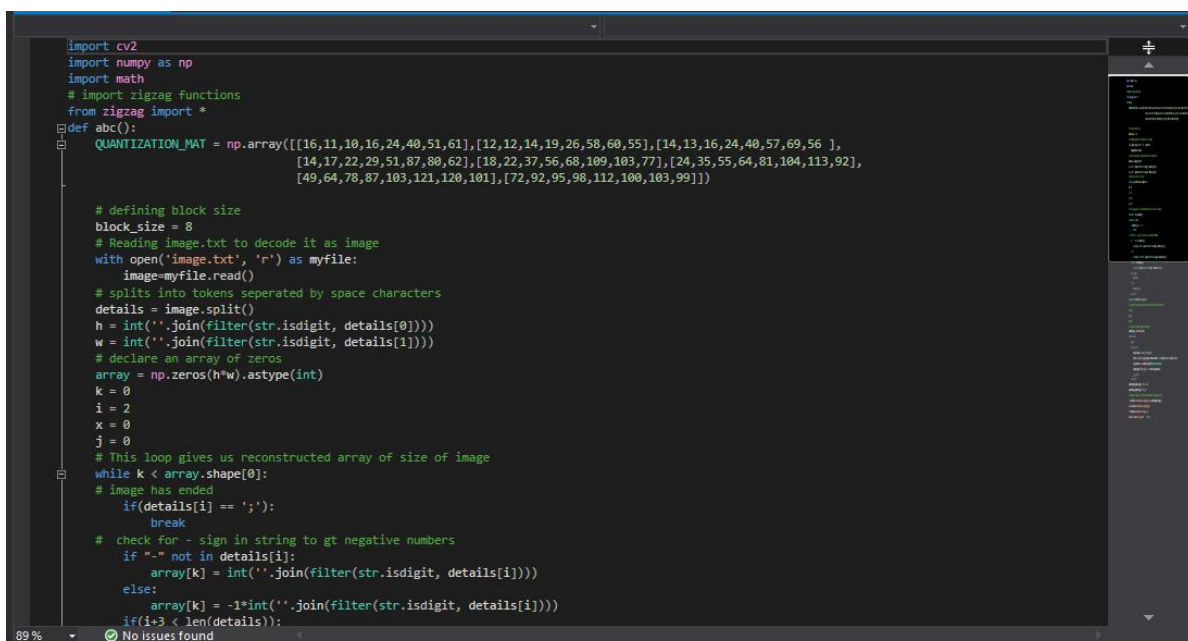
The Dequantization function was also written in this project and was implemented in decompress.py.

as the finer detail of the image is lost due to these truncation's of the high frequency DCT coefficients to zeros. The original values cannot be restored back in the decompression step.

Inverse DCT:

Inverse Discrete Cosine Transform was applied on all 8x8 blocks of the image. The same approach for DCT was used to implement IDCT but the order of the multiplications was reversed. Thus, each 8x8 DCT matrix was first multiplied by the transposed cosine matrix and then by the 24 original cosine transform matrix of the same size. The image was converted back to the spatial domain from the frequency domain after the application of IDCT. The original image was regenerated and successfully

The image was reconstructed after performing the JPEG decompressing steps including RLE decoding, Inverse zigzagging, Dequantization and Inverse DCT. This proved the correctness of my JPEG & PNG compression implementation. The reconstructed image after applying the JPEG decompression is shown in window.



