Image Processing Programming Assignment #3

In this assignment, our goal is to implement a basic encoder-decoder style JPEG codec. The code is done with Python programming language and is released on https://github.com/KennethYapWL/NCTU-IP-2020/tree/main/Project%203. The references are listed on the last page of this report.

The implementation should include multiple components below:

1. Block-based Discrete Cosine Transform (DCT)
2. Quantization of Discrete Cosine Transform (DCT) coefficients
3. Predictive coding between DC coefficients
4. Run-length coding of the AC coefficients
5. Chromatic subsampling
6. Huffman coding

This report is organized as follow:

→ Section 1 (Experimental Results): This part will show the result produced by the encoder and decoder.
→ Section 2 (Observations and Discussions): This part will give brief descriptions of the other experiment I have tried, also the further observations of section 1 will be discussed.
→ Section 3 (Code Analysis): This part will show all the code used in this assignment.
→ Section 4 (References): This part will list all the articles I have read.

Section 1: Experimental Results

This section is focused on showing the results produced by the encoder and decoder. The compressed image cannot be shown in JPEG format without inserting JFIF format information. Hence the result will be shown in two separate ways:

Note that the BMP testing image is obtained from the internet. (*ref[11]*)

A) **The compressed image produced by the encoder**, with inserting JFIF format information: bytes representations of the JPEG header (xFFD8), quantization tables used (xFFDB), size of the image (x FFC0), subsampling ratio, the Huffman table used (xFFC4), etc.

Figure 1. Original Image

Figure 2. Compressed Image

Based on the figures above, without checking the zoom-in details, there is difficult to obtain the differences between the original image and the compressed image. One of the differences can be found in the figures below.

Figure 3. Original Zoomed In Image

Figure 4. Compressed Zoomed-in Image

Notice that the details in the original zoomed-in image are smooth, while the compressed zoomed-in images consist many of “blocks”, which I think it’s probably the effect of applying DCT based block splitting. Besides, here is some evaluation of the compressed image:

Size	Original image: 51.2 MB Compressed image: 2.48 MB
SNR	1.4567
RMSE	31.3599
SSIM	0.5550

Note: the Huffman table used in this part is a little bit different from the standard Huffman table, more information can be found in the file “jpeg_encoder.py” in the provided GitHub. (Huffman table referred to *ref[4]*)

B) **The decompressed image produced by the decoder**, the encoder only inserted the bits representation of the size of the image, Huffman table used for DC and AC coefficients to the compressed image file. The decoder has then obtained this information to decompress the image.

Figure 5. Original Image

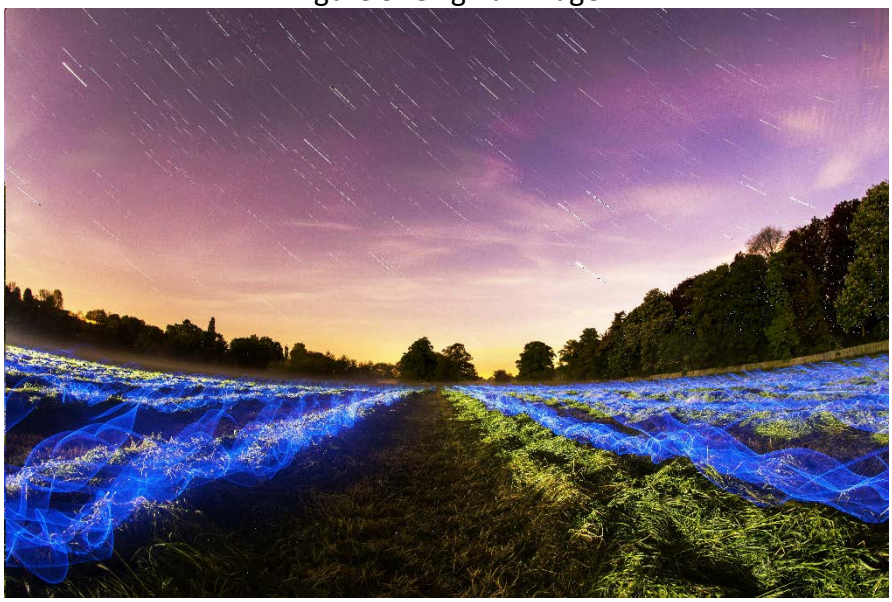

Figure 6. Decompressed Image

Similar to the result of part 1.A, the differences between these two images are hard to obtain without checking on zoom-in details.

Figure 7. Original Zoomed In Image

Figure 8. Decompressed Zoomed-in Image

Notice that the details in the original zoomed-in image are smooth, while the decompressed zoomed-in images consist many of black and white spots. Besides, here is some evaluation of the decompressed image:

Size	Original image: 51.2 MB Decompressed image: 51.2 MB
SNR	1.4771
RMSE	52.6667
SSIM	0.2772

Note: the Huffman encoding in this part is achieved by constructing a Huffman tree, more details can be found on the files “jpeg_encoder.py” and “jpeg_decoder.py” in the provided GitHub.

Section 2: Observations and Discussions

1. *The difference between self-implemented DCT and fftpack package DCT*

   I have implemented the DCT function to transform each split 8x8 blocks. The size of the testing image is 3456x5184, which implies there are 279936 blocks needed to be transformed.

   By experimenting with comparing the performance of using self-implemented DCT function and the DCT function from one of the Python package, fftpack, I have found that the fftpack DCT function is much outperformed:

   - Only a minute is needed to transform all 279936 blocks by using fftpack DCT function.
   - While about 7 minutes is needed to transform only 10000 blocks by using my self-implemented DCT function.

   By considering the efficiency of the compression process, as shown in the file "jpeg_encoder.py", I decided to use the fftpack DCT function, but the self-implemented DCT function remains in the file "utils.py".

