Project Report – Compression Algorithm Design and Development

Catalog

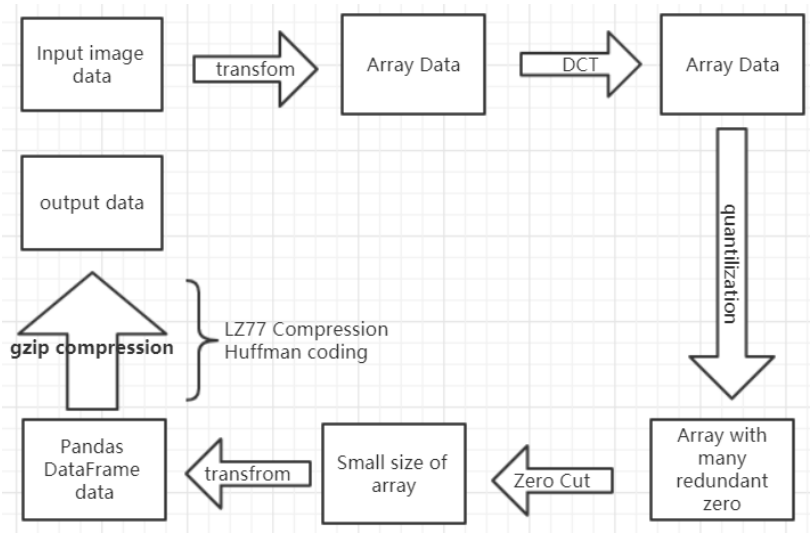
1. **General introduction：**

The compression algorithm is developed by Python3.6 language. Some other compression algorithm such as Human coding may be difficult to be realized by myself and it may increase the encoding and decoding complexity but Python library data tools such as NumPy and Pandas help me handle the image pixels data with less complexity. Some processes are not introduced in lectures such as zero cut and gzip provided by Pandas. These new adding methods help me to do further compression to improve the data compression ratio.

2. **The Compression Algorithm Design and Reasons:**

1) Encoding processes：

A. The global encoding processes view:



picture 1

B. The specific steps:

a) Transform image into Array data:[i]

The provided sample images are gray scale pictures so I only need to deal with one plane with 8-bit bit deep. Firstly, I use OpenCV library to load image pixel values into NumPy array matrix.

```
img_location = 'Textures/'+img_location
img = cv2.imread(img_location, 0)
```

loading image picture-2

b) Discrete Cosine Transform (DCT)[ii]

Discrete Cosine Transform is often used in image compression processes. It can find the correlation among adjacent pixels so it has a strong energy compaction property, compacting the energy on the left top corner. So, unquestionably, this technique is included in my compression algorithms and creating algorithm codes are below.

But the previous compression step is different to corresponding step of JEPE. I directly compact energy of whole image by Discrete Cosine Transform without division of 8 over 8 blocks. The division to be 8 over 8 blocks increases the processing complexity thought it may caught more energy on the overall pixel

distribution and reduce the number of redundant zeros. But after the Discrete Cosine Transform, we will handle this problem later.

```
def create_transform_table(N):
  transform_table = np.zeros((N,N))
  transform_table[0, :] = 1 * np.sqrt(1/N)
  for i in range(1, N):
      for j in range(N):
          transform_table[i, j] = np.cos(np.pi * i * (2*j+1) / (2 * N )
  ) * np.sqrt(2 / N )

  return transform_table
```

Creating DCT Table picture-3

c) Quantization:

Quantization is the simple lossy technique to do the compression. The larger quantization value will achieve better compression perform but it will lead to the bad quality of recovery image. So how to choose the best Quantization value is also a key part of compression algorithm development.

We know that after the Discrete Cosine Transform, the most energy will locate on the left top corner. This means that on this corner the pixel values are far larger than other. At the same time, these values are more important and sensitive. So, the corresponding quantization value on this corner are smaller than other areas reverse. This mechanism will do a better tradeoff between compression excellence and recovery image quality.