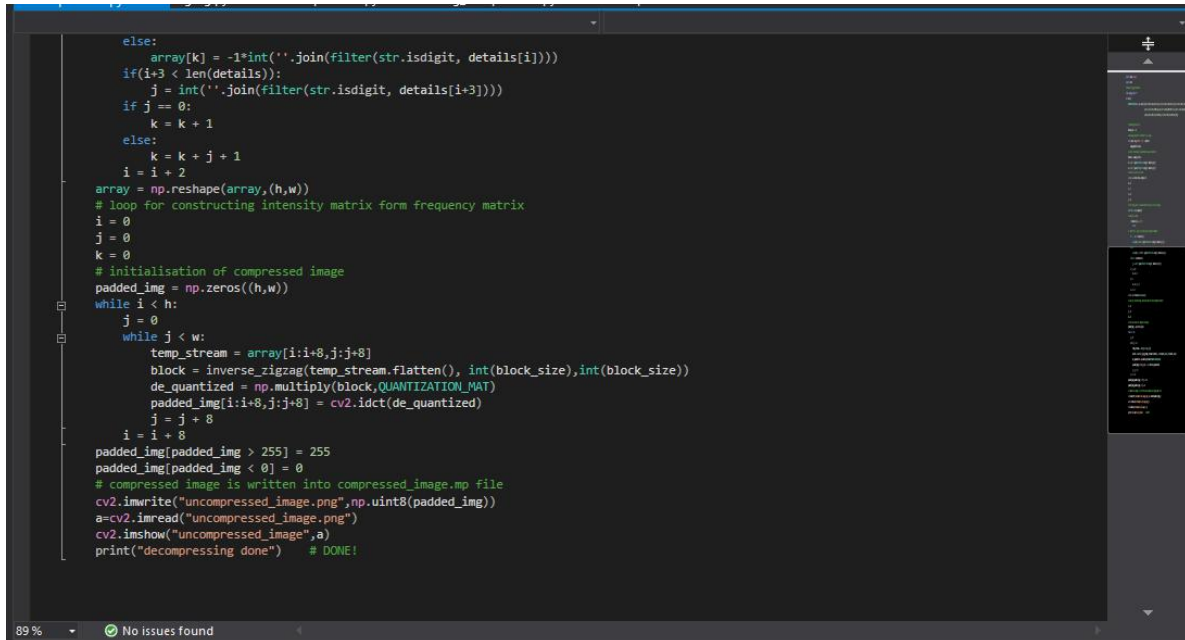
```
import cv2
import numpy as np
import math
# import zigzag functions
from zigzag import *

def abc():
    QUANTIZATION_MAT = np.array([[16,11,10,16,24,40,51,61],[12,12,14,19,26,58,60,55],[14,13,16,24,40,57,69,56 ],
                                  [14,17,22,29,51,87,80,62],[18,22,37,56,68,109,103,77],[24,35,55,64,81,104,113,92],
                                  [49,64,78,87,103,121,120,101],[72,92,95,98,112,100,103,99]])

    # defining block size
    block_size = 8
    # Reading image.txt to decode it as image
    with open('image.txt', 'r') as myfile:
        image=myfile.read()

    # splits into tokens separated by space characters
    details = image.split()
    h = int(''.join(filter(str.isdigit, details[0])))
    w = int(''.join(filter(str.isdigit, details[1])))
    # declare an array of zeros
    array = np.zeros(h*w).astype(int)
    k = 0
    i = 2
    x = 0
    j = 0
    # This loop gives us reconstructed array of size of image
    while k < array.shape[0]:
        # image has ended
        if(details[i] == ';'):
            break
        # check for - sign in string to get negative numbers
        if "-" not in details[i]:
            array[k] = int(''.join(filter(str.isdigit, details[i])))
        else:
            array[k] = -1*int(''.join(filter(str.isdigit, details[i])))
        if(i-3 < len(details)):
            i+=3
        k+=1
```


Fig 3.4.1



```
else:
    array[k] = -1*int(''.join(filter(str.isdigit, details[i])))
if(i+3 < len(details)):
    j = int(''.join(filter(str.isdigit, details[i+3])))
    if j == 0:
        k = k + 1
    else:
        k = k + j + 1
    i = i + 2
array = np.reshape(array,(h,w))
# loop for constructing intensity matrix form frequency matrix
i = 0
j = 0
k = 0
# initialisation of compressed image
padded_img = np.zeros((h,w))
while i < h:
    j = 0
    while j < w:
        temp_stream = array[i:i+8,j:j+8]
        block = inverse_zigzag(temp_stream.flatten(), int(block_size),int(block_size))
        de_quantized = np.multiply(block,QUANTIZATION_MAT)
        padded_img[i:i+8,j:j+8] = cv2.idct(de_quantized)
        j = j + 8
    i = i + 8
padded_img[padded_img > 255] = 255
padded_img[padded_img < 0] = 0
# compressed image is written into compressed_image.mp file
cv2.imwrite("uncompressed_image.png",np.uint8(padded_img))
a=cv2.imread("uncompressed_image.png")
cv2.imshow("uncompressed_image",a)
print("decompressing done") # DONE!
```

