在作業三中，為了要更好的了解 JPEG 壓縮演算法的原理與細節，我們實作了整個 JPEG 壓縮與解壓縮的步驟。

JPEG 是一個針對相片或影像的失真壓縮方法，演算法是由多個步驟所組成，每個步驟也有許多細節可以做修改或者是給予不同的參數設定。JPEG 可以分為編碼 (壓縮)、解碼(解壓縮)。編碼的部分主要由以下步驟所組成:

1. Color space transformation
2. Downsampling
3. Block splitting
4. Discrete cosine transform
5. Quantization
6. Entropy coding

解碼的部分主要由以下步驟所組成:

1. inverse Entropy coding
2. inverse Quantization
3. inverse Discrete cosine transform
4. inverse Block splitting
5. inverse Downsampling
6. inverse Color space transformation

基本上解碼的部分，和編碼的順序是相反的，也就是說我們要復原成原本做操作之前的狀態，以下我會針對上述各個步驟做出解釋和我是如何實作該功能的。

Color space transformation:

在這邊我們需要從原本 RGB 色彩模型的圖片轉換至 YCbCr 色彩模型，RGB 色彩模型利用三個 channel 分別表示紅、綠、藍的數值，並用這三個 channel 來組合成一個 pixel 的顏色。而 YCbCr 色彩模型則是利用三個 channel 來表示亮度、藍色色度、紅色色度。這樣的色彩模型會給我們後續的步驟提供許多好處，算是 JPEG 演算法的關鍵步驟之一。

而這個轉換本身是一個無失真的轉換，也就是說我們依樣可以從 YCbCr 轉換成 RGB，且不會有任何的損失，這兩個轉換可以直接套用特定的公式，詳細的實作如圖一所示。

```python
def color_space_transformation(self):
    # formula
    # Y = 0.299R + 0.587G + 0.114B
    # Cb = 128 - 0.1687R - 0.3313G + 0.5B
    # Cr = 128 + 0.5R - 0.4187G - 0.0813B
    rgb = self.data
    YCbCr = np.empty(rgb.shape)
    for y in range(self.height):
        for x in range(self.width):
            R, G, B = rgb[y,x]
            Y = 0.299 * R + 0.587 * G + 0.114 * B
            Cb = 128 - 0.168736 * R - 0.331264 * G + 0.5 * B
            Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B
            YCbCr[y,x] = np.array([Y,Cb,Cr])
    self.data = YCbCr
```

圖一，color space transformation 實作。

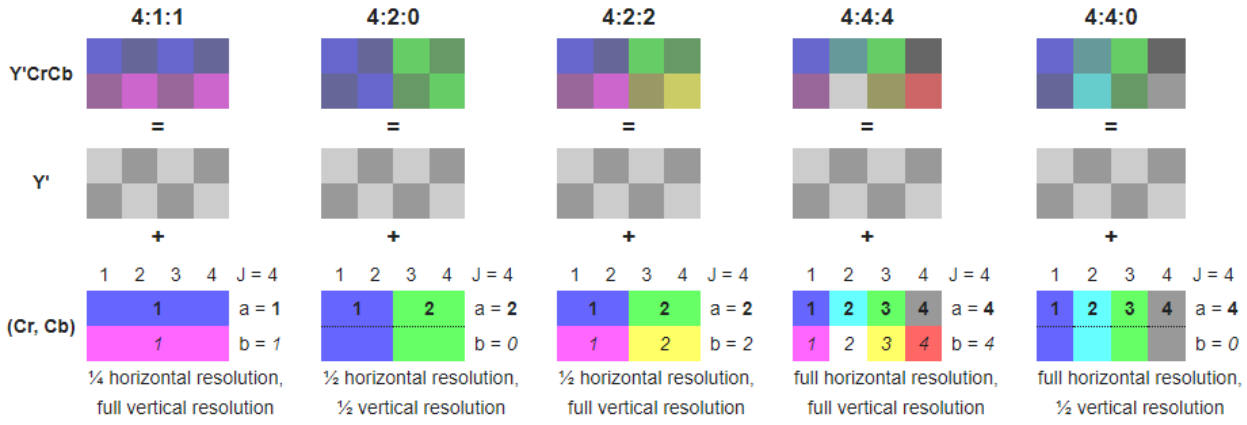
Downsampling:

而在 downspaling 的部分，我們可以利用 YCbCr 模型和人類眼睛視覺的特性，刪減一些對眼睛不敏感的資訊，人類的眼睛對 Cb、Cr 所提供的資訊較不敏感，因此針對這兩個 channel 做操作。YCbCr 的格式為 j:a:b，j 代表著水平的取樣個數，通常會為 4。a 為 j 個像素第一列色度的抽樣數目。b 為為 j 個像素第二列色度的額外抽樣數目。而 YCbCr 有以下形式 4:4:4、4:2:2、4:2:0，分別對應要刪減的類型和程度。

4:4:4 : 在這個格式中，我們保留原本的色彩模型，不刪減任何資料。

4:2:2 : 在這個格式中，我們將水平的取樣率改為 1/2。垂直的完整保留。

4:2:0 : 在這個格式中，我們將水平的取樣率和垂直的取樣率皆改為 1/2。



```
def downsampling(self):
    if self.mode == "444":
        pass
    elif self.mode == "422":
        YCbCr = self.data[:,:,1:]
        #print(YCbCr)
        for y in range(self.height):
            for x in range(1,self.width,2):
                YCbCr[y,x] =YCbCr[y,x-1]
        self.data[:,:,1:] = YCbCr
    elif self.mode == "420":
        YCbCr = self.data[:,:,1:]
        for y in range(1,self.height,2):
            for x in range(1,self.width,2):
                YCbCr[y,x] =YCbCr[y,x-1]
            YCbCr[y] = YCbCr[y-1]

        self.data[:,:,1:] = YCbCr
    else:
        raise NameError('downsampling mode error')
```

圖二，downsampling 實作。

Block splitting :

在這個步驟中，我們要把整張圖片切成，一個一個 block 的形式，這樣在後續也可以提供更多的好處，在上述說，YCbCr 有以下形式 4:4:4、4:2:2、4:2:0，分別對應這邊 block 的大小，分別為 8*8、16*8、16*16。也跟下面步驟的操作有相關，簡單的說，在後續的步驟中，我們要要用頻率來保存資訊，因此，取樣率越低的像素，像素間包含較多共同資訊，可以用越大的 block 來裝，來裝取更多不同的資料，反之若取樣率低，則只能用較小的 block 來裝。

然而，並不是每一個圖像都可以用夠多的 block，剛好的拼湊出來，會有些許 block 無法被圖像所塗滿，這邊我們就需要做 padding 的，通常會補黑色的 pixel。圖如三所示。

```
def block_init(channel, height, width):
    block = np.empty((height, width))
    if channel == 0:
        block.fill(0)
    else:
        block.fill(128)
    return block
```

圖三，初始化每個 block，若沒被圖片覆蓋則為黑色。

```
def split(h,w):
    num_row_block = self.width // w
    num_row_remain = self.width % w
    num_col_block = self.height // h
    num_col_remain = self.height % h
    width = w
    height = h
    block_cude = []
    for c in range(3):
        block_matrix = []
        for y in range(0, self.height, h):
            row_block = []
            for x in range(0, self.width, w):
                block = block_init(c,h,w)
                width = w
                height = h
                if y == num_col_block * h:
                    height = num_col_remain
                if x == num_row_block * w:
                    width = num_row_remain
                for j in range(height):
                    for i in range(width):
                        # print(height,width)
                        # print(y+j,x+i)
                        block[j,i] = self.data[y+j,x+i,c]
                row_block.append(np.array(block))
            block_matrix.append(np.array(row_block))
        block_cude.append(np.array(block_matrix))
    return np.array(block_cude,dtype=int)
```

圖四，Block splitting 實作。

Discrete cosine transform :

在這邊我們想要將資訊從原本的空間資訊，轉換至頻率資訊，目的是因為原本的 block 大小為 8 * 8 至 16 * 16 ，絕大多數的資訊都很像，用頻率來保存可以用較少的資訊來儲存。這邊我們可以用 DCT 來轉換，DCT 有數種不同的形態，使用 type-II 來轉換，公式如下圖五。

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad \text{for } k = 0, \ldots N-1 .$$

圖五，DCT-II 公式。

在這邊我直接使用 scipy.fftpack library 來實作 DCT 的部分，圖六為 DCT 實作，將每一個 block 轉換成頻率的型態。

```
def discrete_cosine_transform(self):
    def DCT_type2(block):
        #return dctn(block)
        return dct(dct(block.T, norm='ortho').T, norm='ortho')
    data = self.data
    data -= 128
    data_shape = data.shape
    #print(data[0,0,0])
    for c in range(data_shape[0]):
        for y in range(data_shape[1]):
            for x in range(data_shape[2]):
                data[c,y,x] = DCT_type2(data[c,y,x])
    self.data = data
```