2. *The result of decompression with and without predictive coding.*

   The predictive coding example I used for the DC coefficients, is called Delta coding (*ref[2]* & *ref[12]*). I have found that the performance of the decompressed image produced by the decoder is better when using delta encoded DC coefficients. The figures below shows the result:

Figure 9. Decompressed image without Delta encoded DC coefficients

Figure 10. Decompressed image with Delta encoded DC coefficients

Notice that there exists a black border on the left-hand side of the image in Figure 9, while Figure 10 does not, which is one of the differences between the decompressed image with and without Delta coding.
The experiment also shows that the Delta coding can help to improve the SSIM and RMSE of the decompressed image, while the SNR and the size of the images remain unchanged.

Metrics	With Delta Coding	Without Delta Coding
SNR	1.4771	1.4771
RMSE	**21.3207**	52.6667
SSIM	**0.6493**	0.2772

Note: the image in Figures 9 and 10 can be found on the files “recontructedImage.bmp” and “recontructedImage_Delta.bmp” in the provided GitHub respectively.

Section 3: Code Analysis

I think this implementation is much easier can be achieved by coding with Python compared to Matlab, hence the code this time are all in py file format. As shown in the provided Github, it consists of several files:

1. color_space_conversion.py (which I didn't use this time, but it's used to make conversion between RGB and YCbCr)
2. huffman.py (the file which stores the construction of Huffman tree, and standard Huffman table)
3. huffman2.py (the file which stores the Huffman table of ref[4])
4. jpeg_encoder.py
5. jpeg_decoder.py
6. metrics.py (the file which stores the SNR, RMSE, SSIM functions)
7. tables.py (the file which declares the tables used, eg. Quantization tables)
8. test.py (just a sample file for checking if the implemented function works well)
9. utils.py (the file which stores common functions)
10. write_file.py (the file which stores the writing jpeg file functions)

Note that the explanations of each part of the code are shown in the comment (font color in grey color).

1. color_space_conversion.py

```python
import numpy as np

def RGB_2_YCbCr(img):
    #ref https://zhuanlan.zhihu.com/p/88933905
    if len(img.shape) != 3 or img.shape[2] != 3:
        raise ValueError('the input image is not a rgb image')

    hg, wd, ch = img.shape
    img = img.astype(np.float32)
    coeffs = np.array([[ 0.257,  0.504,  0.098],
                       [-0.148, -0.291,  0.439],
                       [ 0.439, -0.368, -0.071]])

    offset = np.array([16, 128, 128])
    ycbcr = np.zeros(shape=[hg, wd, ch])

    for row in range(hg):
        for col in range(wd):
            ycbcr[row,col,:] = np.dot(coeffs, img[row,col,:]) + offset

    return ycbcr
```

```
def YCbCr_2_RGB(img):
    #ref https://zhuanlan.zhihu.com/p/88933905
    if len(img.shape) != 3 or img.shape[2] != 3:
        raise ValueError('the input image is not a ycbcr image')

    hg, wd, ch = img.shape
    img = img.astype(np.float32)
    coeffs = np.array([[ 0.257,  0.504,  0.098],
                       [-0.148, -0.291,  0.439],
                       [ 0.439, -0.368, -0.071]])

    offset = np.array([16, 128, 128])
    coeffs_inv = np.linalg.inv(coeffs)
    rgb = np.zeros(shape=[hg, wd, ch])

    for row in range(hg):
        for col in range(wd):
            rgb[row,col,:] = np.dot(coeffs_inv, img[row,col,:]) \
                           - np.dot(coeffs_inv, offset)
```

2. huffman.py

```
import numpy as np
import pandas as pd
import utils
from queue import PriorityQueue

#refs:
#1. https://github.com/ghallak/jpeg-python/blob/2fe1bd2244c3090543695b106866dfa0a3b48f6c/utils.py

#ref: ITU-T.81, Annex-K, Table-K.3, Page-149
HUFF_DICT_LUM_DC = dict()
path = 'table refs/Table_DC_lum.xlsx'
catecol, codecol = 'Category', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values
for cat, cw in zip(categ, code):
    HUFF_DICT_LUM_DC[int(cat)] = cw


#ref: ITU-T.81, Annex-K, Table-K.5, Page-150
HUFF_DICT_LUM_AC = dict()
path = 'table refs/Table_AC_lum.xlsx'
catecol, codecol = 'Run/Size', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    rs = cat.split('/')
    HUFF_DICT_LUM_AC[(int('0x'+rs[0],16), int('0x'+rs[1],16))] = cw
```

```
#ref: ITU-T.81, Annex-K, Table-K.4, Page-149
HUFF_DICT_CHROM_DC = dict()
path = 'table refs/Table_DC_chrom.xlsx'
catecol, codecol = 'Category', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    HUFF_DICT_CHROM_DC[int(cat)] = cw


#ref: ITU-T.81, Annex-K, Table-K.6, Page-154
HUFF_DICT_CHROM_AC = dict()
path = 'table refs/Table_AC_chrom.xlsx'
catecol, codecol = 'Run/Size', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    rs = cat.split('/')
    HUFF_DICT_CHROM_AC[(int('0x'+rs[0],16), int('0x'+rs[1],16))] = cw
```

```
def huffman_coding_standard_ac(ac_arr, BStream, component):
    if not component in ['lum', 'chrom']:
        raise ValueError((
            "component should be either 'lum' or 'chrom',"
            "but '{comp}' was found").format(comp=component))

    # Run Length Coding
    rlc_arr = utils.run_length_coding(ac_arr);
    if component == 'lum':
        huff_table = HUFF_DICT_LUM_AC
    else:
        huff_table = HUFF_DICT_CHROM_AC

    for rlc in rlc_arr:
        run, size = int(rlc[0]), utils.bits_length(int(rlc[1]))
        _bin = utils.int_to_binstr(int(rlc[1]))

        BStream.write([int(b) for b in (huff_table[(run, size)] + _bin)], bool)

def huffman_coding_standard_dc(dc, BStream, component):
    if not component in ['lum', 'chrom']:
        raise ValueError((
            "component should be either 'lum' or 'chrom',"
            "but '{comp}' was found").format(comp=component))

    if component == 'lum':
        huff_table = HUFF_DICT_LUM_DC
    else:
        huff_table = HUFF_DICT_CHROM_DC

    categ = utils.bits_length(int(dc))
    BStream.write([int(b) for b in (huff_table[categ] + utils.int_to_binstr(int(dc)))], bool)
```

```python
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
```

