Module Code: CS3IA16

Assignment report Title: Technical Report

Date (when the work completed): 06/12/22

Student Number: 29015642, 29021376

Actual hrs spent for the assignment: 6 hours each

Assignment evaluation (3 key points):

- Getting some libraries to work in python was a pain as we had to reinstall python in an earlier version
- Learning the jpeg algorithm was a blast for both of us
- James' advice for the creation of the algorithm was very helpful

Contents

Abstract

The purpose of image compression is to reduce the file-size of the image by getting rid of redundant data. Redundant data is defined by certain details in an image that if they were to be removed the human eye wouldn’t be able to notice the difference. Applying image compression allows for files to occupy less storage on a device as well as require less bandwidth when sending data (in this context images) over a network.[1]

The purpose of this report is to highlight the effects of lossy and lossless compression applied to images.

Introduction

Image compression is one of the most important concepts in the modern era. Constantly, images and videos are being sent around the world. The problem that occurs is that all these media sources require a large amount of data to be transmitted from one device to another. Most devices are restricted with a limit to the amount of data they can receive and send at a given moment. A solution to this problem is compression. Compression is a term used to describe the act of pressing something into a smaller space [2]. In terms of computer systems, compression can reduce file sizes. Compression falls into two categories, lossy and lossless. Lossy compression is when the data or information is removed permanently and cannot be recovered, resulting in a reduction in file sizes and quality. It is most used for image and video compression. Lossless compression, on the other hand, is a form of compression that reduces the size of the file without removing the data permanently. With lossless, it is possible to reconstruct the original file perfectly after decompression. With the combination of these, images and video file sizes can be greatly reduced when being transmitted over a network. This is when JPEG compression is utilised and most applied.

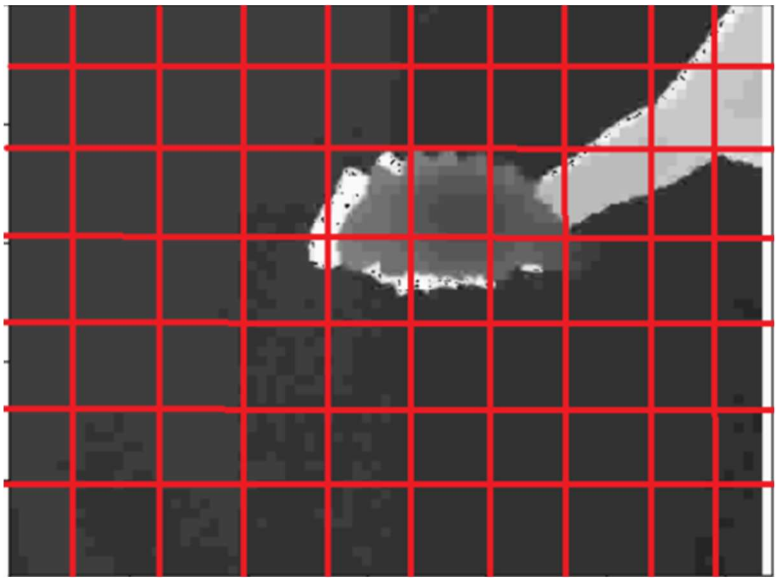
Within this report, we will be discussing and implementing the methods and algorithms of JPEG to compress and decompress a variety of different images. Metrics such as file size, MSE and compression ratio will be used as statistics to visualise the effects of the JPEG algorithm. This will be made possible using a user interface at which an image can be loaded, compressed and then decompressed into and out of memory.

Methodology

Theory

JPEG algorithm is used to compress images by getting rid of redundant data. The algorithm works by first dividing the RGB of the image data into luminance and colour components. The components are then divided into pixel blocks, with the block size normally being 8x8. The reason for that is because a block of size 4x4 would create poor quality (we define quality by calculating how different the decompressed images are from their original form) in the image when decompressed, while anything above a size 8x8 would create a higher quality image but at the cost of performance issues [3]. Figure 1 represents how the image is divided into pixel blocks

Figure1 Representing how pixel blocks are applied to an image

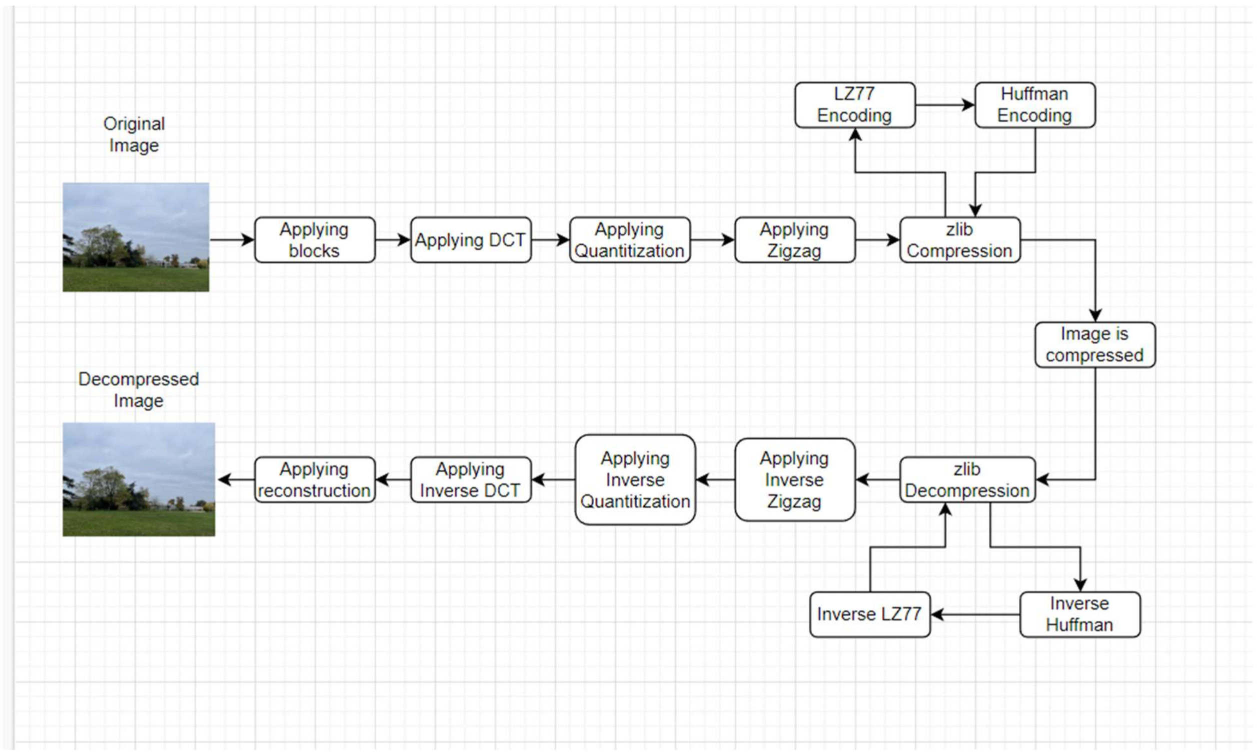


Each pixel block is then transformed using discrete cosine transformation (DCT). The image data in each block is represented using pixel numbers and values, after DCT the image data is represented as coefficients of the spatial domain frequencies for vertical and horizontal orientation. The next step is quantization. Quantization works by having each DCT element divided by the respective element inside a quantization table. Each application can supply its own values inside the quantization table as there is no standard, allowing control of the loss-compression trade-off depending on the needs of the application. This is done to get rid of redundant data or image data that wouldn't be noticed by the human eye if it was removed. The elements are then linearized using the zigzag method, and then run length and Huffman encoding are applied respectively. Both are types of lossless data compression algorithms.[4][5]
For decompression of the image, the explained method above is applied in reverse, producing an image with less file size but reduced quality. The reduced quality is dependent on different factors such as the size of the pixel blocks and the values inside the quantization table, however, under normal conditions, the reduction in quality shouldn't be easily recognizable by the human eye.

For this project, we have followed the format of the JPEG algorithm but instead of run-length encoding, we have used zlib, which is a software library written in C, used for data compression which utilizes the DEFLATE data compression format [6]. The DEFLATE format uses a combination of Huffman encoding and LZ77, which is another lossless data compression algorithm [7][8]. The use of the zlib software library has proven to be more efficient on our machines than implementing run-length and Huffman encoding separately.

Implementation details:

Figure 2. Shows the design of our compression and decompression algorithm, inspired by the JPEG algorithm from [4]



The design above has been implemented with PyCharm: python 3.9, using the following python libraries: *openCV; Scipy; Os; Numpy; Zlib; Matplotlib; warnings*

Functions

Function name	Purpose
encoding_dct_matrix()	This function applies the block padding of the image, applies DCT, quantization and zigzag then returns an encoded value.
compress()	Call all the functions needed to compress and decompress the image and display the image and all the statistics of the image.
CompressMenu()	Responsible for getting the user input regarding block size and quality factor and passing them to the function compress()
get_file_size()	Takes the images as a parameter and returns the file size in MB
menu()	Contains all the UI elements needed to allow the user to select, compress and display details about images.
decode_dct()	Does the inverse of all the methods that the *encoding_dct_matrix* does return the decompressed image
encoding_compressed_image()	This function applies lossless compression using the zlib compression software library.
decoding_compressed_image()	This function applies the inverse of the compression algorithms used by the zlib compression software library.
zigzag()	This applies the zigzag method after the image has been quantized.