圖六，DCT 實作。

Quantization:

在這個步驟中，我們要將從 DCT 轉過來的資訊，最進一步的處理。在頻率中，低頻的資訊，佔絕大多數，低頻的表現為背景的輪廓。高頻的資訊，佔較少，高頻的表現為圖像中的細節。而低頻的資訊主宰了整個 block，這邊我們也可以從 Q table 來看到這個關係圖七、圖八。

```
table1 = [[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]]
```

圖七，Q table 1。

```
table2 = [[17, 18, 24, 47, 99, 99, 99, 99],
          [18, 21, 26, 66, 99, 99, 99, 99],
          [24, 26, 56, 99, 99, 99, 99, 99],
          [47, 66, 99, 99, 99, 99, 99, 99],
          [99, 99, 99, 99, 99, 99, 99, 99],
          [99, 99, 99, 99, 99, 99, 99, 99],
          [99, 99, 99, 99, 99, 99, 99, 99],
          [99, 99, 99, 99, 99, 99, 99, 99]]
```

圖八，Q table 2。

在這兩個 table 中可以看到，越靠近左上角的係數較低，代表我們想要保留較多的低頻資訊，而越靠近右下角的係數較高，代表我們想要捨棄越多的高頻資訊。下圖九為實作的方式，就是將原本的 block 對 table 做相除。

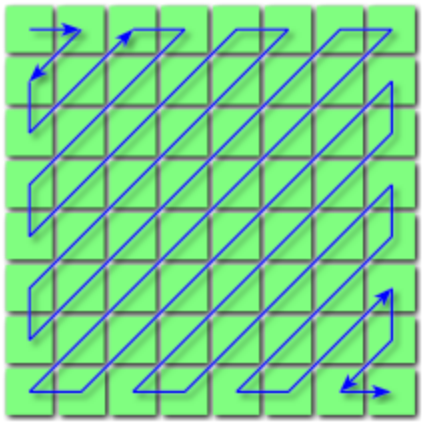
```
def quantization(self):
    def quan(block,table):
        block=np.round(block/table)
        block.astype(int)
        return block
    table1 = [[16, 11, 10, 16, 24, 40, 51, 61],
              [12, 12, 14, 19, 26, 58, 60, 55],
              [14, 13, 16, 24, 40, 57, 69, 56],
              [14, 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]]

    table2 = [[17, 18, 24, 47, 99, 99, 99, 99],
              [18, 21, 26, 66, 99, 99, 99, 99],
              [24, 26, 56, 99, 99, 99, 99, 99],
              [47, 66, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99]]
    T1 = np.empty((8,8),dtype=int)
    T2 = np.empty((8,8),dtype=int)
    for i in range(8):
        for j in range(8):
            T1[i,j]=table1[i][j]
            T2[i,j]=table2[i][j]
    data = self.data
    data_shape = data.shape
    for c in range(data_shape[0]):
        for y in range(data_shape[1]):
            for x in range(data_shape[2]):
                data[c,y,x] = quan(data[c,y,x],T1)
    self.data = data
```

圖九，quantization 實作。

entropy_coding:

在編碼的最後一個步驟，就是要對濾化後的資訊做進一步的資訊，我們以每一個 block 為一個單位來處理，這個步驟中，主要是利是資訊理論中的壓縮演算法來處理。

首先第一步就是要，將 block 的資訊給萃取出來，要有一個固定的走訪順序，在這邊是用 zigzag 來走訪，簡單來講就是對圖片走 Z 字形。如圖十所示。



圖十，zigzag ordering。

走訪之後就可以有一個固定的順序來示這個 block，以便後續的回復。有了這個陣列資訊後，因為我們知道高頻的資訊濾掉的比較多，後方會有非常多數的 0，因為我們知道 block 的大小，所以我們可以再次的減少資訊，陣列保留至最後的非 0 元素即可。

接著我們就可以對這個陣列來做編碼，例如 huffman coding。因此在最後的保存的資訊，就會是有數個 block 來表示一張圖片，每個 block 代表著圖片中不同的位置，且每個 block 中的資訊使用 huffman coding 來保存。