The sample image size is 512×512 and 1024×1024. I design a quantization table to fit the size. The values in quantization table are exponentially increase from the left-top corner to the right bottom-right corner. In addition, I add the parameter to adjust the last exponential number to make sure the number will not increase too large.

```
def create_quantilisation_table (N):

    #创建指数型增加的量化矩阵
    power = math.log2(N) #梯度
    division_index = np.logspace(1,power,power,base = 2,dtype = 'int') -1 #梯度列表

    quantilization_table = np.zeros((N,N))#创建量化矩阵
    adjusted_index = -2 #调整最后梯度的索引
    last_index = 0 #梯度起始值

    #量化矩阵赋值
    for q_index in division_index[:adjusted_index]:
      quantilization_table[last_index:q_index,:] = q_index
      quantilization_table[:,last_index:q_index] = q_index
      last_index  = q_index

    #量化矩阵调整赋值
    for q_index in division_index[adjusted_index:]:
      quantilization_table[last_index:q_index,:] = division_index[adjusted_index-1]
      quantilization_table[:,last_index:q_index] = division_index[adjusted_index-1]
      last_index  = q_index

    quantilization_table[division_index[-1],:] = division_index[adjusted_index-1]
    quantilization_table[:,division_index[-1]:] = division_index[adjusted_index-1]

    return quantilization_table
```

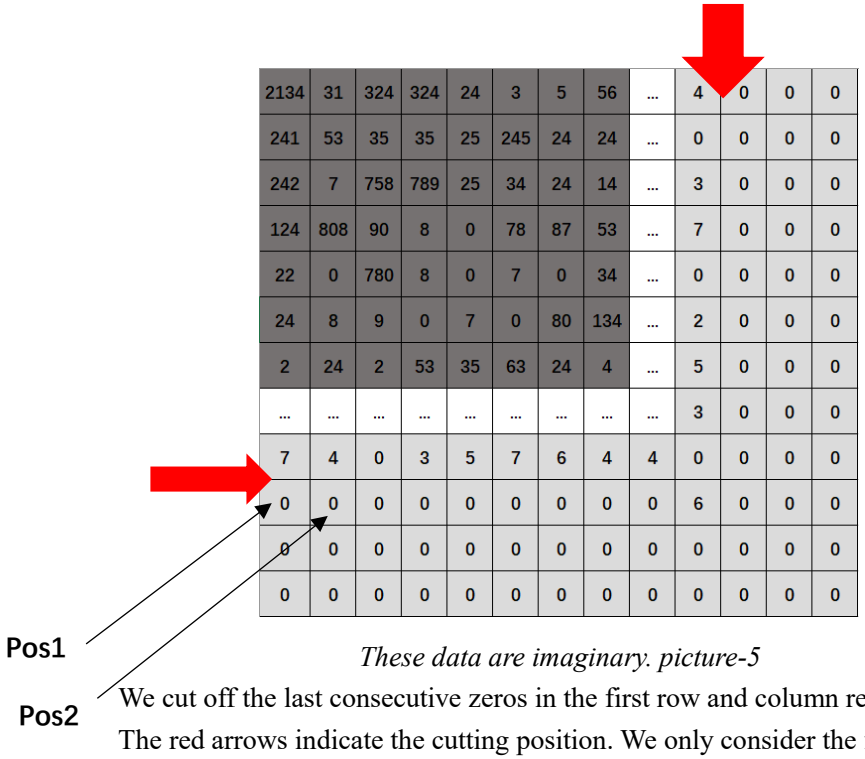
The creating personalized quantization table algorithm picture-4

The detail values of quantization are shown in the csv file in the folder: '512.csv' and '1024.csv'. The first row of quantization in the csv files is the column index, which are not loaded when we import these tables.

d) Zero Cut

The Zero Cut method is a **self-designed** algorithm. We know that applying the whole image into DCT and quantization will generate many redundant zeros without division into 8 over 8 blocks. The Zero Cut methods may handle this problem to great extent. The main ideal of this method is to cut off the right bottom corner with many zeros to discard zeros.

For example, the below array is the data after DCT and quantization



These data are imaginary. picture-5

We cut off the last consecutive zeros in the first row and column respectively. The red arrows indicate the cutting position. We only consider the first rows and columns because the energy is compacted on the left top corner. So, in this situation, the corresponding positions of remaining columns and rows have lower possibility to have none-zero values. For example, in this chart, If the value of position 1 is zero, the value of position 2 will be zero with high possibility.