Design of the Data Structures

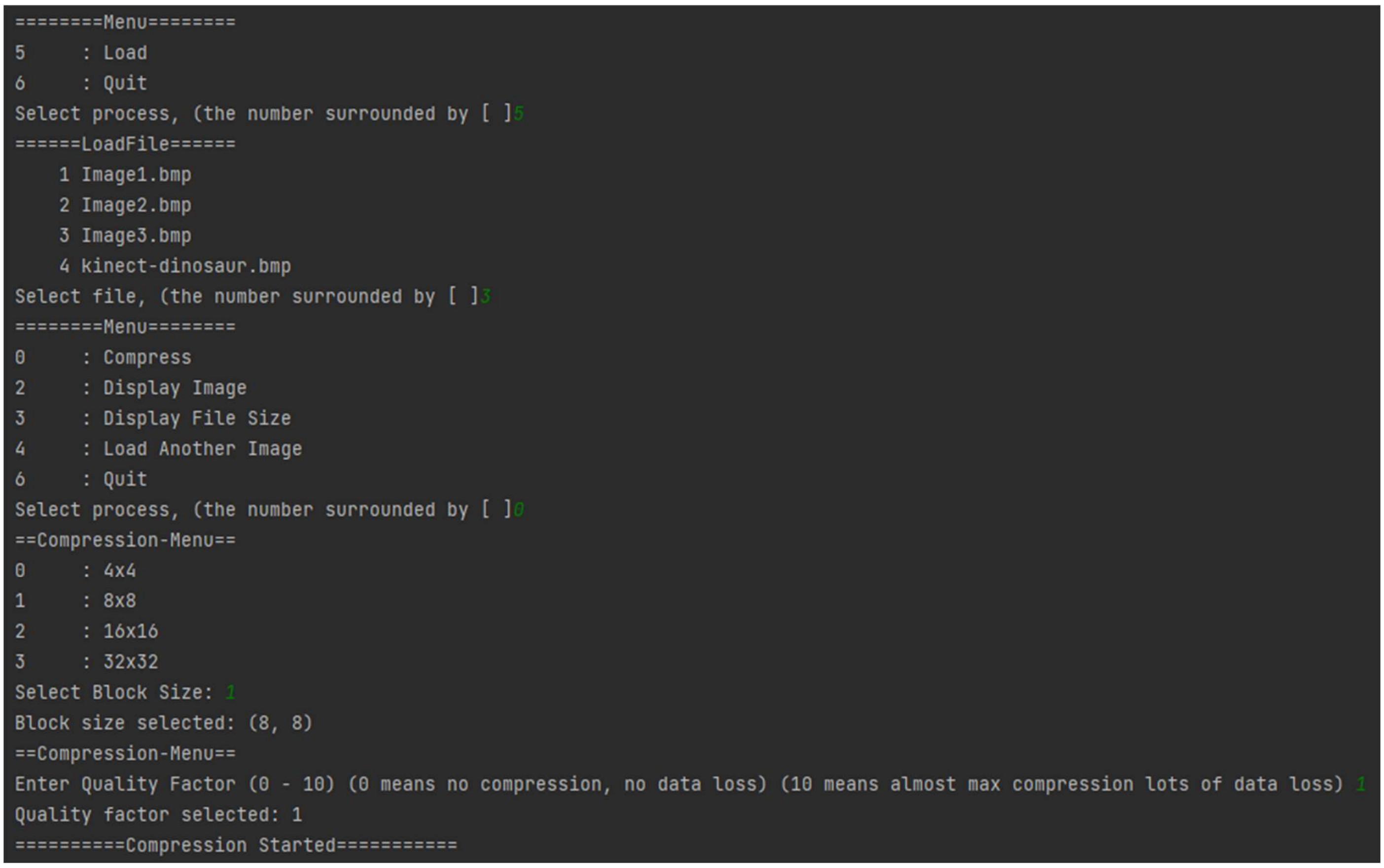
Used numpy arrays to hold and manipulate the images . Used numpy integers for conversion of images. Dictionaries to hold all the images for user selection and 2D arrays to hold quantisation matrices. All the data structures are present in the code under the *Appendix.*

Results/Testing/Achievements

The program consists of a console-based user interface that allows the user to select an image, display the image, compress/decompress the image and switch between different images.

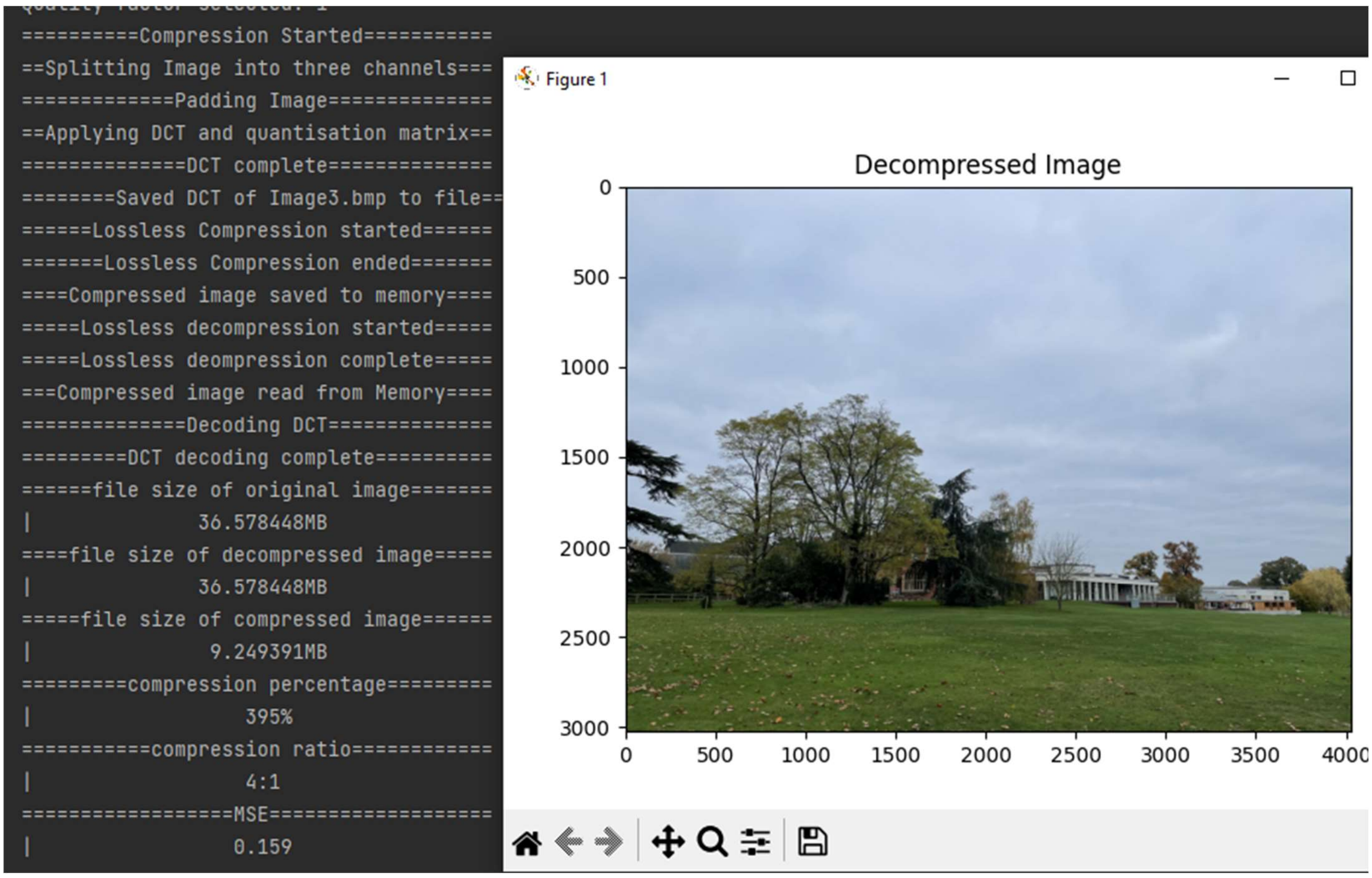
As we are saving and displaying the image as BMP, the file size on the disk does not change. Bmp files contain raw data of the image. The file size of a BMP can only be reduced by resizing the dimensions of the image. This is why the decompressed image that we display has the same file size as the original image. If we were to save it as a .jpeg or .png then we would get a significantly lower file size regardless of the quality. The quality of the produced decompressed image depends on the block size and quant table used in the JPEG algorithm. It was found that if we altered these values, we would get different qualities of the original image.

Figure 3 shows the console-based UI selected image3.bmp, compression mode selected, and block size and quality factor selected.



```
========Menu========
5     : Load
6     : Quit
Select process, (the number surrounded by [ ]5
======LoadFile======
    1 Image1.bmp
    2 Image2.bmp
    3 Image3.bmp
    4 kinect-dinosaur.bmp
Select file, (the number surrounded by [ ]3
========Menu========
0     : Compress
2     : Display Image
3     : Display File Size
4     : Load Another Image
6     : Quit
Select process, (the number surrounded by [ ]0
==Compression-Menu==
0     : 4x4
1     : 8x8
2     : 16x16
3     : 32x32
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 1
Quality factor selected: 1
==========Compression Started==========
```

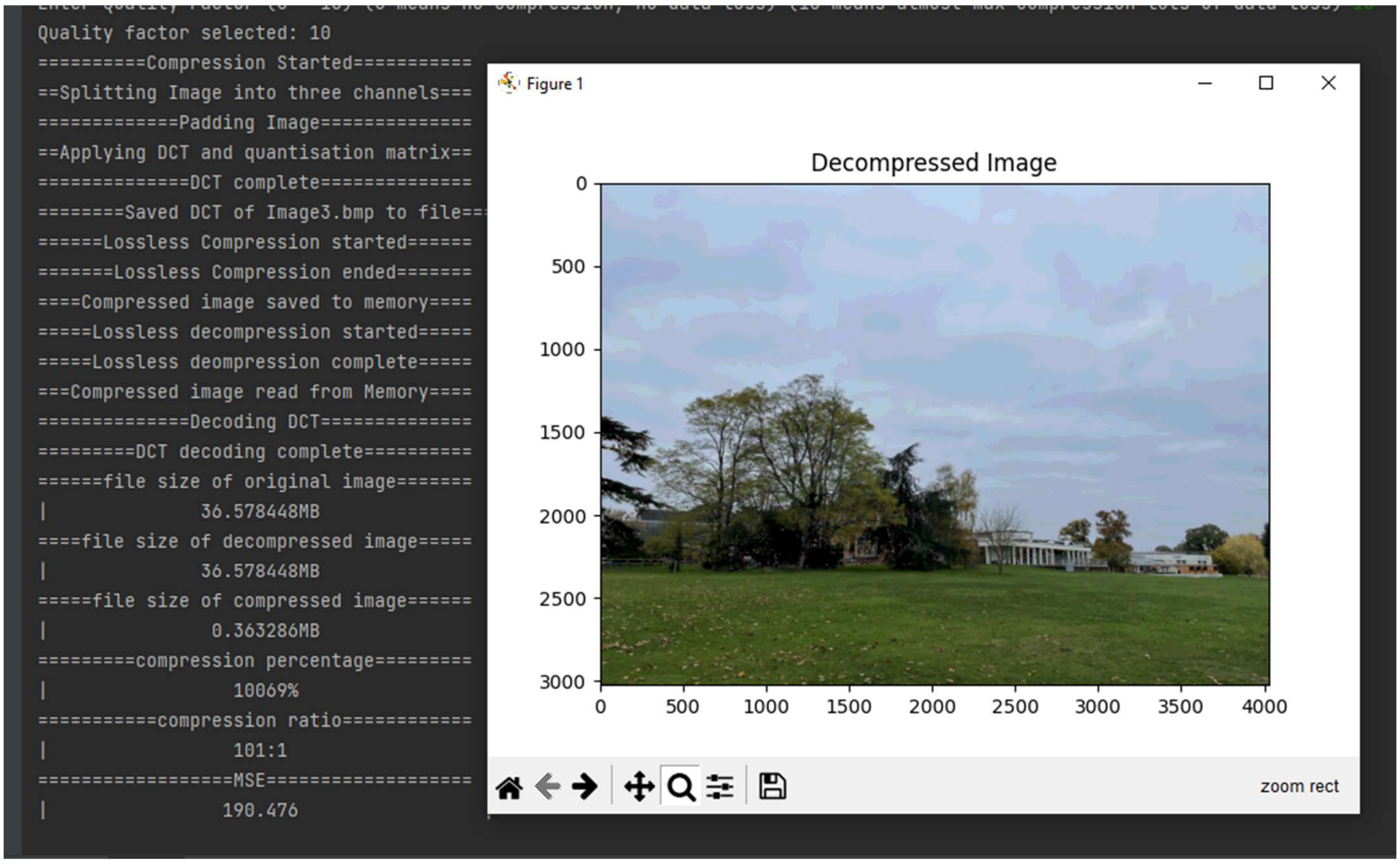
Figure 4 shows the console-based UI displaying metrics of the image after decompression



```
==========Compression Started==========
==Splitting Image into three channels===
=============Padding Image=============
==Applying DCT and quantisation matrix==
==============DCT complete=============
========Saved DCT of Image3.bmp to file==
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image=======
|               36.578448MB
====file size of decompressed image=====
|               36.578448MB
=====file size of compressed image======
|                9.249391MB
=========compression percentage=========
|                   395%
===========compression ratio===========
|                   4:1
==================MSE==================
|                  0.159
```