下方四張圖，圖十一、圖十二、圖十三、圖十四，則代表了我們上述所說的不同狀態下的 block 資訊，分別為，YCrCb 中 Y channel 中的資訊、經過 DCT 轉換過的資訊、使用 Q table 濾化後的資訊、經 zigzag 且截至最後非 0 係數的陣列。

```
[[214 216 216 213 209 206 203 201]
 [212 213 213 211 208 206 205 203]
 [212 211 210 209 208 206 206 206]
 [215 212 210 210 209 207 206 207]
 [221 215 211 213 212 207 206 207]
 [224 216 213 215 214 208 205 205]
 [224 215 212 216 216 208 203 204]
 [222 213 210 216 216 208 203 203]]
```

圖十一，YCrCb block。

```
[[660  33  -2   5   9   0   0   0]
 [ -9  -4   0  -7 -10   0   0   0]
 [  0   5  -6   0   0   0   0   0]
 [  6   6   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0]]
```

圖十二，DCT block。

```
[[41  3  0  0  0  0  0  0]
 [-1  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
```

圖十三，quantization block。

```
[41, 3, -1]
```

圖十四，huffman coding 編碼對象。

在這邊因為時間因素，我沒有實作 huffman coding，但基本上不影響理解 JPEG 壓縮演算法的概念。

```
data = self.data
data_shape = data.shape
new_data = []
self.block_shape = data_shape[:3]
for c in range(data_shape[0]):
    for y in range(data_shape[1]):
        for x in range(data_shape[2]):
            #print(data[c,y,x])
            data_list = zigzag_iterator(data[c,y,x])
            data_list, fre_dict, str_list = compute_frequency(data_list)
            new_data.append(data_list)
            #new_data[c,y,x](data_list)
self.data = new_data
```

圖十五，entropy_coding 實作。

在解碼的部分，基本上就是將上述的每一個步驟做還原，基本上將反著做就可以了，沒有甚麼特別的地方。

inverse Entropy coding:

```
data = self.data
data_shape = self.shape
block_shape = self.block_shape
ext_block_shape = (block_shape[0], block_shape[1], block_shape[2], 8, 8)
new_data = np.empty(ext_block_shape)

index = 0
for c in range(block_shape[0]):
        for y in range(block_shape[1]):
            for x in range(block_shape[2]):
                data[index] = exten_list(data[index])
                inverse_zigzag_iterator(data[index],new_data[c,y,x])
                #print(new_data[c,y,x])
                index += 1
self.data = np.array(new_data,dtype=int)
```

圖十六，de_entropy_coding 實作。

inverse Quantization:

```
def de_quan(block,table):
    block=np.round(block*table)
    block.astype(int)
    return block
data = self.data
data_shape = data.shape
for c in range(data_shape[0]):
    for y in range(data_shape[1]):
        for x in range(data_shape[2]):
            data[c,y,x] = de_quan(data[c,y,x],T1)
self.data = data
```

圖十七，de_quantization 實作。

inverse Discrete cosine transform:

```
def de_discrete_cosine_transform(self):
    def IDCT_type2(block):
        #return idctn(block)
        return idct(idct(block.T, norm='ortho').T, norm='ortho')

    data = self.data
    data_shape = data.shape
    for c in range(data_shape[0]):
        for y in range(data_shape[1]):
            for x in range(data_shape[2]):
                data[c,y,x] = IDCT_type2(data[c,y,x])
    data += 128
    #print(data)
    self.data = data
```

圖十八，de_discrete_cosine_transform 實作。

inverse Block splitting:

```
def de_block_splitting(self):
    def merge(data, shape, h, w):
        new_data = np.empty(shape)
        width = w
        height = h
        for c in range(3):
            for j in range(data.shape[1]):
                for i in range(data.shape[2]):
                    block = data[c,j,i]
                    #print(block)
                    yy = j * height
                    xx = i * width
                    for y in range(height):
                        for x in range(width):
                            if yy+y >= self.height or xx+x >= self.width:
                                break
                            new_data[yy+y, xx+x, c] = block[y, x]
        return new_data
```

圖十九，de_block_splitting 實作。

inverse Color space transformation:

```
def de_color_space_transformation(self):
    YCbCr = self.data
    rgb = []
    for y in range(self.height):
        row = []
        for x in range(self.width):
            Y, Cb, Cr = YCbCr[y,x]
            R = int(1 * Y + 1.402 * (Cr-128))
            G = int(1 * Y - 0.344136 * (Cb-128) - 0.714136 * (Cr-128))
            B = int(1 * Y + 1.772 * (Cb-128) )
            rgb.append((R,G,B))
            #row.append((R,G,B))
        #rgb.append(row)
    self.data = rgb
```