3. huffman2.py (Notes: This file stores the declaration of AC Huffman tables, which are too many contents, I will skip this here, but the declaration code is still available in the provided Github.)

```
import numpy as np
import tables
import utils
from queue import PriorityQueue

#refs:
#1. https://github.com/ghallak/jpeg-python/blob/2fe1bd2244c3090543695b106866dfa0a3b48f6c/utils.py

#The DC Hoffman coding table for luminance recommended by JPEG
DCLuminanceSizeToCode = [
    [1,1,0],              #0 EOB
    [1,0,1],            #1
    [0,1,1],            #2
    [0,1,0],            #3
    [0,0,0],            #4
    [0,0,1],            #5
    [1,0,0],            #6
    [1,1,1,0],        #7
    [1,1,1,1,0],      #8
    [1,1,1,1,1,0],    #9
    [1,1,1,1,1,1,0],  #10 0A
    [1,1,1,1,1,1,1,0] #11 0B
]



#The DC Hoffman coding table for chrominance recommended by JPEG
DCChrominanceSizeToCode = [
    [0,1],                #0 EOB
    [0,0],                #1
    [1,0,0],              #2
    [1,0,1],              #3
    [1,1,0,0],            #4
    [1,1,0,1],            #5
    [1,1,1,0],            #6
    [1,1,1,1,0],          #7
    [1,1,1,1,1,0],     #8
    [1,1,1,1,1,1,0],   #9
    [1,1,1,1,1,1,1,0], #10 0A
    [1,1,1,1,1,1,1,1,0]#11 0B
]
```

```
def huffman_coding_standard_ac(ac_arr, BStream, component):
    if not component in ['lum', 'chrom']:
        raise ValueError((
            "component should be either 'lum' or 'chrom',"
            "but '{comp}' was found").format(comp=component))

    i = 0
    maxI = np.size(ac_arr)
    while 1:
        if(i==maxI):
            break
        run = 0

        # check if rest of ACArray are all zero. If so, just write EOB and return
        j = i
        while 1:
            if(ac_arr[j]!=0):
                break
            if(j==maxI - 1):
                if (component == 'lum'):
                    BStream.write(ACLuminanceSizeToCode['00'], bool)  # EO
                else:
                    BStream.write(ACChrominanceToCode['00'], bool)
                return
            j = j + 1



        while 1:
            if(ac_arr[i]!=0 or i==maxI - 1 or run==15):
                break
            else:
                run = run + 1
                i = i + 1

        value = int(ac_arr[i])

        if(value==0 and run!=15):
            break # Rest of the components are zeros therefore we simply put the EOB to signify this fact

        size = int(value).bit_length()
```

```
        runSizeStr = str.upper(str(hex(run))[2:]) + str.upper(str(hex(size))[2:])

        if (component == 'lum'):
            BStream.write(ACLuminanceSizeToCode[runSizeStr], bool)
        else:
            BStream.write(ACChrominanceToCode[runSizeStr], bool)


        if(value<=0):# if value==0, codeList = [], (SIZE,VALUE)=(SIZE,[])=EOB
            codeList = list(bin(value)[3:])
            for k in range(len(codeList)):
                if (codeList[k] == '0'):
                    codeList[k] = 1
                else:
                    codeList[k] = 0
        else:
            codeList = list(bin(value)[2:])
            for k in range(len(codeList)):
                if (codeList[k] == '0'):
                    codeList[k] = 0
                else:
                    codeList[k] = 1
        BStream.write(codeList, bool)
        i = i + 1
```

```
def huffman_coding_standard_dc(dc, BStream, component):
    dc = int(dc)
    if not component in ['lum', 'chrom']:
        raise ValueError((
            "component should be either 'lum' or 'chrom',"
            "but '{comp}' was found").format(comp=component))

    boolList = []
    size = int(dc).bit_length()
    if(component == 'lum'):
        boolList = boolList + DCLuminanceSizeToCode[size]
    else:
        boolList = boolList + DCChrominanceSizeToCode[size]
    if(dc <= 0):
        codeList = list(bin(dc)[3:])
        for i in range(len(codeList)):
            if (codeList[i] == '0'):
                codeList[i] = 1
            else:
                codeList[i] = 0
    else:
        codeList = list(bin(dc)[2:])
        for i in range(len(codeList)):
            if (codeList[i] == '0'):
                codeList[i] = 0
            else:
                codeList[i] = 1
    boolList = boolList + codeList
    BStream.write(boolList, bool)
```