Fig 3.4.2

CHAPTER 4
RESULTS AND SNAPSHOTS

CHAPTER 4

RESULTS AND SNAPSHOTS

Snapshots

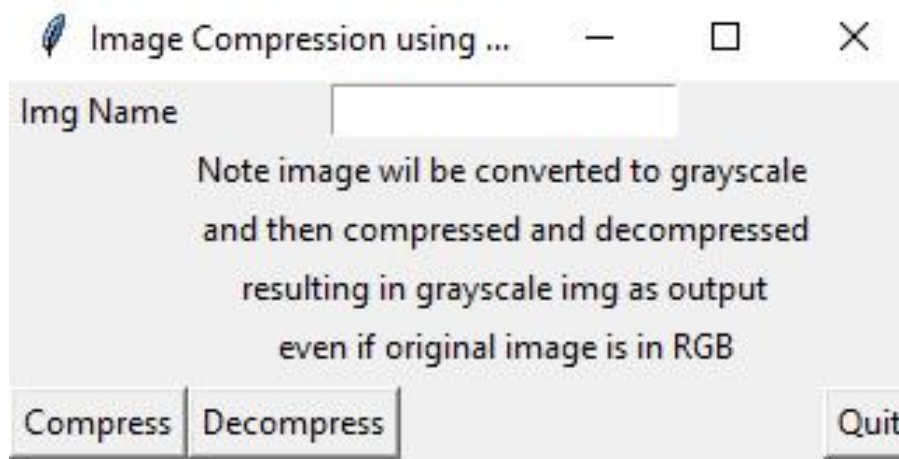


Fig 4.1.1

This window appears as soon as the user runs the program, which contains a text field to enter the name of the image and Two Buttons to Compress and Decompress the image and a Button to quit the application

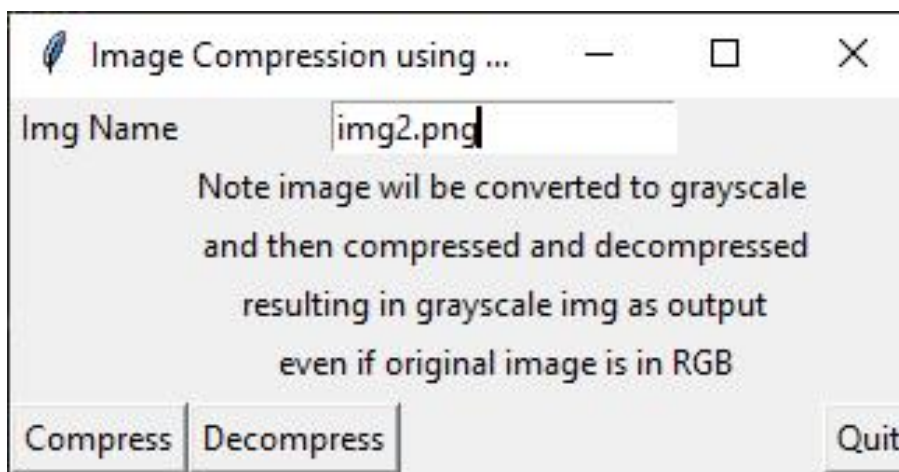


Fig 4.1.2

The user can type in the image name with its extension in text field and press on compress, then a window opens displaying the selected image after viewing it user can close the window containing the image.



Fig 4.1.3
(Img2.png)

A window containing the original gets displayed as shown in fig 4.3 the user can close it in mean time the image is being compressed.