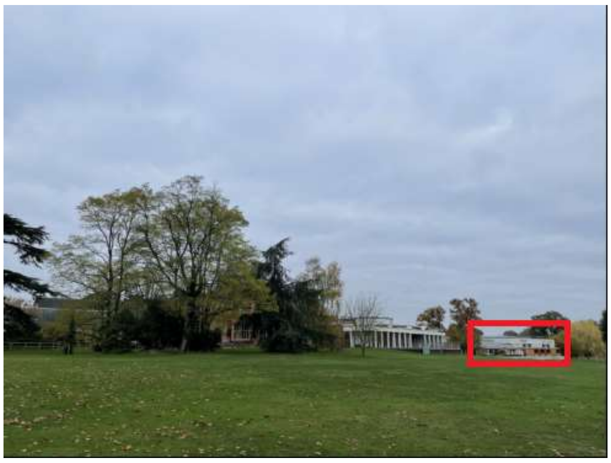
Figure 4 shows the UI displaying when each process of compression/ decompression starts and finishes during execution. The metrics are also displayed, including the MSE, compression ratio, compressed file size and original file size. The MSE is 0.159 which implies that the image is extremely similar to the original image in terms of its shapes and pixel arrangements. A higher MSE implies that the image's quality has changed from the original, it is most likely that in this case, the image's quality will seem much less. We successfully reduced the file size of this image to around 9.24MB in compressed memory from 36.57MB. This gave us a compression ratio of around 4:1 with a compression percentage of 395%. The image displayed is of the decompressed image, which has the same file size as the original image since it is saved as a BMP. If we were to save the image as a .jpeg or .png file, then we would see the exact same image as produced above.

Figure 5 shows metrics for quality factor 10



```
Enter quality factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss)
Quality factor selected: 10
==========Compression Started==========
==Splitting Image into three channels===
=============Padding Image==============
==Applying DCT and quantisation matrix==
==============DCT complete==============
========Saved DCT of Image3.bmp to file==
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT==============
=========DCT decoding complete==========
======file size of original image=======
|              36.578448MB
====file size of decompressed image=====
|              36.578448MB
=====file size of compressed image======
|               0.363286MB
=========compression percentage=========
|                 10069%
===========compression ratio============
|                 101:1
=================MSE====================
|                190.476
```

Figure 6 Segmentation of area of interest in image



When deciding to see where we can detect changes in the image when applying compression, we decided to focus on a specific point in the image. The hollow red box applied to the image in figure (X)

Figure 7 Image3_recompressed_10_8x8.bmp Snipped Section

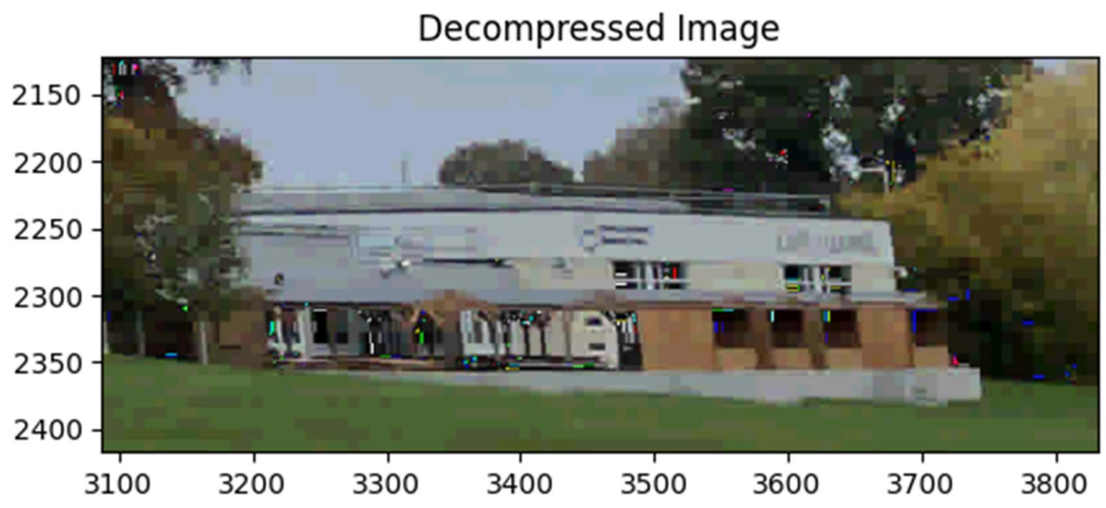


Figure 8. Image3_recompressed_1_8x8.bmp Snipped Section

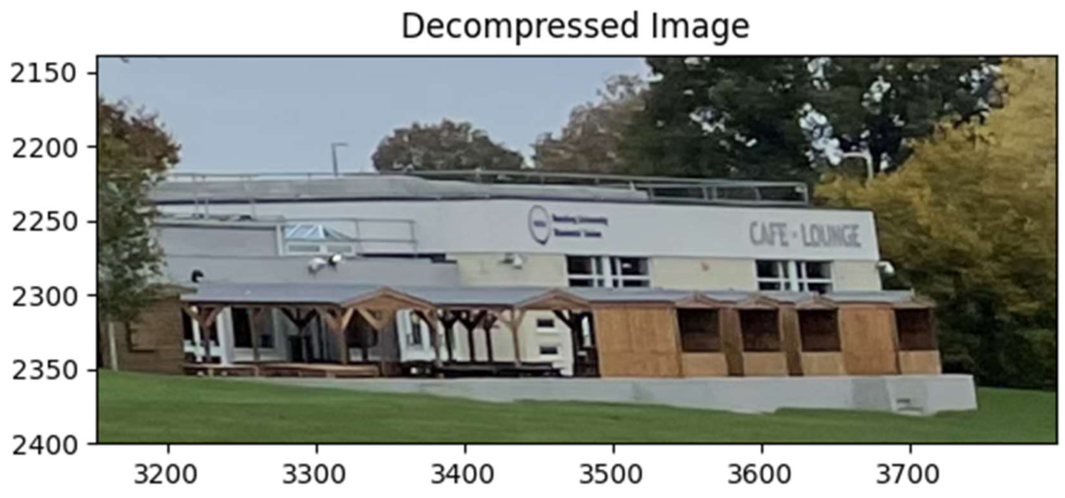


Image	MSE	Compression Ratio	Compression %	File size
Image3_recompressed_1_8x8.bmp	0.159	4:1	395%	9.249391MB
Image3_recompressed_10_8x8.bmp	190.476	101:1	10069%	0.363286MB

Figures 7 and 8 have been compressed using a pixel block size of 8x8 however Figure 7 uses a quality factor of 10 whereas Figure 8 uses a quality factor of 1. The quality factor defines the values inside the quantization table. The higher the quality factor the higher the values inside the table which will cause quantization to get rid of more DCT coefficients, reducing the data of the image. With a higher loss in data means less memory is occupied, however at the cost of having the quality of the image heavily reduced. This is evidenced in the figures above, where the compression with a quality factor of 10 reduces the size of the image by a ratio of 101:1, in comparison to the original size, but at the cost of having a much higher MSE (as shown by the pixel tearing and blur), whereas the compression with a quality factor of 1 achieves the opposite.

Similarly, the same process was applied to the other 3 images, the images for these are in the appendix of this report.

Conclusion

For this coursework we have developed an application for compressing, decompressing images, and displaying their metrics on a console GUI. The compression algorithm used by the application works by applying pixel blocks, DCT, Quantization, Zigzag method, the LZ77 and the Huffman Encoding lossless algorithms from the zlib compression software library to the images, then decompressing them by using the inverse of each method.

We have evidenced how loss-compression trade-off impacts the resultant quality and file size of the compressed images by applying pixel blocks of different sizes and quantization tables of different values. We have shown that higher values in the quantization table result in a bigger file size reduction at the cost of quality by calculating the MSE between compressed and original images.

Based on our testing and design of the compression algorithm, the best result in terms of memory and quality of the images is achieved when we apply a quality factor of 1 and pixel blocks of size 8x8. Any quality factor higher than 1 will result in a higher size reduction, however at the cost of the image being exponentially worse. In theory, a pixel block size higher than 8x8 will increase the quality of the resultant image however the calculation takes a long time, making it hard to measure.

If we could further improve our findings of compression and decompression, we would implement other lossless compression algorithms, such as Arithmetic, Entropy or bit-plane coding. By implementing these we would have been able to compare which algorithm performs faster and more efficiently than the already lossless algorithms we have implemented in our design. This could have resulted in a much greater reduction in file size when our image is compressed.

References/Acknowledgements

[1] WhatIs.com. (n.d.). What is image compression and how does it work? [online] Available at: https://www.techtarget.com/whatis/definition/image-compression#:~:text=Image%20compression%20is%20a%20process.

[2] dictionary.cambridge.org. (n.d.). COMPRESSION | meaning in the Cambridge English Dictionary. [online] Available at: https://dictionary.cambridge.org/dictionary/english/compression.

[3] Stack Overflow. (n.d.). Why JPEG compression processes image by 8x8 blocks? [online] Available at: https://stackoverflow.com/questions/10780425/why-jpeg-compression-processes-image-by-8x8-blocks [Accessed 6 Dec. 2022].

[4] www.image-engineering.de. (n.d.). How does the JPEG compression work? [online] Available at: https://www.image-engineering.de/library/technotes/745-how-does-the-jpeg-compression-work#:~:text=The%20JPEG%20compression%20is%20a.