4. jpeg_encoder.py

```
import numpy as np
from bitstream import BitStream
from PIL import Image
from scipy import fftpack
import sys
from datetime import datetime

import utils
import huffman
import huffman2
import write_file
import metrics


#ref:
#1. https://koushtav.me//jpeg/tutorial/2017/11/25/lets-write-a-simple-jpeg-library-part-1
#2. https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/
#3. https://boisgera.github.io/bitstream/
#4. https://github.com/fangwei123456/python-jpeg-encoder
#5. https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-jpeg-librar
#6. https://ithelp.ithome.com.tw/articles/10230274
```

```python
def main():
#    if(len(sys.argv)!=3):
#        print('inputBMPFileName outputJPEGFilename')
#        print('example:')
#        print('./lena.bmp ./output.jpg')
#        return

    is_include_JFIF_format = False
    srcFileDir = 'img/img1.bmp'#sys.argv[1]
    output_Dir = 'compressed/JFIF_out.jpg'#sys.argv[2]
    if is_include_JFIF_format == False:
        output_Dir = 'compressed/noJFIF_out.jpg'


    print("{}, start compressing.".format(datetime.now()))
    srcImage = Image.open(srcFileDir)
    imgHg, imgWd = srcImage.size

    padd_y, padd_x = 0, 0
    # make sure the width and height of the image are the multiple of 8
    if imgHg % 8 != 0:
        padd_y = 8 - (imgHg % 8)
    if imgWd % 8 != 0:
        padd_x = 8 - (imgWd % 8)

    imgMatr = np.asarray(srcImage)
    imgHg, imgWd, imgCh = imgMatr.shape
    imgMatr = np.asarray([np.pad(chl, pad_width=((0,padd_y),(0,padd_x)), mode='symmetric') for chl in imgMatr])
    imgHg, imgWd, imgCh = imgMatr.shape

    # Step 1: Colour space transformation from RGB to Y-Cb-Cr
    #YCbCr = color_space.RGB_2_YCbCr(imgMatr)
    yImage,uImage,vImage = Image.fromarray(imgMatr).convert('YCbCr').split()

    yImageMatrix = np.asarray(yImage).astype(int)
    uImageMatrix = np.asarray(uImage).astype(int)
    vImageMatrix = np.asarray(vImage).astype(int)
```

```python
    YCbCr = np.array([yImageMatrix, uImageMatrix, vImageMatrix])

    # Step 2: Chroma Subsampling
    #TODO

    # Step 3: Level Shifting
    YCbCr = YCbCr - 128

    # Step 4: Discrete Cosine Transform (DCT) of Minimum Coded Units (MCU)
    blockTotal = imgHg // 8 * imgWd // 8 # How many blocks needed to split the image
    print('block Total:' + str(blockTotal))
    components = ['lum', 'chrom', 'chrom']
    block_idx = 0
    DCs, deltaDCs = np.zeros([blockTotal, imgCh]), np.zeros([blockTotal, imgCh])
    ACs = np.empty((blockTotal, 63, imgCh), dtype=np.int32)
    BStream = BitStream() # see details on ref[3]
    for y in range(0, imgHg, 8):
        for x in range(0, imgWd, 8):
            for ch in range(imgCh):
                block = YCbCr[ch][y:y+8, x:x+8] # 8x8 Block Splitting

                #block = utils.dct_2D(block) # DCT  ==> this is so slow, use fftpack better
                block = utils.fftpack_dct_2d(block) # DCT

                # Quantization
                block = utils.quantization(block, component=components[ch])
                # Zigzag Traversal
                zz = utils.zigzag_traversal(block)

                # Step 5: Delta Coding (Predictive Coding)
                DCs[block_idx, ch] = zz[0]
                if block_idx == 0:
                    deltaDCs[block_idx, ch] = DCs[block_idx, ch]
                else:
                    deltaDCs[block_idx, ch] = DCs[block_idx, ch] - DCs[block_idx - 1, ch]
```

```
            # Except for the first element of zz, the remains are AC coefficients
            ACs[block_idx, :, ch] = zz[1:]
            if is_include_JFIF_format == True:
                # Step 6: Huffman Coding for DC
                huffman2.huffman_coding_standard_dc(deltaDCs[block_idx, ch],
                                                    BStream,
                                                    component=components[ch])

                # Step 7: Run Length Coding and Huffman Coding for AC
                huffman2.huffman_coding_standard_ac(ACs[block_idx, :, ch],
                                                    BStream,
                                                    component=components[ch])




        block_idx += 1
        if block_idx % 100000 == 0:
            print(str(block_idx) + ' blocks are done encoding! {}'.format(datetime.now()))


print("{}, writing file...".format(datetime.now()))
if is_include_JFIF_format == True:
    write_file.write_jpgfile_JFIF(output_Dir, BStream, imgHg, imgWd)

else:
    H_DC_Y = huffman.HuffmanTree(np.vectorize(utils.bits_length)(deltaDCs[:, 0]))
    H_DC_C = huffman.HuffmanTree(np.vectorize(utils.bits_length)(deltaDCs[:, 1:].flat))
    H_AC_Y = huffman.HuffmanTree(
            utils.flatten(utils.run_length_encode_binstr(ACs[i, :, 0])[0] for i in range(blockTotal)))
    H_AC_C = huffman.HuffmanTree(
            utils.flatten(utils.run_length_encode_binstr(ACs[i, :, j])[0] for i in range(blockTotal) for j in [1, 2]))

    tables = {'dc_y': H_DC_Y.value_to_bitstring_table(),
              'ac_y': H_AC_Y.value_to_bitstring_table(),
              'dc_c': H_DC_C.value_to_bitstring_table(),
              'ac_c': H_AC_C.value_to_bitstring_table()}

    write_file.write_jpgfile_without_JFIF(output_Dir, imgHg, imgWd, DCs, ACs, blockTotal, tables)
```

```
print("{}, finish.".format(datetime.now()))

if is_include_JFIF_format == True:
    # Performance
    compressedImage = Image.open(output_Dir)
    compressedImage = np.array(compressedImage)
    srcImage = np.array(srcImage)

    print('SNR of image: ', end='')
    print(metrics.SNR(compressedImage).mean())

    print('RMSE of original image and the reconstruted image: ', end='')
    print(metrics.RMSE(srcImage, compressedImage))

    print('SSIM of original image and the reconstruted image: ', end='')
    print(metrics.SSIM(srcImage, compressedImage))



if __name__ == '__main__':
    main()
```