2134	31	324	324	24	3	5	56	...	4
241	53	35	35	25	245	24	24	...	0
242	7	758	789	25	34	24	14	...	3
124	808	90	8	0	78	87	53	...	7
22	0	780	8	0	7	0	34	...	0
24	8	9	0	7	0	80	134	...	2
2	24	2	53	35	63	24	4	...	5
...	...	...	...	...	...	...	...	...	3
7	4	0	3	5	7	6	4	4	0

The data after Zero Cut picture - 6

e) Transform array data into DataFrame:

The Pandas data processing tool provides us with a high-efficiency method to store and load data, particularly array data. So, I use this data type to store the compressed image data because it provides a way to output and store data. In addition, it provided many different kind of output file types such as CSV, XSL an TXT etc.

f) Gzip DataFrame to compress and output data:

The Gzip compression technique combines LZ77 and Huffman coding, these two kinds of compression algorithms. LZ77 compresses data by finding repeated occurrences of data and building the reference index of a single copy of that uncompressed data occurring earlier. Huffman code is introduced in our lecture and it is the lossless data compression algorithms.

So, the combination of two lossless data compression algorithms will perform better in compression than any one of two.[iii]

At beginning, I planned to output data to be txt or csv files. But these kinds of datatype are difficult to be reloaded to recover the image and generate redundant information such as column index.

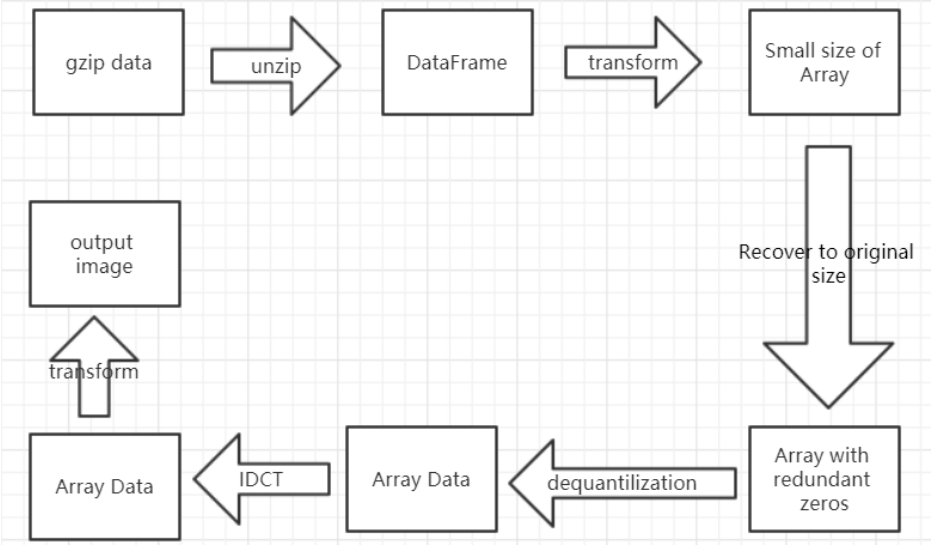
Luckily and occasionally, I find a compression parameter when I write out DataFrame into the disk. This parameter is provided by DataFrame data structure. I choose ‘gzip’ compression parameter to output the data.

```
# 输出压缩的encode图片，并且gzip 进行压缩
encode_data_path = 'encode_data/'+img_name
df_compress_img = pd.DataFrame(compressed_img)
df_compress_img.to_csv(encode_data_path,
                       compression = 'gzip',
                       index = False,
                       encoding = 'utf-8')
```

The gzip compression codes picture-7

2) Decoding processes

A. The global view of encoding:



Picture-8

B. The specific steps:

The processes of decoding are to reverse steps of encoding. All steps are almost similar to general encoding processes.

The one process I want to mention is recovering to original size. Because of Zero Cut that we abandon many zero values on the right-bottom corner. When decoding, we need to fill zero values to enlarge the width and height to be original size.In addition, when decoding because of different size of width and height, we must choose the corresponding quantization table to do the dequantization.

These two decoding processes of decoding (recovering to original width and height, dequantization) must need the original size information.