[5] Al-Ani, Muzhir & Awad, Fouad. (2013). THE JPEG IMAGE COMPRESSION ALGORITHM. International Journal of Advances in Engineering & Technology. 6. 1055-1062. https://www.researchgate.net/publication/268523100_THE_JPEG_IMAGE_COMPRESSION_ALGORITHM

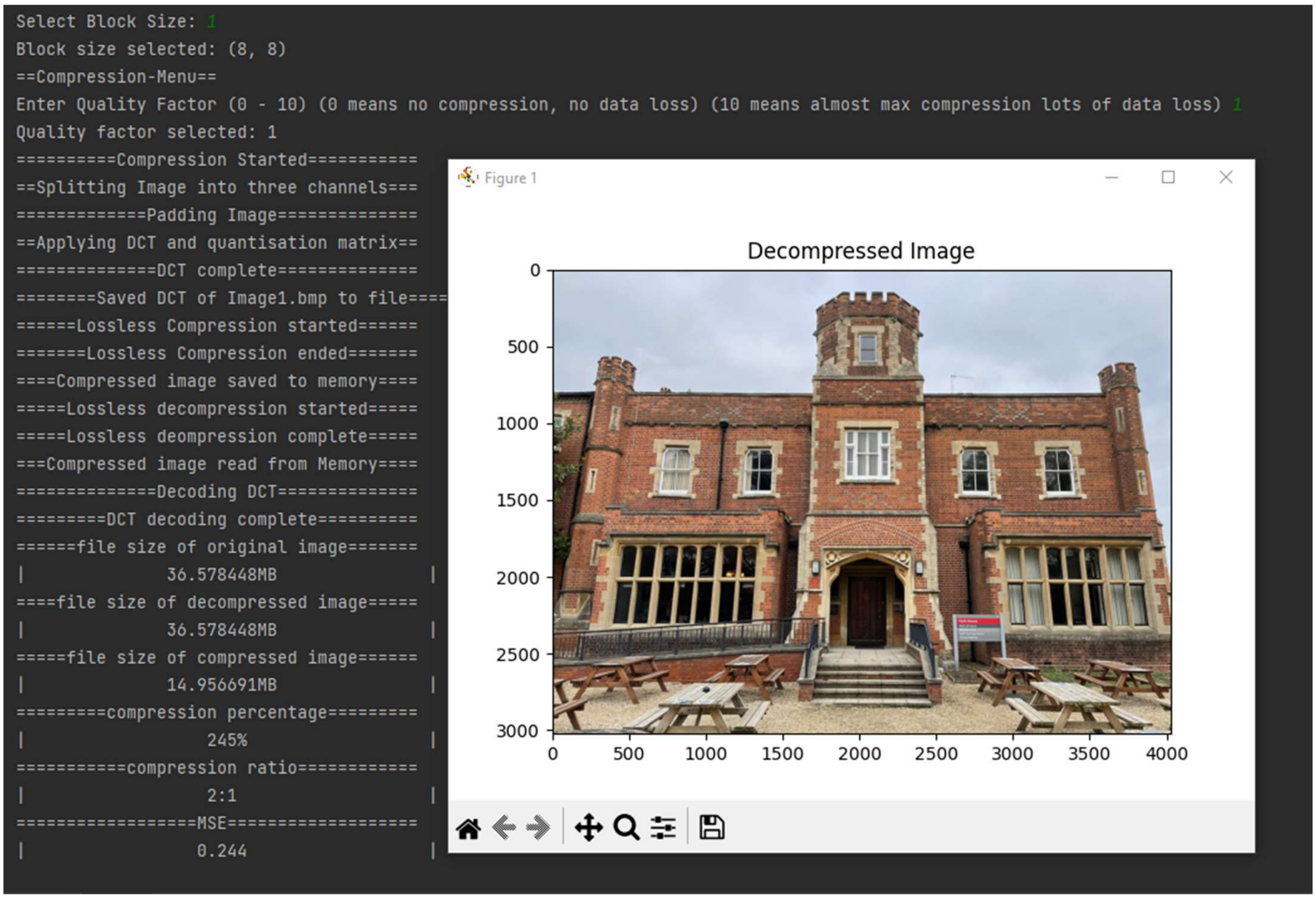
[6] Wikipedia. (2022). zlib. [online] Available at: https://en.wikipedia.org/wiki/Zlib [Accessed 6 Dec. 2022].

[7] Wikipedia Contributors (2019). DEFLATE. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/DEFLATE.

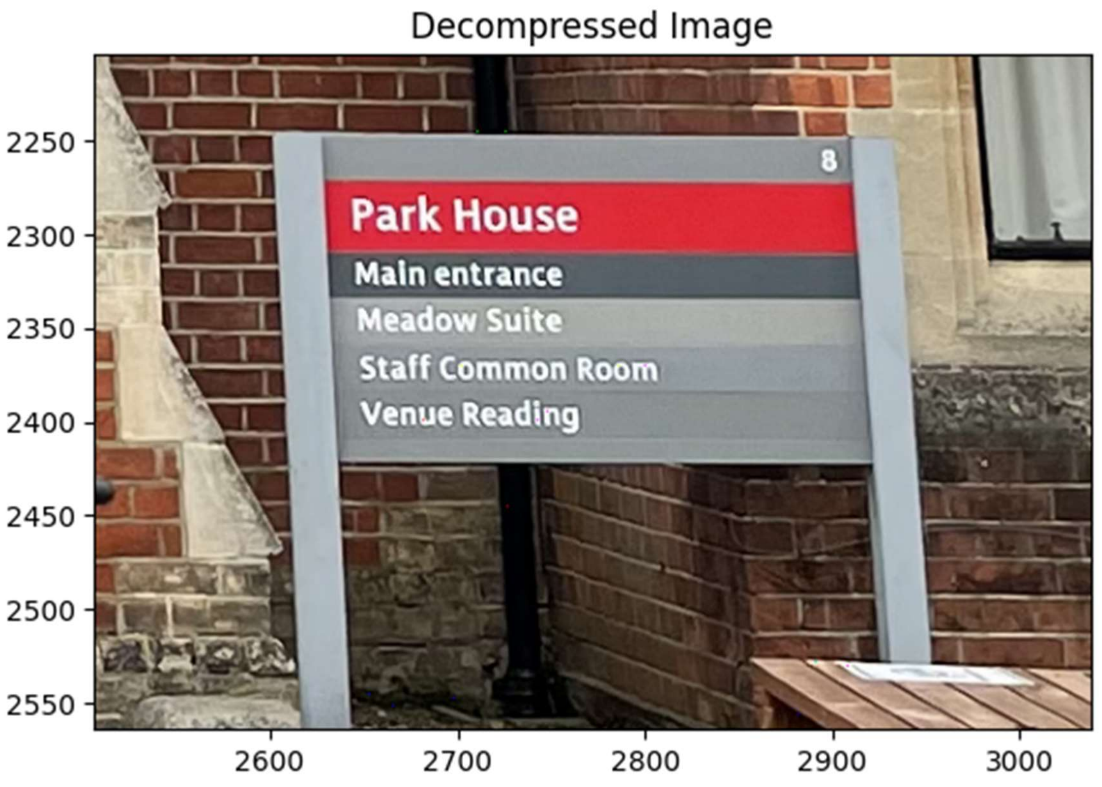
[8] Wikipedia. (2021). LZ77 and LZ78. [online] Available at: https://en.wikipedia.org/wiki/LZ77_and_LZ78.

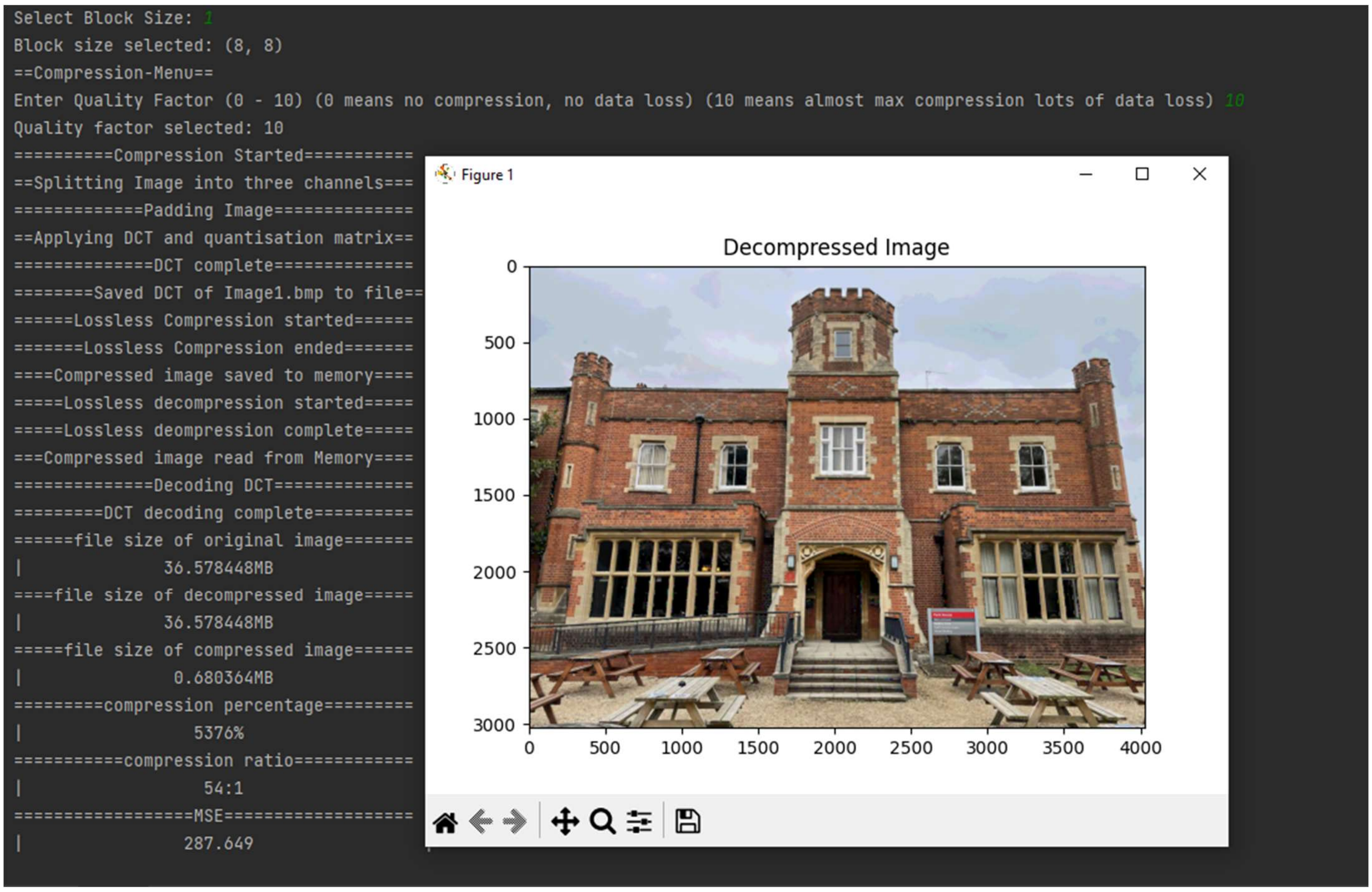
[9] Dubey, S. (2022). compression-DCT. [online] GitHub. Available at: https://github.com/getsanjeev/compression-DCT [Accessed 6 Dec. 2022].