5. jpeg_decoder.py

```python
import numpy as np
import utils
from PIL import Image
from datetime import datetime
import metrics

IMG_SIZE_BITS = 16
TABLE_SIZE_BITS = 16
BLOCKS_COUNT_BITS = 32

DC_CODE_LENGTH_BITS = 4
CATEGORY_BITS = 4

AC_CODE_LENGTH_BITS = 8
RUN_LENGTH_BITS = 4
SIZE_BITS = 4


def read_dc_table(jpgFile):
    table = dict()

    table_size = int(jpgFile.read(TABLE_SIZE_BITS), 2)
    for _ in range(table_size):
        category = int(jpgFile.read(CATEGORY_BITS), 2)
        code_length = int(jpgFile.read(DC_CODE_LENGTH_BITS), 2)
        code = jpgFile.read(code_length)
        table[code] = category
    return table
```

```python
def read_ac_table(jpgFile):
    table = dict()

    table_size = int(jpgFile.read(TABLE_SIZE_BITS), 2)
    for _ in range(table_size):
        run_length = int(jpgFile.read(RUN_LENGTH_BITS), 2)
        size = int(jpgFile.read(SIZE_BITS), 2)
        code_length = int(jpgFile.read(AC_CODE_LENGTH_BITS), 2)
        code = jpgFile.read(code_length)
        table[code] = (run_length, size)
    return table

def read_huffman_code(jpgFile, table):
    prefix = ''
    # TODO: break the loop if __read_char is not returing new char
    while prefix not in table:
        prefix += jpgFile.read(1)
    return table[prefix]

def read_int(jpgFile, size):
    if size == 0:
        return 0

    # the most significant bit indicates the sign of the number
    bin_num = jpgFile.read(size)
    if bin_num[0] == '1':
        return int(bin_num, 2)
    else:
        return int(utils.binstr_flip(bin_num), 2) * -1
```

```
def decode_data(jpgDir):
    imgHg, imgWd, blockTotal = 0, 0, 0
    jpgFile = open(jpgDir, "r")

    imgHg = int(jpgFile.read(IMG_SIZE_BITS), 2)
    imgWd = int(jpgFile.read(IMG_SIZE_BITS), 2)
    print("size %ix%i" % (imgHg,  imgWd))

    tables = dict()
    for table_name in ['dc_y', 'ac_y', 'dc_c', 'ac_c']:
        if 'dc' in table_name:
            tables[table_name] = read_dc_table(jpgFile)
        else:
            tables[table_name] = read_ac_table(jpgFile)

    blockTotal = imgHg // 8 * imgWd // 8 # How many blocks needed to split the image
    dc = np.empty((blockTotal, 3), dtype=np.int32)
    ac = np.empty((blockTotal, 63, 3), dtype=np.int32)

    for block_index in range(blockTotal):
        for component in range(3):
            dc_table = tables['dc_y'] if component == 0 else tables['dc_c']
            ac_table = tables['ac_y'] if component == 0 else tables['ac_c']

            category = read_huffman_code(jpgFile, dc_table)
            dc[block_index, component] = read_int(jpgFile, category)

            cells_count = 0

            # TODO: try to make reading AC coefficients better
            while cells_count < 63:
                run_length, size = read_huffman_code(jpgFile, ac_table)
```

```
                if (run_length, size) == (0, 0):
                    while cells_count < 63:
                        ac[block_index, cells_count, component] = 0
                        cells_count += 1
                else:
                    for i in range(run_length):
                        ac[block_index, cells_count, component] = 0
                        cells_count += 1
                    if size == 0:
                        ac[block_index, cells_count, component] = 0
                    else:
                        value = read_int(jpgFile,  size)
                        ac[block_index, cells_count, component] = value
                    cells_count += 1


    jpgFile.close()
    return dc, ac, imgHg,  imgWd, blockTotal
```

```python
def main():
    jpgDir = 'compressed/noJFIF_out.jpg'
    dc, ac, imgHg,  imgWd, blockTotal = decode_data(jpgDir)

    image = np.empty((imgHg, imgWd, 3), dtype=np.uint8)
    blockTotal = imgHg // 8 * imgWd // 8 # How many blocks needed to split the image
    print('block total: ' + str(blockTotal))
    print("{}, start decompressing.".format(datetime.now()))

    block_index = 0
    for y in range(0, imgHg, 8):
        for x in range(0, imgWd, 8):
            for ch in range(3):
                zigzag = [dc[block_index, ch]] + list(ac[block_index, :, ch])
                quant_matrix = utils.zigzag_to_block(zigzag)
                dct_matrix = utils.de_quantization(quant_matrix, 'lum' if ch == 0 else 'chrom')
                block = utils.fftpack_idct_2d(dct_matrix)
                image[y:y+8, x:x+8, ch] = block + 128

            block_index += 1
            if block_index % 100000 == 0:
                print(str(block_index) + ' blocks are done decoding! {}'.format(datetime.now()))

    print("{}, finish decompressing.".format(datetime.now()))
    image = Image.fromarray(image, 'YCbCr')
    image = image.convert('RGB')
    image.show()
    image.save('recontructedImage2.bmp')


    # Performance
    srcFileDir = 'img/img1.bmp'
    oriImage = Image.open(srcFileDir)
    image = np.array(image)
    oriImage = np.array(oriImage)

    print('SNR of image: ', end='')
    print(metrics.SNR(image).mean())
    print('RMSE of original image and the reconstruted image: ', end='')
    print(metrics.RMSE(oriImage, image))

    print('SSIM of original image and the reconstruted image: ', end='')
    print(metrics.SSIM(oriImage, image))



if __name__ == "__main__":
    main()
```