To make the recovering algorithm complexity simple, I find that image names contain the size information. Between 1.1.02.tiff and 1.2.13.tiff, the size of these images is 512 × 512 and the size of remaining image is 1024 × 1024.
The corresponding recovering codes shown below:

```python
#----------------------decode--------------
#input your encode file
def decode(encode_data = '1.1.13'):
  quantilisation_table =  np.array([])
  transform_table = np.array([])
  N= 0
  dir_name = 'encode_data/'
  encode_data_path = dir_name +encode_data
  try:
    decompressed_read = pd.read_csv(encode_data_path,compression = 'gzip',encoding = 'utf-8')
    decompressed_read_img = decompressed_read.values
    if int(encode_data_path[-4])<3:
      N = 512
      quantilisation_table =  quantilisation_table_512
      transform_table = transform_table_512
    else:
      N = 1024
      quantilisation_table = quantilisation_table_1024
      transform_table = transform_table_1024
  except:
    print('Please enter the correct file path:')
    sys.exit(1)
```

3. Code execution instructions:

There are 5 Python running code files.

1) 'single_encode_decode.py':

This Python file encodes and decode the image files successively. You can run the Python files independently and input the image name:

The console is shown below:

```
In [2]: runfile('C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/Project/
single_encode_decode.py', wdir='C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/
Project')
quantilisation table created
transform table created

please enter the img name you need to encode:
```

Input the image names in Textures Folder: such as 1.4.10.tiff

```
In [2]: runfile('C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/Project/
single_encode_decode.py', wdir='C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/
Project')
quantilisation table created
transform table created

please enter the img name you need to encode:1.4.10.tiff
1.4.10
Encode & Decode Done!
```

The output encode destination is 'encode_data' folder and the name of encoded files is the original image name excluded suffix. Such as 1.4.10.At same time, there is will be decoded output image in 'all_decode_img' folder. The decode output name is same as original one.

2) 'multithreading.py'

This Python running codes use the multithreading technique to improve the execution efficiency to encode and decode 50 images concurrently, which requires larger memory space but you can adjust the 'max_workers' parameter to reduce the memory requirement.

The output encoded data are stored in 'encode_data' folder. Because the multithreading program calls the functions in 'single_encode_decode.py', the

output decode images are also stored in ‘all_decode_img ’folder.

After finishing every task in multithreading pool will returns the measure indies, PSNR, bit rate, and compression ratio and automatedly write them out in the excel file **‘measure_index.xls’.**

At last, the average completed time of successive encoding and decoding of 50 images is about 28 seconds. The use time will be shown on console after completing.

3) **‘PSNR.py’**

You inputting the image name, this PSNR Python codes will return the corresponding PSNR values. And in order to reduce the number of codes this python function is imported by other python codes.

4) **‘encoder.py’**

The python file is an encoder. You can encode a single image by inputting the image name in Textures’ folder.

The output encoded data will be stored in ‘encode_data’ folder. And its files name is same as the original images name without suffix.

```
In [2]: runfile('C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/Project/
encoder.py', wdir='C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU Courses/COMP 7790 Special Topics/Project')
quantilisation table created
transform table created

please enter the img name you need to encode:1.4.09.tiff
1.4.09
Encode Done!
```

5) **‘decoder.py’**

The python file is a decoder. You can decode a single encoded data by inputting the files in ‘encode_data ’folder.

The output encoded data will be stored in ‘single_decode’ folder. And the decoded image name is same as the original one.

4. Discuss the results and possible reasons:

The average bit rate, PSNR and compression ratio for 50 images are shown at the ‘measure_index.xls’ Excel files.

1) **The results:**

- **Average bit rate：0.13**

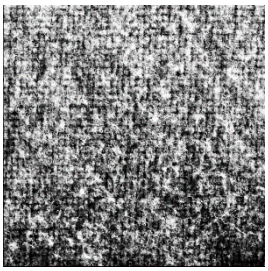
The bit rate of every encode images is shown at “measure_index.xls”. And the average of bite rate for the all images is 0.13. This figure shows the good performance of compression.

- **Average peak signal-to- noise ratio: 29.32**

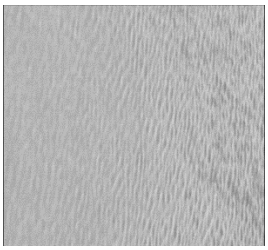
The peak signal-to-noise ratios for every image are shown at “measure_index.xls”. the peak signal-to noise ratio remains at around 30. The average PSNR for all images is 29.32. The figure is reasonable and tells us that there is no big difference between

the recovery images and the original images.