圖二十，de_color_space_transformation 實作。

圖二十一，Demo 圖 1 壓縮前。

圖二十二，Demo 圖 1 解壓縮後。

這邊可以看到，再還原的過程，就算 JPEG 是失真壓縮，肉眼基本上很難看到有什麼不同的地方，但如果放大來看細節就可以看出來一些不同的地方，例如，圖二十三及圖二十四中，我們放大來看，就可以看到很明顯的 block，

圖二十三，Demo 圖 1 壓縮前，細節。

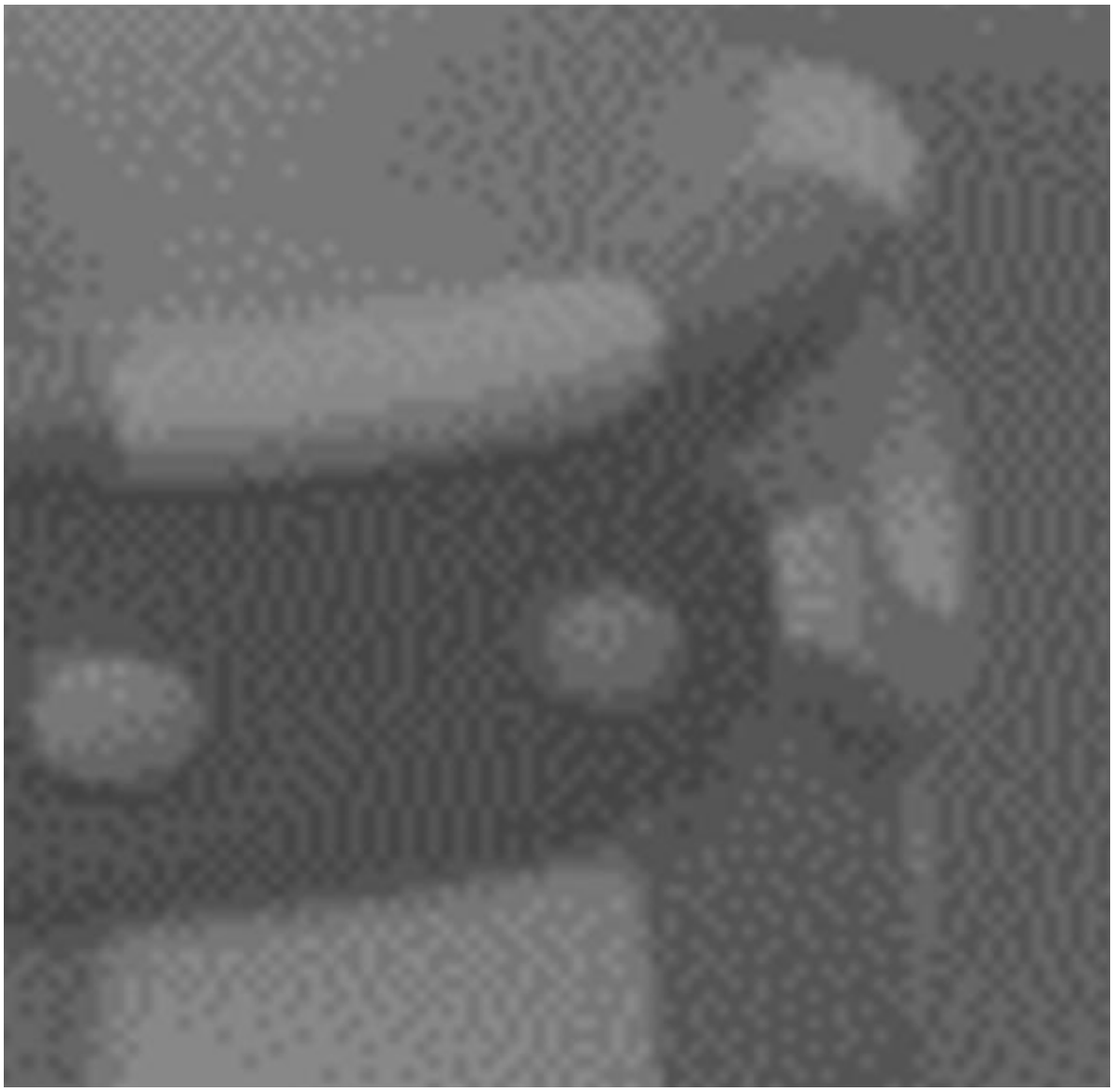
圖二十四，Demo 圖 1 解壓縮後，細節。


圖二十五，Demo 圖 2 壓縮前。

圖二十六，Demo 圖 2 壓縮前，細節。

圖二十七，Demo 圖 2 解壓縮後，細節。

圖二十八，Demo 圖 3 壓縮前。



圖二十九，Demo 圖 3 壓縮前，細節。

圖三十，Demo 圖 3 解壓縮後，細節。

圖三十一，Demo 圖 4 壓縮前。

圖三十二，Demo 圖 3 壓縮前，細節。

圖三十三，Demo 圖 3 解壓縮後，細節。

```
import numpy as np
```

```
import numba as nb
import sys
from scipy.fftpack import dct, idct
from PIL import Image

def read_image(path):
    with Image.open(path) as im:
        im = im.convert("RGB")
        width, height = im.size
        data = np.array(im.getdata()).reshape(height,width,3)
        im.show()
        return (data, height, width)


class Jpeg:
    def __init__(self,img_info, mode):
        self.data = img_info[0]
        self.height = img_info[1]
        self.width = img_info[2]
        self.shape = (self.width, self.height)
        self.block_shape = None
        self.mode = mode

    def encoding(self):
        img.color_space_transformation()
        img.downsampling()
        img.block_splitting()
        print(img.data[0,0,0])
        img.discrete_cosine_transform()
        print(img.data[0,0,0])
        img.quantization()
        print(img.data[0,0,0])
        img.entropy_coding()
        print(img.data[0])

    def deconding(self):
        img.de_entropy_coding()
        img.de_quantization()
        img.de_discrete_cosine_transform()
        img.de_block_splitting()
        img.de_color_space_transformation()

    # for encoding
    def color_space_transformation(self):
        # formula
```

```
        # Y = 0.299R + 0.587G + 0.114B
        # Cb = 128 - 0.1687R - 0.3313G + 0.5B
        # Cr = 128 + 0.5R - 0.4187G - 0.0813B
        rgb = self.data
        YCbCr = np.empty(rgb.shape)
        for y in range(self.height):
            for x in range(self.width):
                R, G, B = rgb[y,x]
                Y = 0.299 * R + 0.587 * G + 0.114 * B
                Cb = 128 - 0.168736 * R - 0.331264 * G + 0.5 * B
                Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B
                YCbCr[y,x] = np.array([Y,Cb,Cr])
        self.data = YCbCr


    def downsampling(self):
        if self.mode == "444":
            pass
        elif self.mode == "422":