6. metrics.py

```python
import numpy as np
import math
from skimage.metrics import structural_similarity as ssim
import scipy


#ref:
#1. https://github.com/tarikd/python-compare-two-images/blob/master/compare.py
#2. https://stackoverflow.com/questions/60057795/attributeerror-module-scipy-stats-has-no-attribute-signaltonoise

def MSE(oriImage, recImage):
    err = np.sum((oriImage.astype("float") - recImage.astype("float")) ** 2)
    err /= float(oriImage.shape[0] * oriImage.shape[1])

    return err


def SSIM(oriImage, recImage):
    return ssim(np.moveaxis(oriImage, 1, -3), np.moveaxis(recImage, 1, -3), multichannel=True)


def RMSE(oriImage, recImage):
    return math.sqrt(MSE(oriImage, recImage))


def SNR(image, axis=0, ddof=0):
    image = np.asanyarray(image)
    mean = image.mean(axis)
    std = image.std(axis=axis, ddof=ddof)
    return np.where(std == 0, 0, mean/std)
```

7. tables.py

```python
import numpy as np
import pandas as pd


#ref: ITU-T.81, Annex-K, Table-K.3, Page-149
HUFF_DICT_LUM_DC = dict()
path = 'table refs/Table_DC_lum.xlsx'
catecol, codecol = 'Category', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values
for cat, cw in zip(categ, code):
    HUFF_DICT_LUM_DC[int(cat)] = cw



#ref: ITU-T.81, Annex-K, Table-K.5, Page-150
HUFF_DICT_LUM_AC = dict()
path = 'table refs/Table_AC_lum.xlsx'
catecol, codecol = 'Run/Size', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    rs = cat.split('/')
    HUFF_DICT_LUM_AC[(int('0x'+rs[0],16), int('0x'+rs[1],16))] = cw
```

```
#ref: ITU-T.81, Annex-K, Table-K.4, Page-149
HUFF_DICT_CHROM_DC = dict()
path = 'table refs/Table_DC_chrom.xlsx'
catecol, codecol = 'Category', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    HUFF_DICT_CHROM_DC[int(cat)] = cw



#ref: ITU-T.81, Annex-K, Table-K.6, Page-154
HUFF_DICT_CHROM_AC = dict()
path = 'table refs/Table_AC_chrom.xlsx'
catecol, codecol = 'Run/Size', 'Code Word'
df = pd.read_excel(path, dtype=str)
categ, code = df[catecol].values, df[codecol].values

for cat, cw in zip(categ, code):
    rs = cat.split('/')
    HUFF_DICT_CHROM_AC[(int('0x'+rs[0],16), int('0x'+rs[1],16))] = cw
```

```
def quantization_table(component):
    if component == 'lum':
        #ref: ITU-T.81, Annex-K, Table-K.1, Page-143
        quan_matr = 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]])

    elif component == 'chrom':
        #ref: ITU-T.81, Annex-K, Table-K.2, Page-143
        quan_matr = 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]])

    else:
        raise ValueError((
            "component should be either 'lum' or 'chrom', "
            "but '{comp}' was found").format(comp=component))

    return quan_matr
```

```
def huffman_table(component, coef_type):
    if component == 'lum' and coef_type == 'DC':
        huff_dict = HUFF_DICT_LUM_DC
    elif component == 'chrom' and coef_type == 'DC':
        huff_dict = HUFF_DICT_CHROM_DC
    elif component == 'lum' and coef_type == 'AC':
        huff_dict = HUFF_DICT_LUM_AC
    elif component == 'chrom' and coef_type == 'AC':
        huff_dict = HUFF_DICT_CHROM_AC
    else:
        raise ValueError((
            "component should be either 'lum' or 'chrom',"
            "and coef_type should be either 'DC' or 'AC', "
            "but '{comp}', '{coef}' was found")
            .format(comp=component, coef=coef_type))


    return huff_dict


def zigzag_table():
    zigzagOrder = np.array([0,1,8,16,9,2,3,10,17,24,32,25,18,11,4,5,12,19,
                            26,33,40,48,41,34,27,20,13,6,7,14,21,28,35,42,
                            49,56,57,50,43,36,29,22,15,23,30,37,44,51,58,
                            59,52,45,38,31,39,46,53,60,61,54,47,55,62,63])
    return zigzagOrder
```