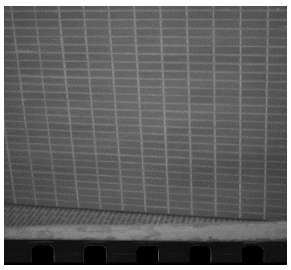
- The images with highest bit rate is ‘1.2.05.tiff’:
  The bit rate is 0.31 with PSNR 28.32



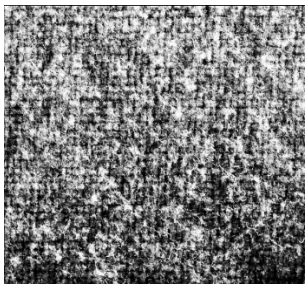
- The images with lowest bit rate is ‘1.3.08.tiff’:
  The bit rate is 0.31 with PSNR 31.24



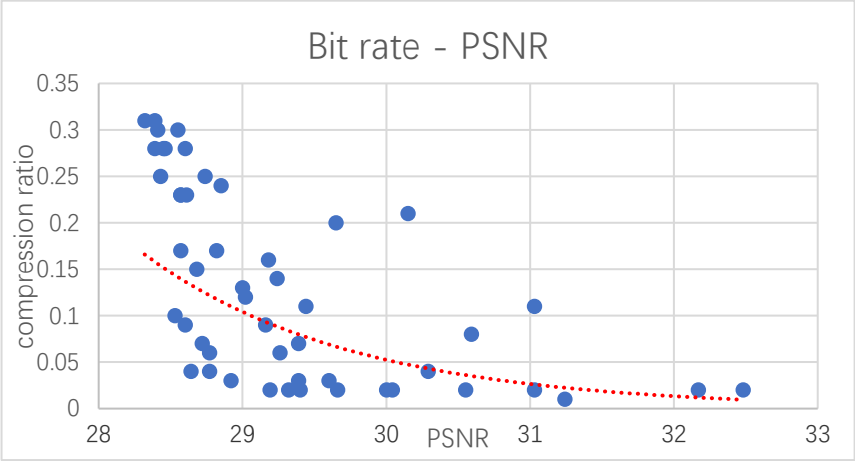
- The images with highest PSNR is ‘1.4.04.tiff’
  The PSNR is 32.48 with bit rate 0.02



- The images with lowest PSNR is ‘1.2.05.tiff’
  The PSNR is 28.32 with bit rate 0.31



2) The possible reasons:



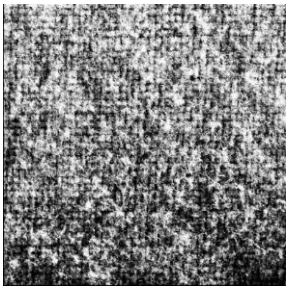
The chart of relation of Compression Ratio – PSNR

Form the chart, we can infer that in this compression algorithm, there may be a exponential relationship between Bit rate and PSNR. The higher the PSNR value, meaning that quality of the recovery image is worse, may lead to the lower bit rate. The possible reason is that if the image achieves the compression excellence, this means that it may lose great amount of energy. In other words, there is a big difference from the original images, leading to the high PSNR.

Overall, the combination of compression algorithms controls the PSNR at the stable and narrow range though with a great variety of compression ratio and bit rate

To more details, the reasons for 4 images are discussed below:

1) **The images with highest bit rate is '1.2.05.tiff':**
   The bit rate is 0.31 with PSNR 28.32



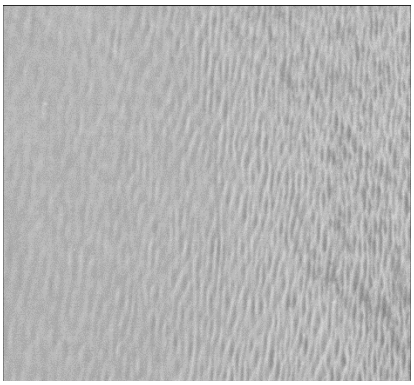
   This image achieves the worst compression performance after being encoded. From the image, we can see that the whole picture remains at the high frequency. and pixel values changes frequently and randomly. In addition, there is no particular pattern in the images, which means that it has much lower correlation among adjacent pixel values. In this case, after DCT, the image will remain many non-zero value and it lead to the bigger size for next further compression. Let me show the more confident and concrete proof.

```
In [4]: runfile('C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU C
encoder.py', wdir='C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU
quantilisation table created
transform table created

please enter the img name you need to encode:1.2.05.tiff
1.2.05
504
495
Encode Done!
```

We encode the images 1.2.05.tiff, because after DCT, the image will remain many non-zero value. Even after Zero Cut, the size just reduce a little bit from 512 × 512 to 504-1 × 495-1(because the index start from zero). This means that DCT has a not good enough performance to compact energy on the left-top corner.