Appendix

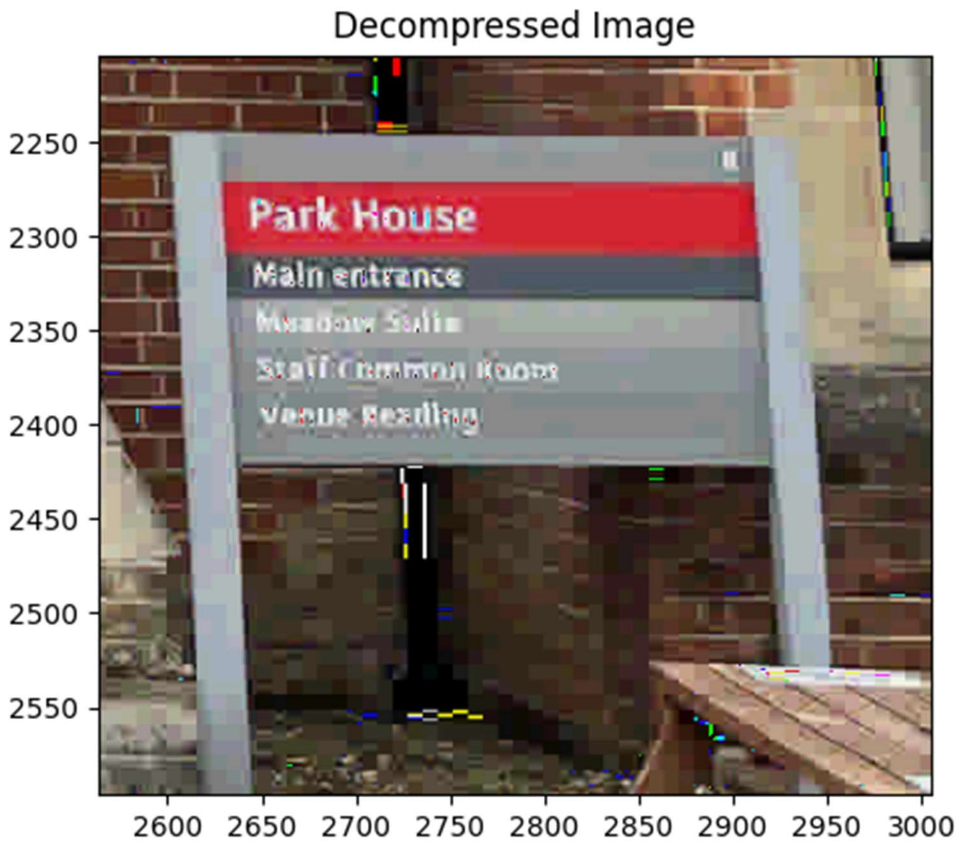


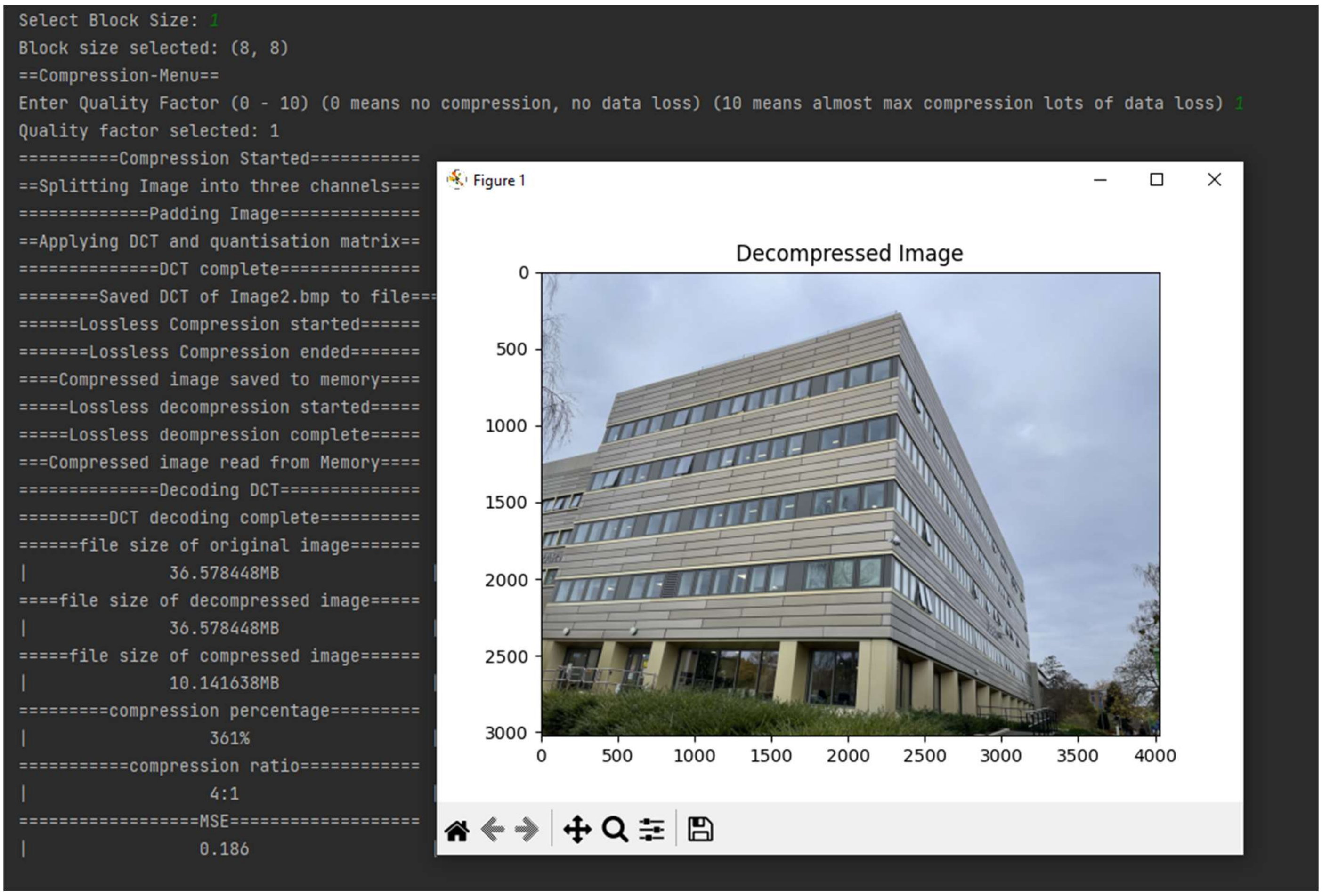
```
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 1
Quality factor selected: 1
==========Compression Started==========
==Splitting Image into three channels===
============Padding Image=============
==Applying DCT and quantisation matrix==
=============DCT complete=============
========Saved DCT of Image1.bmp to file====
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image======
|              36.578448MB              |
====file size of decompressed image=====
|              36.578448MB              |
=====file size of compressed image======
|              14.956691MB              |
=========compression percentage=========
|                  245%                 |
===========compression ratio===========
|                  2:1                  |
=================MSE==================
|                 0.244                 |
```