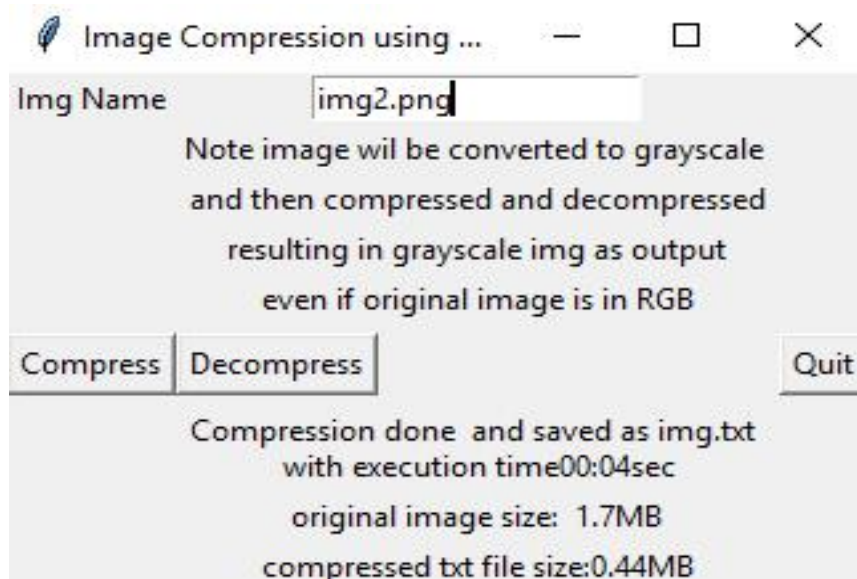


Fig 4.4

In this window user is displayed with all information about the compression performed on the image with its original size, compressed size and time taken to compress. The user can see that compressed image is stored in directory with name image.txt which will be used to decompress



Fig 4.1.5
(Uncompressed_img.png)

As soon as user tap's on Decompress Button the decompressed image is displayed on the screen user can close it pressing '0' in keyboard or simply closing it.

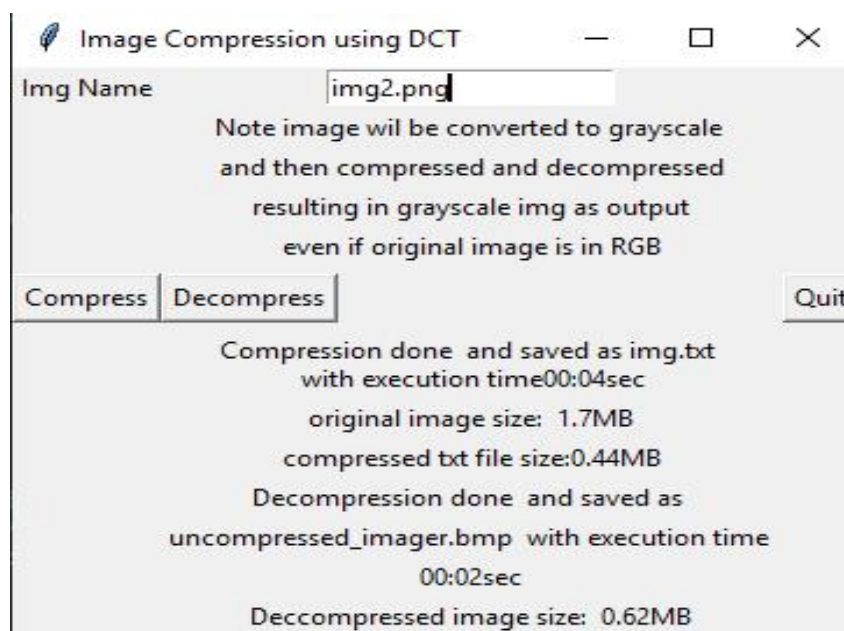


Fig 4.1.6

In this window user is displayed decompression statistics like time taken, image size. The user can observe the decompressed image saved as uncompressed_img.png in current directory