2) The images with lowest bit rate is '1.3.08.tiff':

The lower bit rate means that the higher compression ratio.



Compared to 1.2.05.tiff, the 1.3.08.tiff image has how much lower frequency change, indicating that adjacent pixel values have high possibility to be the same value. In this way, the DCT can caught repeated information in this picture and have excellent performance in energy compaction. After DCT, the images will generate many redundant zeros so we can use 'Zero Cut' to reduce the number of zero to make the image become the smaller size.

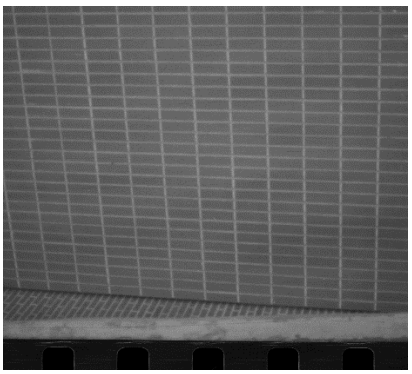
```
In [5]: runfile('C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU C
encoder.py', wdir='C:/Users/DELL/Desktop/HKBU/HKBU-ITM/BU
quantilisation table created
transform table created

please enter the img name you need to encode:1.3.08.tiff
1.3.08
120
173
Encode Done!
```

After DCT and Zero Cut, the size of image is reduced from 1024 × 1024 to

120-1× 127-1. After that, the size is reduced to be 1/69 of original.

3) **The images with highest PSNR is '1.4.04.tiff':**

The corresponding bit rate for this image is 0.02. This figure shows the good compression preformation at top 2 but the PSNR index has the relative value, which indicates that recovery images lose relatively great amount of energy and has the bigger difference from the original image.



1.4.04.tiff

The image has many small cells at nearly same gray level. This mean that the image contains a great number of redundant values of adjacent pixels. In this situation, the DCT has the excellence in compacting the energy on the left- top corner, and after that Zero Cut will shorten it to be an extremely small size.
The cell pattern exists on the images, but the boundary of cells experiences the large the frequency changes. This causes that the result after DCT must store these unrepeated data.

These mean that the compression lost more information of original images.
Because the high compression ratio achieved often means encoded data lost more data to recover the images. Furthermore, the high frequency of boundary makes the recovery images quality worse. These may be the possible for the high PSNR value.

4) **The images with lowest PSNR is '1.2.05.tiff':**

This images also has lowest compression ratio with 3.19 and highest bit rate 0.31. The other figures indicate that the compression for this image does not achieve a good performance.
We know from the 1) **The images with lowest bit rate is '1.3.08.tiff':** After DCT and Zero Cut, the size just reduces a little bit from 512 × 512 to 504 × 495. This means that it still stores many pixel data for recover. The image remain many non-zero value and it lead to the bigger size for next further compression. In this way, after compression the image store the largest data set for recovery. So, images after recovery has a smallest difference from original images with highest PSNR.

5. **Additional specifications:**

The almost Python codes are written by myself so this report does not have a great number references.

The report may not cover all techniques details and description so if you have any questions about the source code please contact me.

Before run the codes, you should guarantee the all necessary packet are installed in python execution environment. And inputted image name is extremely strict so please make sure you enter the correct files name. The project folder is generated by Spyder a kind of development environment so you can use the Spyder open the project and execute source codes.

References:

[i] *"基于 Python 二维离散余弦变换（DCT）及其反变换（IDCT）---程序对比"*. csdn.net. January 2018. Retrieved from https://blog.csdn.net/james_ray_murphy/article/details/79173388

[ii] *"基于 Python 二维离散余弦变换（DCT）及其反变换（IDCT）---程序对比"*. csdn.net. January 2018. Retrieved from https://blog.csdn.net/james_ray_murphy/article/details/79173388

[iii] *"gzip 压缩算法"*. csdn.net. July 2017. Retrieved from https://blog.csdn.net/hguisu/article/details/7795435