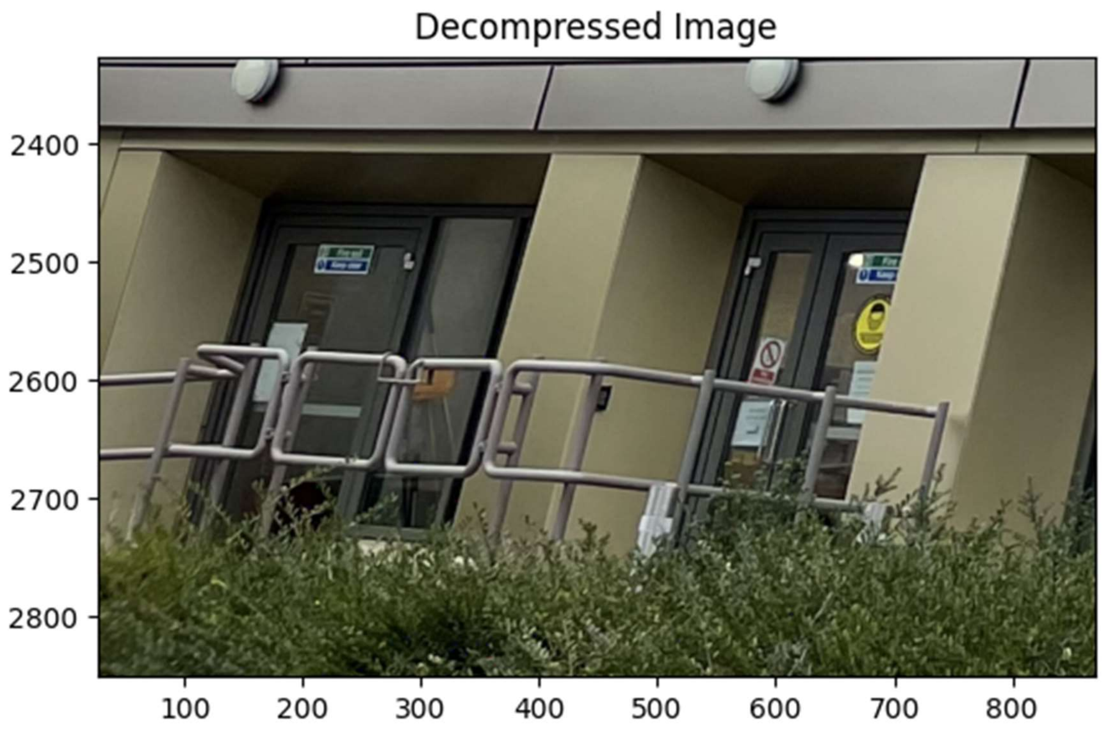


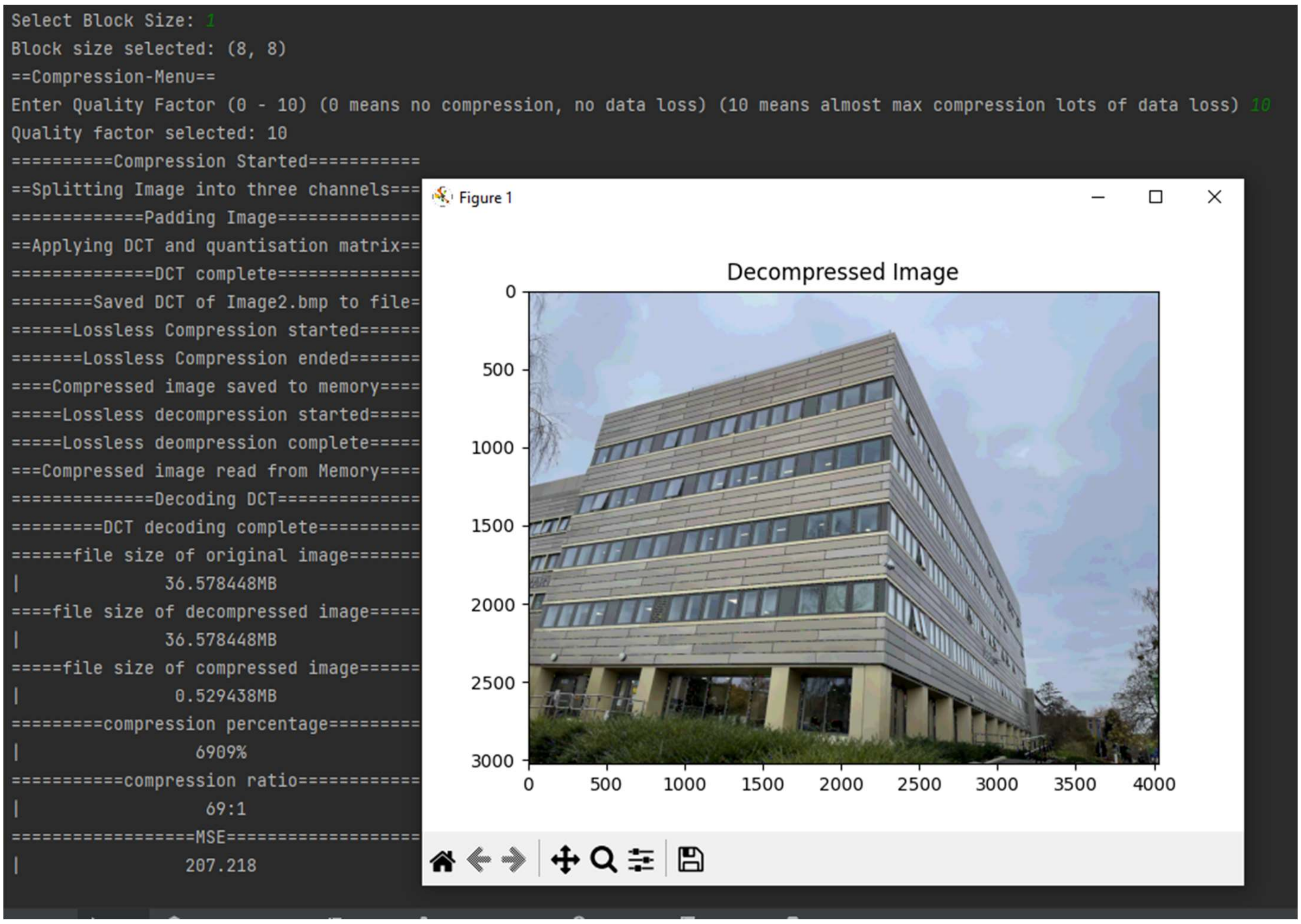
```
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 10
Quality factor selected: 10
==========Compression Started==========
==Splitting Image into three channels===
=============Padding Image=============
==Applying DCT and quantisation matrix==
==============DCT complete=============
========Saved DCT of Image1.bmp to file==
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image=======
|               36.578448MB
====file size of decompressed image=====
|               36.578448MB
=====file size of compressed image======
|                0.680364MB
========compression percentage========
|                  5376%
==========compression ratio===========
|                   54:1
=================MSE==================
|                 287.649
```



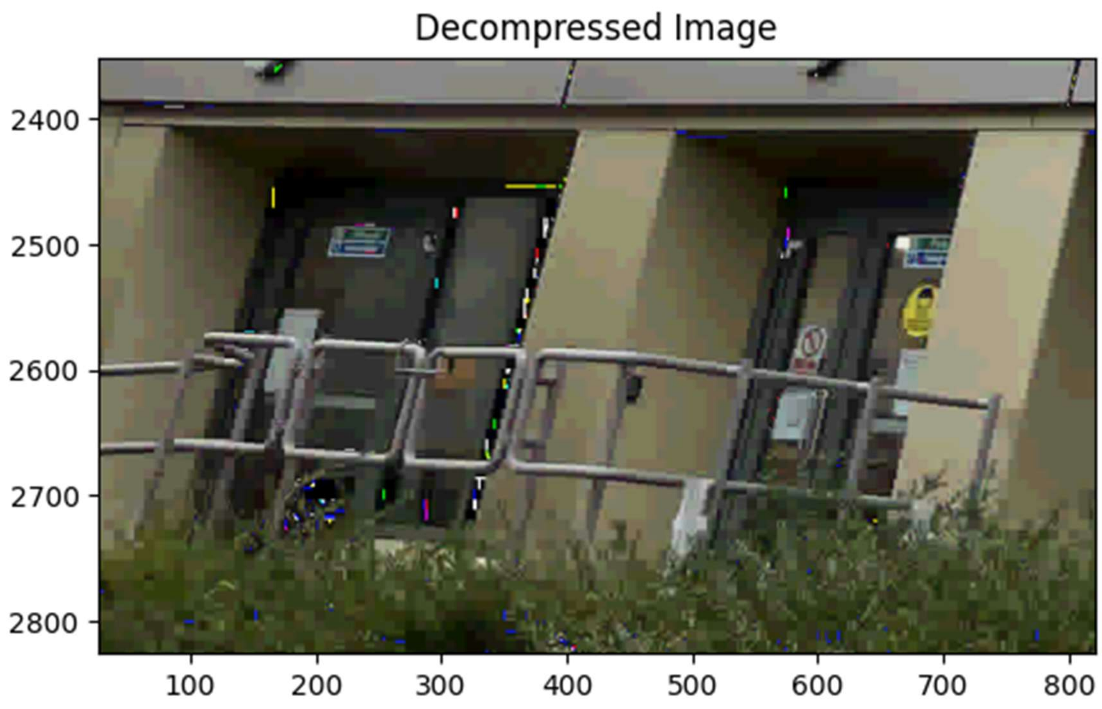


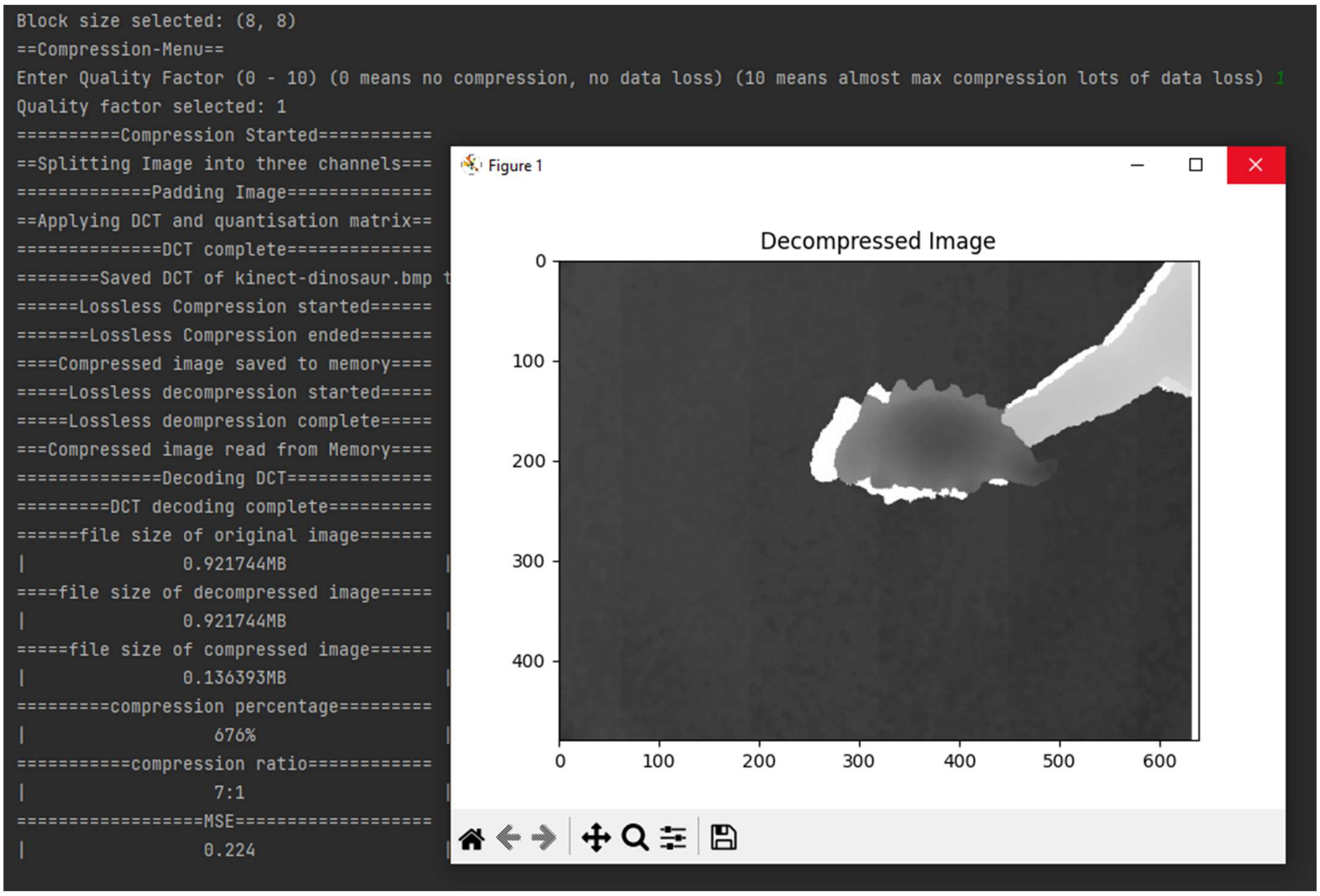
```
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 1
Quality factor selected: 1
==========Compression Started==========
==Splitting Image into three channels===
=============Padding Image=============
==Applying DCT and quantisation matrix==
==============DCT complete=============
========Saved DCT of Image2.bmp to file===
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image=======
|               36.578448MB
====file size of decompressed image=====
|               36.578448MB
=====file size of compressed image======
|               10.141638MB
=========compression percentage=========
|                   361%
===========compression ratio===========
|                   4:1
=================MSE==================
|                  0.186
```