Observation

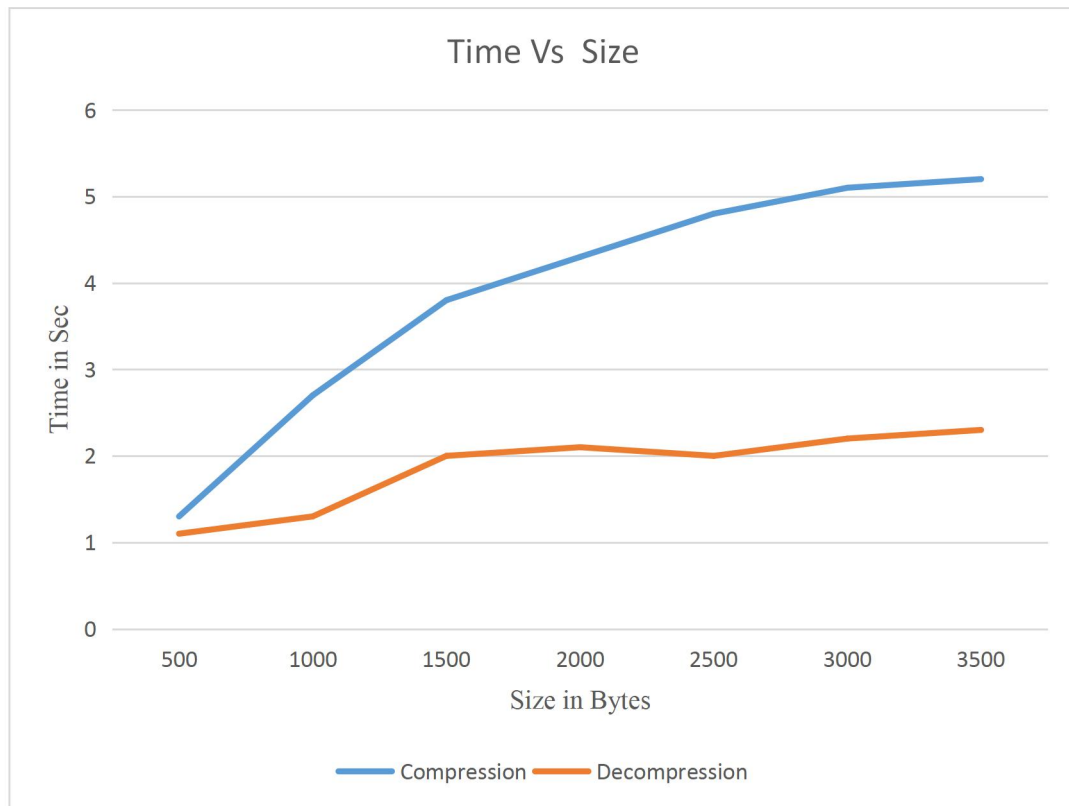


Fig 4.1.7

It has been observed that the time taken for compression increases with the size of the image almost linearly at the beginning, and the curve softens up as the size increases. It is also observed that sometimes the time taken is too long for images containing many colors or shades of gray in a Grayscale image. As I've used Quantization, DCT, RLE for compression, the compressed image loses its quality due to quantization. As the image is converted to Grayscale and then compression is performed, this is a drawback that the user won't get a colored image after compression or decompression. But the time taken for decompression remains almost consistent for the tests performed on this project. The above graph supports the observation done; it was plotted using 3 to 4 PNG images of each size, and the average time consumed was taken. It may change a little for JPEG and different images.