8. utils.py

```
import numpy as np
import tables
import math
from scipy import fftpack

#refs:
#1. https://www.includehelp.com/python/find-number-of-bits-necessary-to-represent-an-integer-in-binary.aspx
#2. https://github.com/ghallak/jpeg-python/blob/2fe1bd2244c3090543695b106866dfa0a3b48f6c/utils.py

def bits_length(num):
    return abs(int(num)).bit_length()


def binstr_flip(binstr):
    #check if binstr is a binary string
    if not set(binstr).issubset('01'):
        raise ValueError("binstr should have only '0's and '1's")
    return ''.join(map(lambda c: '0' if c == '1' else '1', binstr))


def int_to_binstr(num):
    if num == 0:
        return ''

    binstr = bin(abs(num))[2:]
    #change every 0 to 1 and vice verse when n is negative
    return binstr if num > 0 else binstr_flip(binstr)

def uint_to_binstr(number, size):
    return bin(number)[2:][-size:].zfill(size)

def hex_to_bytes(hexstr):
    return bytes.fromhex(hexstr)
```

```
def fftpack_dct_2d(block):
    return fftpack.dct(fftpack.dct(block, norm='ortho').T, norm='ortho').T

def fftpack_idct_2d(block):
    return fftpack.idct(fftpack.idct(block.T, norm='ortho').T, norm='ortho')


def dct_2D(block):
    if len(block.shape) != 2:
        raise ValueError('the input block should be 2D dimension')

    hg, wd = block.shape
    dct = np.zeros([hg, wd])

    for v in range(hg):
        for u in range(wd):
            Sum = 0
            Cv = (1/math.sqrt(2) if v == 0 else 1 )
            Cu = (1/math.sqrt(2) if u == 0 else 1 )

            for y in range(hg):
                for x in range(wd):
                    Sum += block[y, x] * math.cos((2*x + 1)*u*math.pi / 16) \
                         * math.cos((2*y + 1)*v*math.pi / 16)

            Sum *= 1/4 * Cv * Cu
            dct[v,u] = Sum

    return dct


def quantization(block, component):
    quan_table = tables.quantization_table(component)
    return np.divide(block, quan_table).round().astype('int32')

def de_quantization(block, component):
    quan_table = tables.quantization_table(component)
    return np.multiply(block, quan_table).astype('int32')
```

```python
def zigzag_traversal(block):
    zz_order = tables.zigzag_table()
    return block.reshape([64])[zz_order]

def zigzag_to_block(sequence):
    block = np.array([[ 1, 2, 6, 7,15,16,28,29],
                      [ 3, 5, 8,14,17,27,30,43],
                      [ 4, 9,13,18,26,31,42,44],
                      [10,12,19,25,32,41,45,54],
                      [11,20,24,33,40,46,53,55],
                      [21,23,34,39,47,52,56,61],
                      [22,35,38,48,51,57,60,62],
                      [36,37,49,50,58,59,63,64]]) - 1

    hg, wd = block.shape
    for row in range(hg):
        for col in range(wd):
            block[row][col] = sequence[block[row][col]]

    return block


def run_length_coding(sequence):
    sequence = sequence.astype('int32')
    zero_count = 0
    rl = []
    for num in sequence:
        if num == 0:
            zero_count += 1
        else:
            if zero_count > 15:
                zero_count = 15
            rl.append([zero_count, num])
            zero_count = 0

    #End of bits
    rl.append([0,0])
    return np.array(rl)
```

```
def run_length_encode_binstr(arr):
    # determine where the sequence is ending prematurely
    last_nonzero = -1
    for i, elem in enumerate(arr):
        if elem != 0:
            last_nonzero = i

    # each symbol is a (RUNLENGTH, SIZE) tuple
    symbols = []

    # values are binary representations of array elements using SIZE bits
    values = []

    run_length = 0

    for i, elem in enumerate(arr):
        if i > last_nonzero:
            symbols.append((0, 0))
            values.append(int_to_binstr(0))
            break
        elif elem == 0 and run_length < 15:
            run_length += 1
        else:
            size = bits_length(elem)
            symbols.append((run_length, size))
            values.append(int_to_binstr(elem))
            run_length = 0
    return symbols, values


def flatten(lst):
    return [item for sublist in lst for item in sublist]
```