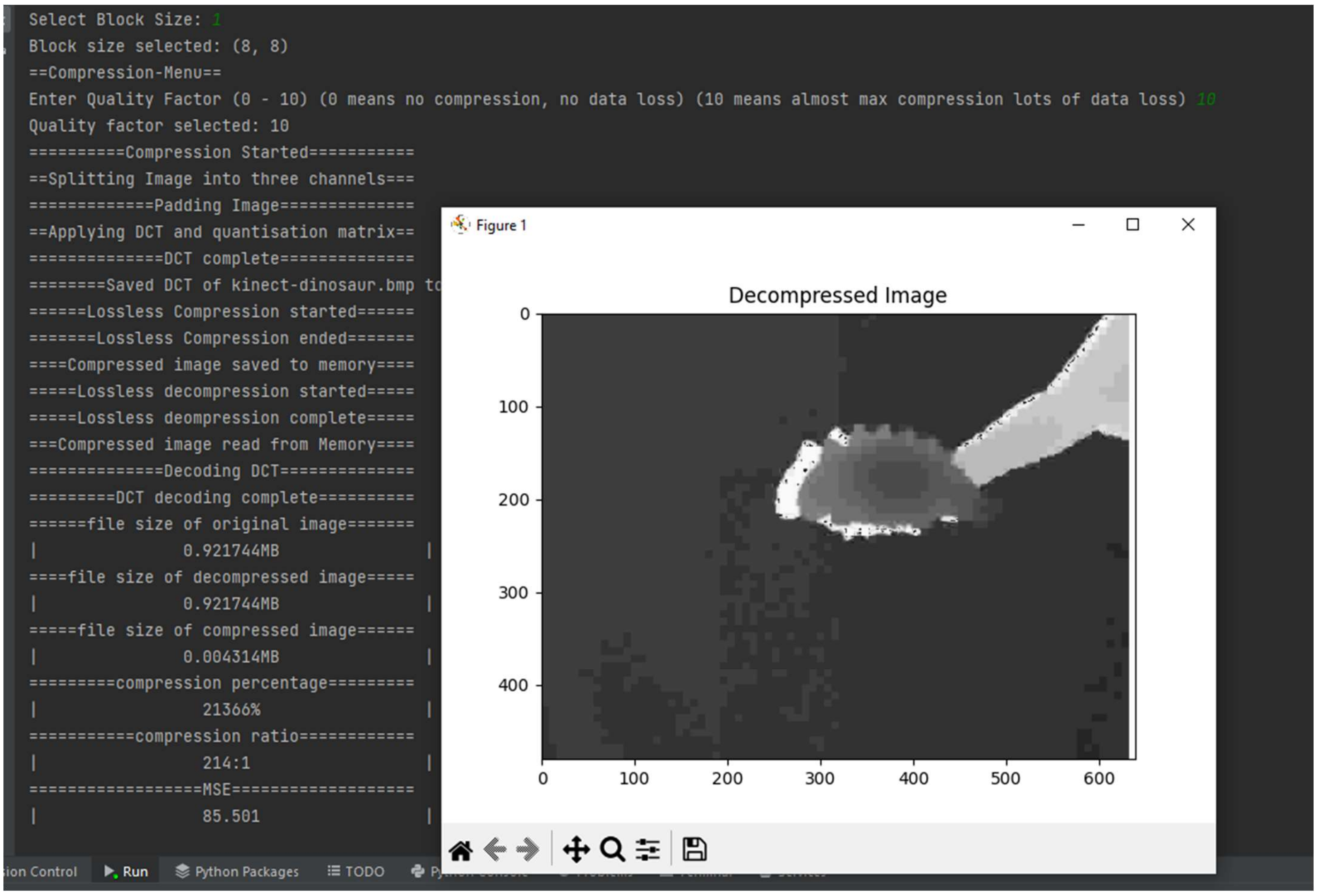


```
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 10
Quality factor selected: 10
==========Compression Started==========
==Splitting Image into three channels===
============Padding Image=============
==Applying DCT and quantisation matrix==
==============DCT complete=============
========Saved DCT of Image2.bmp to file=
======Lossless Compression started=====
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
==============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image======
|               36.578448MB
====file size of decompressed image=====
|               36.578448MB
=====file size of compressed image======
|                0.529438MB
=========compression percentage=========
|                  6909%
===========compression ratio============
|                   69:1
=================MSE==================
|                  207.218
```





```
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 1
Quality factor selected: 1
=========Compression Started==========
==Splitting Image into three channels===
============Padding Image=============
==Applying DCT and quantisation matrix==
=============DCT complete=============
========Saved DCT of kinect-dinosaur.bmp t
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
=============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image=======
|               0.921744MB               |
====file size of decompressed image=====
|               0.921744MB               |
=====file size of compressed image======
|               0.136393MB               |
========compression percentage=========
|                  676%                  |
==========compression ratio===========
|                  7:1                   |
=================MSE==================
|                 0.224                  |
```



```
Select Block Size: 1
Block size selected: (8, 8)
==Compression-Menu==
Enter Quality Factor (0 - 10) (0 means no compression, no data loss) (10 means almost max compression lots of data loss) 10
Quality factor selected: 10
=========Compression Started==========
==Splitting Image into three channels===
============Padding Image=============
==Applying DCT and quantisation matrix==
=============DCT complete=============
========Saved DCT of kinect-dinosaur.bmp to
======Lossless Compression started======
=======Lossless Compression ended=======
====Compressed image saved to memory====
=====Lossless decompression started=====
=====Lossless deompression complete=====
===Compressed image read from Memory====
=============Decoding DCT=============
=========DCT decoding complete=========
======file size of original image=======
|               0.921744MB               |
====file size of decompressed image=====
|               0.921744MB               |
=====file size of compressed image======
|               0.004314MB               |
========compression percentage=========
|                 21366%                 |
==========compression ratio===========
|                 214:1                  |
=================MSE==================
|                 85.501                 |
```

Original Image	Andreality Factor	Block size	Compressed image size	MSE	Comp ratio	Comp %	Decompressed Image
	1	8x8	14.95MB	0.244	245%	2:1	
	1	4x4	15.273	0.241	239%	2:1	
	5	4x4	2.86MB	55.976	1279%	13:1	
	5	8x8	2.46MB	51.409	1490%	15:1	
	5	16x16	2.31MB	52.876	1582%	16:1	

Code

Compression.py

```
import sys
import time

import cv2 as cv
from matplotlib import pyplot as plt
import os
import numpy as np
import argparse
```

```python
import os
import math
import numpy as np
from utils import *
from scipy import fftpack
from PIL import Image
from huffman import HuffmanTree
import os
from PIL import Image
import cv2
import numpy as np
import scipy.fftpack as fftpack
import zlib

from zigzag import zigzag
import warnings
warnings.filterwarnings("ignore")
prev_16 = lambda x: x >> 4 << 4
bSize = 8

import matplotlib.image as mpimg
chosen_file = ""
image_dict = {
}


# plt.imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
# plt.show()


def onRun():
    get_file_names()
    menu()


def get_file_names():
    x = 0
    directory = os.fsencode("images_before_compression")
    for file in os.listdir(directory):
        filename = os.fsdecode(file)
        if filename.endswith(".bmp"):
            x += 1
            image_dict[x] = filename
            continue
        else:
            continue
    x = 0
    directory = os.fsencode("images_after_compression")
    for file in os.listdir(directory):
        filename = os.fsdecode(file)
        if filename.endswith(".bmp"):
            x += 1
            continue
        else:
            continue


def display_files():
    print(f'{"LoadFile":=^20}')
    for key in image_dict:
        print(f'{key:5}', image_dict[key])
    user_input = input("Select file, (the number surrounded by [ ]")
    global chosen_file
    chosen_file = image_dict[int(user_input)]
    loadFile(chosen_file)
    return True
```

```python
def loadFile(string):
    global image
    global name
    image = cv.imread('images_before_compression/'+string)
    name = string
def Encoding_Quantitisation_Matrix(orig, quant):
    # import code
    # code.interact(local=vars())
    return (orig / quant).astype(np.int)


def Decoding_Quantitsation_Matrix(orig, quant):
    return (orig * quant).astype(float)


def encoding_dct_matrix(orig, bx, by):
    new_shape = (
        orig.shape[0] // bx * bx,
        orig.shape[1] // by * by,
        3
    )
    new = orig[
        :new_shape[0],
        :new_shape[1]
    ].reshape((
        new_shape[0] // bx,
        bx,
        new_shape[1] // by,
        by,
        3
    ))

    QUANT = 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]])
    img1 = cv2.imread('images_before_compression/'+name, -1)

    # splits image into three colour channels ~~ open cv takes BGR not RGB
    print(f'{"Splitting Image into three channels":=^40}')
    time.sleep(1)
    Blue, Green, Red = cv2.split(img1)

    [height, width] = Green.shape  # gets height and width of image, blue green or red
all have same heights

    # creates boxes with heights and assign widths
    heightOne = height
    widthOne = width
    height = np.float32(height)
    width = np.float32(width)
    boxHeight = math.ceil(height / bSize)
    boxHeight = np.int32(boxHeight)
    boxWidth = math.ceil(width / bSize)
    boxWidth = np.int32(boxWidth)
    Height = bSize * boxHeight
    Width = bSize * boxWidth
    # done creating boxes and assigning widths
    print(f'{"Padding Image":=^40}')
    time.sleep(1)
    # Image pad
    PaddedImageBlue = np.zeros((Height, Width))
    PaddedImageGreen = np.zeros((Height, Width))
    PaddedImageRed = np.zeros((Height, Width))
    # done padding with empty arrays

    # fill image with colours
    PaddedImageBlue[0:heightOne, 0:widthOne] = Blue[0:heightOne, 0:widthOne]
    PaddedImageGreen[0:heightOne, 0:widthOne] = Green[0:heightOne, 0:widthOne]
    PaddedImageRed[0:heightOne, 0:widthOne] = Red[0:heightOne, 0:widthOne]
    # done filling images with colours from imported image

    # display image in openCV before any compression (including padding)
    print(f'{"Applying DCT and quantisation matrix":=^40}')
    ImageWithPaddingBeforeCompression = cv2.merge([PaddedImageBlue, PaddedImageGreen,
PaddedImageRed])
    for y in range(boxHeight):
        rowOne = y * bSize
        rowTwo = rowOne + bSize

        for z in range(boxWidth):
            colOne = z * bSize
            colTwo = colOne + bSize

            BluePadded = PaddedImageBlue[rowOne: rowTwo, colOne: colTwo]
            GreenPadded = PaddedImageGreen[rowOne: rowTwo, colOne: colTwo]
            RedPadded = PaddedImageRed[rowOne: rowTwo, colOne: colTwo]

            # does dct
            BlueDCT = cv2.dct(BluePadded)
            GreenDCT = cv2.dct(GreenPadded)
            RedDCT = cv2.dct(RedPadded)
            # finish dct

            # start normalisations of dct
            BlueDCTNormalised = np.divide(BlueDCT, QUANT).astype(int)
            GreenDCTNormalised = np.divide(GreenDCT, QUANT).astype(int)
            RedDCTNormalised = np.divide(RedDCT, QUANT).astype(int)
            # end normalisation of dct

            # reodering through zigzag
            BlueReordering = zigzag(BlueDCTNormalised)
            GreenReordering = zigzag(GreenDCTNormalised)
            RedReordering = zigzag(RedDCTNormalised)
            # finishing zizag reordring

            # reshaping starting
            BlueReshaping = np.reshape(BlueReordering, (bSize, bSize))
            GreenReshaping = np.reshape(GreenReordering, (bSize, bSize))
            RedReshaping = np.reshape(RedReordering, (bSize, bSize))
            # reshaping done
            # applying change to padded channels
            PaddedImageBlue[rowOne: rowTwo, colOne: colTwo] = BlueReshaping
            PaddedImageGreen[rowOne: rowTwo, colOne: colTwo] = GreenReshaping
            PaddedImageRed[rowOne: rowTwo, colOne: colTwo] = RedReshaping
            # finished applying change to padded channels
    print(f'{"DCT complete":=^40}')
    DCTtransformOfImage = cv2.merge([PaddedImageBlue, PaddedImageGreen,
PaddedImageRed])
    cv2.imwrite('images_with_dct/{}encoded.bmp'.format(name), DCTtransformOfImage)
    cv2.imwrite('images_with_dct/{}encoded_as_uint8.bmp'.format(name),
np.uint8(DCTtransformOfImage))
    print(f'{"Saved DCT of {} to file":=^40}'.format(name))
    time.sleep(2)
    return fftpack.dctn(new, axes=[1,3], norm='ortho')


def decode_dct(orig, bx, by):
    print(f'{"Decoding DCT":=^40}')
    time.sleep(5)
    print(f'{"DCT decoding complete":=^40}')
    return fftpack.idctn(orig, axes=[1,3], norm='ortho'
    ).reshape((
        orig.shape[0]*bx,
        orig.shape[2]*by,
        3
    ))


def encoding_compressed_image(x):
    print(f'{"Lossless Compression started":=^40}')
    time.sleep(1)
    print(f'{"Lossless Compression ended":=^40}')
    return zlib.compress(x.astype(np.int8).tobytes())


def decoding_compressed_image(orig, shape):
    print(f'{"Lossless decompression started":=^40}')
    time.sleep(5)
    print(f'{"Lossless deompression complete":=^40}')
    return np.frombuffer(zlib.decompress(orig),
dtype=np.int8).astype(float).reshape(shape)


def compress(blocksize,quality):
    im = image
    quants = [quality]  # [0.5, 1, 2, 5, 10]
    blocks = []  # [(2, 8), (8, 8), (16, 16), (32, 32), (200, 200)]
    blocks.append(blocksize)

    for qscale in quants:
        for bx, by in blocks:
            quant = (
                (np.ones((bx, by)) * (qscale * qscale))
                .clip(-100, 100)  # to prevent clipping
                .reshape((1, bx, 1, by, 1))
            )



            #encoding using quality factor block size and the image
            enc = encoding_dct_matrix(im, bx, by)
            encq = Encoding_Quantitisation_Matrix(enc, quant)
            encz = encoding_compressed_image(encq)
            print(f'{"Compressed image saved to memory":=^40}')

            #decoding using old encoding lossless compression
            decz = decoding_compressed_image(encz, encq.shape)
            print(f'{"Compressed image read from Memory":=^40}')
            decq = Decoding_Quantitsation_Matrix(encq, quant)

            dec = decode_dct(decq, bx, by)
            cv2.imwrite("images_after_compression/" +name
+"_recompressed_quant_{}_block_{}x{}.bmp".format(qscale, bx, by), dec.astype(np.uint8))
            cv2.imwrite(
                "images_after_compression/" + name +
"_recompressed_quant_{}_block_{}x{}.jpeg".format(qscale, bx, by),
                dec.astype(np.uint8))

            # closing all open windows
            cv2.destroyAllWindows()
            MSE = round(np.square(np.subtract(dec, im)).mean(),3)
            compression_percentage =
str(round(sys.getsizeof(np.uint8(im))/sys.getsizeof(encz) * 100)) + "%"
            compression_ratio =
str(round(sys.getsizeof(np.uint8(im))/sys.getsizeof(encz))) + ":1"
```

```
            originalFileSize = str(sys.getsizeof(np.uint8(im)) /1000000) + "MB"
            compressedFileSize = str(sys.getsizeof(encz) /1000000) + "MB"
            decompressedFileSize = str(sys.getsizeof(np.uint8(dec)) /1000000) + "MB"
            print(f'{"file size of original image":=^40}')
            print(f"|{originalFileSize:^40}|")
            print(f'{"file size of decompressed image":=^40}')
            print(f"|{decompressedFileSize:^40}|")
            print(f'{"file size of compressed image":=^40}')
            print(f"|{compressedFileSize:^40}|")
            print(f'{"compression percentage":=^40}')
            print(f"|{compression_percentage:^40}|")
            print(f'{"compression ratio":=^40}')
            print(f"|{compression_ratio:^40}|")
            print(f'{"MSE":=^40}')
            print(f"|{MSE:^40}|")

            plt.title("Decompressed Image")
            plt.imshow(cv.cvtColor(dec.astype(np.uint8), cv.COLOR_BGR2RGB))
            plt.show()
            cv2.waitKey(0)



def CompressMenu():
    blocksize = []
    qualityfactor = 0
    print(f'{"Compression-Menu":=^20}')

    print(f'{"0":5} : 4x4')
    print(f'{"1":5} : 8x8')
    print(f'{"2":5} : 16x16')
    print(f'{"3":5} : 32x32')
    user_input = int(input("Select Block Size: "))
    if user_input == 0:
        blocksize = (4,4)
    if user_input == 1:
        blocksize = (8,8)
    if user_input == 2:
        blocksize = (16,16)
    if user_input == 3:
        blocksize = (32,32)
    print(f'{"Block size selected:":=^20}',format(blocksize))
    print(f'{"Compression-Menu":=^20}')
    qualityfactor = int(input("Enter Quality Factor (0 - 10) (0 means no compression,
no data loss) (10 means almost max compression lots of data loss) "))
    print(f'{"Quality factor selected:":=^20}', format(qualityfactor))
    print(f'{"Compression Started":=^40}')
    compress(blocksize,qualityfactor)

def menu():
    is_loaded = False
    while True:
        print(f'{"Menu":=^20}')
        if is_loaded:
            print(f'{"0":5} : Compress')
            #print(f'{"1":5} : Save')
            print(f'{"2":5} : Display Image')
            print(f'{"3":5} : Display File Size')
            print(f'{"4":5} : Load Another Image')
        else:
            print(f'{"5":5} : Load')
        print(f'{"6":5} : Quit')
        user_input = int(input("Select process, (the number surrounded by [ ]"))

        if user_input == 0 and is_loaded != False:
           CompressMenu()
        if user_input == 1 and is_loaded != False:
```

```python
            save_to_file()
        if user_input == 2 and is_loaded != False:
            display_image()
        if user_input == 3 and is_loaded != False:
            print(get_file_size())
        if user_input == 4 and is_loaded != False:
            is_loaded = display_files()
        if user_input == 5 and is_loaded == False:
            is_loaded = display_files()
        if user_input == 6:
            quit()

def save_to_file():
    print("save")
def display_image():
    plt.imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
    plt.show()
def get_file_size():
    global filesize
    filesize = round(os.path.getsize('images_before_compression/'+name) / 1048576,2)
    return "File size is :  "+str(filesize)+ " MB"
onRun()
```