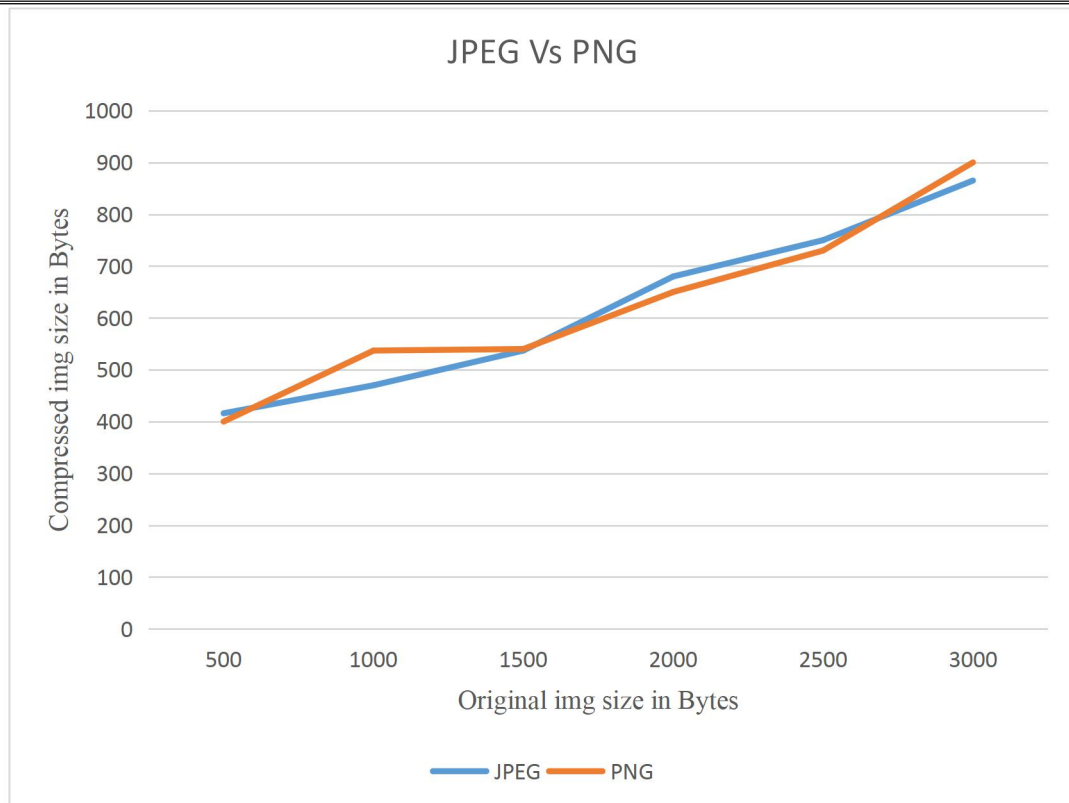


Fig 4.1.8

It has been observed that this algorithm does a slightly better job with PNG images compared to JPEG images, as we can see in the above graph plot taking 3 to 4 images of each type. PNG images' compression rate is better than JPEG most of the times. It may be because of the alpha channel in PNG images used for test cases; sometimes JPEG compressed images are greater than the original image. This algorithm is not too efficient compared to other image compression algorithms as it has its own drawbacks, but it quite gets the job done most of the time. PNG images with more shades of color and having an alpha channel (transparent background) result in distorted images.

Results

Here, we can see a few images on which the above observations made for each image below lead to different outputs which will be discussed briefly.

JPEG IMAGES:

Fig 4.2.1 (687kb)



Fig 4.2.2(584kb)

This ones example for ideal image compression original image, compression rate achieved was 85%



2.

Fig 4.2.3(314kb)



Fig 4.2.4(177kb)

Images made of less colors give better results as in above image the original image was black and white only resulting in 56% compression rate



3.

Fig 4.2.5(1.4Mb)



Fig 4.2.6(985kb)

Images covered majorly one color in this particular example whit background increases compression rate to 69% even though image has many shades of color



4.

Fig 4.2.7(313kb)



Fig 4.2.8(480kb)

In this above example the file size has been increased by 50% instead of getting reduced its due to original image has a gradient background not plain white hence so many color shades to compress and hence algorithm fails to perform better in this kind of situations

PNG IMAGES



1.

Fig 4.2.9(345kb)



Fig 4.2.10(177kb)

This is example for ideal conversion, with compression rate of 51% it performed well as there were only 3 major color transparent, Grey, orange



2.

Fig 4.2.11(689kb)



Fig 4.2.12(303kb)

This example illustrates the major drawback in this algorithm that it performs really bad with gradients combined with alpha channel(transparent background) even though it has good compression rate of 52% the end image is not acceptable



3.

Fig 4.2.13(319kb)



Fig 4.2.14(243kb)

This example illustrates the major drawback in this algorithm that it performs really bad with gradients combined with alpha channel(transparent background) the end image is not acceptable for any application as it has lost most of the details



4 .

Fig 4.2.15(846kb)



Fig 4.2.16(487kb)

This is a exception couldn't understand yet hoe the result is good compared to above two examples even when original image had gradient and was on alpha channel, this has a compression rate of 57%.

CONCLUSION & REFERENCES

Conclusion

The DCT image compression algorithm was studied in much detail and the various steps of the JPEG & PNG including DCT , quantization, Zigzag scanning and run length encoding were implemented to achieve the above results. algorithm was verified for correctness by successfully implementing the Inverse functions such as IDCT, Dequantization, Inverse Zigzag scanning and run length decoding. This project gave me the thorough knowledge and understanding On DCT, Quantization and RLE algorithms. The possible solution to improve the quality of the reconstructed images by means of filtering the values of the DCT co-efficient's after the quantization was proposed and implemented.

References

1. <https://ieeexplore.ieee.org/document/1042372>
2. <https://youtu.be/tW3Hc0Wrgl0>
3. <https://www.geeksforgeeks.org/discrete-cosine-transform-algorithm-program/>
4. <http://csus-dspace.calstate.edu/bitstream/handle/10211.9/742/PDF.pdf?s>
5. <https://books.google.co.in/books?id=fWviBQAAQBAJ&printsec=frontcover&dq=dct+algorithm&hl=en&sa=X&ved=0ahUKEwiYnMiD3uPpAhWlxzgGHY9NAocQ6AEIMDAB#v=onepage&q=dct%20algorithm&f=false>
6. <https://books.google.co.in/books?id=MKhbLPHRb78C&pg=PA212&dq=inverse+zig+zag&hl=en&sa=X&ved=0ahUKEwiLhbud3uPpAhXLYDgGHZvzD3UQ6AEIKDAA#v=onepage&q=inverse%20zig%20zag&f=false>