9. write_file.py

```
import numpy as np
import utils
import tables
import os

def write_jpgfile_JFIF(output_Dir, BStream, imgHg, imgWd):
    try:
        jpegFile = open(output_Dir, 'wb+')
    except FileNotFoundError as e:
        raise FileNotFoundError("No such directory: {}"
                                .format(os.path.dirname(output_Dir))) from e

    # write JPEG Header => FFD8: Start of Image segment (SOI); FFE0: JPEG/JFIF Image segment (APP-
    # meaning of FFD8 FFE0 0010 4A46494600 0101 00 0001 0001 00 00 ? refer to:
    #https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-jpeg-library-par
    jpegFile.write(utils.hex_to_bytes('FFD8FFE000104A4649460001010000010001 0000'))

    # write Y Quantization Table => FFDB: Define Quantization Table segment (DQT)
    # meaning of FFDB 0043 00? refer to:
    #https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-jpeg-library-par
    jpegFile.write(utils.hex_to_bytes('FFDB004300'))
    lum_tbl = tables.quantization_table(component='lum').reshape([64])
    jpegFile.write(bytes(lum_tbl.tolist()))

    # write Cb/Cr Quantization Table => FFDB: Define Quantization Table segment (DQT)
    jpegFile.write(utils.hex_to_bytes('FFDB004301'))
    chrom_tbl = tables.quantization_table(component='chrom').reshape([64])
    jpegFile.write(bytes(chrom_tbl.tolist()))

    # write height and width of img => FFC0: Start of Frame-0 (SOF-0)
    # meaning of FFC0 0011 08 ? refer to:
    # https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-jpeg-library-pa
    jpegFile.write(utils.hex_to_bytes('FFC0001108'))
    hHex = hex(imgHg)[2:]
    while len(hHex) < 4:
        hHex = '0' + hHex
```

```
    jpegFile.write(utils.hex_to_bytes(hHex))

    wHex = hex(imgWd)[2:]
    while len(wHex) < 4:
        wHex = '0' + wHex

    jpegFile.write(utils.hex_to_bytes(wHex))

    # write Subsampling
    jpegFile.write(utils.hex_to_bytes('03011100021101031101'))

    # write Huffman Table => FFC4: Define Huffman Table segment (DHT)
    # FFC4 01A2 00 00000701010101010000000000000000
    # https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-j
    jpegFile.write(utils.hex_to_bytes('FFC401A20000000701010101010000000000000'
                                      + '00004050302060100070809 0A0B010002020301'
                                      + '0101010100000000000000010002030405060 70'
                                      + '8090A0B10000201030302040206070304020602'
                                      + '7301020311040005211231415106136122718 11'
                                      + '43291A10715B14223C152D1E1331662F0247282'
                                      + 'F12543345392A2B26373C235442793A3B336175'
                                      + '46474C3D2E2082683090A181984944546A4B456'
                                      + 'D355281AF2E3F3C4D4E4F465758595A5B5C5D5E'
                                      + '5F566768696A6B6C6D6E6F63747576777879 7A7'
                                      + 'B7C7D7E7F738485868788898A8B8C8D8E8F8293'
                                      + '94959697989 99A9B9C9D9E9F92A3A4A5A6A7A8A'
                                      + '9AAABACADAEAFA11000202010203050405060'
                                      + '40803036D0100021103042112314105511361 22'
                                      + '0671819132A1B1F014C1D1E1234215526272F13'
                                      + '32434382169 25325A263B2C20773D235E24483'
                                      + '17549308090A18192636451A2764745537F2A3B'
                                      + '3C32829D3E3F38494A4B4C4D4E4F465758595A5'
                                      + 'B5C5D5E5F5465666768696A6B6C6D6E6F647576'
                                      + '7778797A7B7C7D7E7F738485868788898A8B8C8'
                                      + 'D8E8F83949596979899 9A9B9C9D9E9F92A3A4A5'
                                      + 'A6A7A8A9AAABACADAEAFA'))
```

```
    # Start of Scan
    # Y_DC => Y_AC => CbDC => CbAC => CrDC => CrAC
    bit_len = BStream.__len__()
    num_of_fill = 8 - bit_len % 8
    if num_of_fill != 0:
        BStream.write(np.ones([num_of_fill]).tolist(), bool)

    # FF DA 00 0C 03 01 00 02 11 03 11 00 3F 00
    # => FFDA: Start of Scan segment (SOS)
    jpegFile.write(utils.hex_to_bytes('FFDA000C03010002110311003F00'))

    # write encoded data
    scanBytes = BStream.read(bytes)
    for sbytes in scanBytes:
        jpegFile.write(bytes([sbytes]))
        if sbytes == 255:
            jpegFile.write(bytes([0])) # FF to FF 00

    # write End Symbol => FFD9:  End of Image segment (EOI)
    jpegFile.write(utils.hex_to_bytes('FFD9')) # FF D9
    jpegFile.close()
```

```
def write_jpgfile_without_JFIF(output_Dir, imgHg, imgWd, dc, ac, blocks_count, tables):
    try:
        jpegFile = open(output_Dir, 'w')
    except FileNotFoundError as e:
        raise FileNotFoundError("No such directory: {}"
                                .format(os.path.dirname(output_Dir))) from e

    print('writing img size: %ix%i' % (imgHg,  imgWd))
    jpegFile.write(utils.uint_to_binstr(imgHg, 16))
    jpegFile.write(utils.uint_to_binstr(imgWd, 16))
    for table_name in ['dc_y', 'ac_y', 'dc_c', 'ac_c']:

        # 16 bits for 'table_size'
        jpegFile.write(utils.uint_to_binstr(len(tables[table_name]), 16))

        for key, value in tables[table_name].items():
            if table_name in {'dc_y', 'dc_c'}:
                # 4 bits for the 'category'
                # 4 bits for 'code_length'
                # 'code_length' bits for 'huffman_code'
                jpegFile.write(utils.uint_to_binstr(key, 4))
                jpegFile.write(utils.uint_to_binstr(len(value), 4))
                jpegFile.write(value)
            else:
                # 4 bits for 'run_length'
                # 4 bits for 'size'
                # 8 bits for 'code_length'
                # 'code_length' bits for 'huffman_code'
                jpegFile.write(utils.uint_to_binstr(key[0], 4))
                jpegFile.write(utils.uint_to_binstr(key[1], 4))
                jpegFile.write(utils.uint_to_binstr(len(value), 8))
                jpegFile.write(value)
```

```
    # 32 bits for 'blocks_count'
    jpegFile.write(utils.uint_to_binstr(blocks_count, 32))

    for b in range(blocks_count):
        for c in range(3):
            category = utils.bits_length(dc[b, c])
            symbols, values = utils.run_length_encode_binstr(ac[b, :, c])

            dc_table = tables['dc_y'] if c == 0 else tables['dc_c']
            ac_table = tables['ac_y'] if c == 0 else tables['ac_c']

            jpegFile.write(dc_table[category])
            jpegFile.write(utils.int_to_binstr(int(dc[b, c])))

            for i in range(len(symbols)):
                jpegFile.write(ac_table[tuple(symbols[i])])
                jpegFile.write(values[i])

    jpegFile.close()
```

Section 4: References

1. https://koushtav.me//jpeg/tutorial/2017/11/25/lets-write-a-simple-jpeg-library-part-1
2. https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/
3. https://boisgera.github.io/bitstream/
4. https://github.com/fangwei123456/python-jpeg-encoder
5. https://koushtav.me/jpeg/tutorial/c++/decoder/2019/03/02/lets-write-a-simple-jpeg-library-part-2/#the-jfif-file-format
6. https://ithelp.ithome.com.tw/articles/10230274
7. https://www.includehelp.com/python/find-number-of-bits-necessary-to-represent-an-integer-in-binary.aspx
8. https://github.com/ghallak/jpeg-python/blob/2fe1bd2244c3090543695b106866dfa0a3b48f6c/utils.py
9. https://github.com/tarikd/python-compare-two-images/blob/master/compare.py
10. https://stackoverflow.com/questions/60057795/attributeerror-module-scipy-stats-has-no-attribute-signaltonoise
11. https://filesamples.com/formats/bmp
12. https://www.dspguide.com/ch27/4.htm