Huffman.py

```python
from queue import PriorityQueue


class HuffmanTree:

    class __Node:
        def __init__(self, value, freq, left_child, right_child):
            self.value = value
            self.freq = freq
            self.left_child = left_child
            self.right_child = right_child

        @classmethod
        def init_leaf(self, value, freq):
            return self(value, freq, None, None)

        @classmethod
        def init_node(self, left_child, right_child):
            freq = left_child.freq + right_child.freq
            return self(None, freq, left_child, right_child)

        def is_leaf(self):
            return self.value is not None

        def __eq__(self, other):
            stup = self.value, self.freq, self.left_child, self.right_child
            otup = other.value, other.freq, other.left_child, other.right_child
            return stup == otup

        def __nq__(self, other):
            return not (self == other)

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

        def __le__(self, other):
            return self.freq < other.freq or self.freq == other.freq

        def __gt__(self, other):
            return not (self <= other)

        def __ge__(self, other):
            return not (self < other)

    def __init__(self, arr):
        q = PriorityQueue()

        # calculate frequencies and insert them into a priority queue
        for val, freq in self.__calc_freq(arr).items():
            q.put(self.__Node.init_leaf(val, freq))

        while q.qsize() >= 2:
            u = q.get()
            v = q.get()

            q.put(self.__Node.init_node(u, v))

        self.__root = q.get()

        # dictionaries to store huffman table
        self.__value_to_bitstring = dict()

    def value_to_bitstring_table(self):
        if len(self.__value_to_bitstring.keys()) == 0:
            self.__create_huffman_table()
        return self.__value_to_bitstring

    def __create_huffman_table(self):
        def tree_traverse(current_node, bitstring=''):
            if current_node is None:
                return
            if current_node.is_leaf():
                self.__value_to_bitstring[current_node.value] = bitstring
                return
            tree_traverse(current_node.left_child, bitstring + '0')
            tree_traverse(current_node.right_child, bitstring + '1')

        tree_traverse(self.__root)

    def __calc_freq(self, arr):
        freq_dict = dict()
        for elem in arr:
            if elem in freq_dict:
                freq_dict[elem] += 1
            else:
                freq_dict[elem] = 1
        return freq_dict
```

zigzag.py [9] This is a third party source, we did not create and or claim ownership to this source code.

```
import numpy as np


def zigzag(imageInput):

    height = 0
    Vertex = 0
    vertexMinimum = 0
    heightMinimum = 0
    vertexMaximum = imageInput.shape[0]
    HeightMaximum = imageInput.shape[1]
    i = 0
```

```
output = np.zeros(vertexMaximum * HeightMaximum)

# ----------------------------------

while Vertex < vertexMaximum and height < HeightMaximum:

    if (height + Vertex) % 2 == 0:

        if Vertex == vertexMinimum:


            output[i] = imageInput[Vertex, height]

            if height == HeightMaximum:
                Vertex = Vertex + 1
            else:
                height = height + 1

            i = i + 1
        elif height == HeightMaximum - 1 and Vertex < vertexMaximum:



            output[i] = imageInput[Vertex, height]
            Vertex = Vertex + 1
            i = i + 1
        elif Vertex > vertexMinimum and height < HeightMaximum - 1:


            output[i] = imageInput[Vertex, height]
            Vertex = Vertex - 1
            height = height + 1
            i = i + 1
    else:


        if Vertex == vertexMaximum - 1 and height <= HeightMaximum - 1:

            output[i] = imageInput[Vertex, height]
            height = height + 1
            i = i + 1
        elif height == heightMinimum:



            output[i] = imageInput[Vertex, height]

            if Vertex == vertexMaximum - 1:
                height = height + 1
            else:
                Vertex = Vertex + 1

            i = i + 1
        elif Vertex < vertexMaximum - 1 and height > heightMinimum:


            output[i] = imageInput[Vertex, height]
            Vertex = Vertex + 1
            height = height - 1
            i = i + 1

    if Vertex == vertexMaximum - 1 and height == HeightMaximum - 1:



        output[i] = imageInput[Vertex, height]
```

```
            break

    return output

def inZ(input, vmax, hmax):

    h = 0
    v = 0

    vmin = 0
    hmin = 0

    output = np.zeros((vmax, hmax))

    i = 0

    # ----------------------------------

    while v < vmax and h < hmax:


        if (h + v) % 2 == 0:  # going up

            if v == vmin:

                # print(1)

                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:


                output[v, h] = input[i]
                v = v + 1
                i = i + 1
            elif v > vmin and h < hmax - 1:



                output[v, h] = input[i]
                v = v - 1
                h = h + 1
                i = i + 1
        else:



            if v == vmax - 1 and h <= hmax - 1:


                output[v, h] = input[i]
                h = h + 1
                i = i + 1
            elif h == hmin:
                output[v, h] = input[i]
                if v == vmax - 1:
                    h = h + 1
                else:
                    v = v + 1
                i = i + 1
            elif v < vmax - 1 and h > hmin:



                output[v, h] = input[i]
                v = v + 1
                h = h - 1
                i = i + 1

        if v == vmax - 1 and h == hmax - 1:


            output[v, h] = input[i]
            break

    return output
```