            YCbCr = self.data[:,:,1:]
            #print(YCbCr)
            for y in range(self.height):
                for x in range(1,self.width,2):
                    YCbCr[y,x] =YCbCr[y,x-1]
            self.data[:,:,1:] = YCbCr
        elif self.mode == "420":
            YCbCr = self.data[:,:,1:]
            for y in range(1,self.height,2):
                for x in range(1,self.width,2):
                    YCbCr[y,x] =YCbCr[y,x-1]
                YCbCr[y] = YCbCr[y-1]

            self.data[:,:,1:] = YCbCr
        else:
            raise NameError('downsampling mode error')

    def block_splitting(self):

        def block_init(channel, height, width):
            block = np.empty((height, width))
            if channel == 0:
                block.fill(0)
            else:
                block.fill(128)
            return block
```

```
        def split(h,w):
            num_row_block = self.width // w
            num_row_remain = self.width % w
            num_col_block = self.height // h
            num_col_remain = self.height % h
            width = w
            height = h
            block_cude = []
            for c in range(3):
                block_matrix = []
                for y in range(0, self.height, h):
                    row_block = []
                    for x in range(0, self.width, w):
                        block = block_init(c,h,w)
                        width = w
                        height = h
                        if y == num_col_block * h:
                            height = num_col_remain
                        if x == num_row_block * w:
                            width = num_row_remain
                        for j in range(height):
                            for i in range(width):
                                # print(height,width)
                                # print(y+j,x+i)
                                block[j,i] = self.data[y+j,x+i,c]
                        row_block.append(np.array(block))
                    block_matrix.append(np.array(row_block))
                block_cude.append(np.array(block_matrix))
            return np.array(block_cude,dtype=int)

        if self.mode == "444":
            # 444 block 8 * 8
            self.data = split(8,8)
        elif self.mode == "422":
            # 422 block 16 * 8
            self.data = split(16,8)
        elif self.mode == "420":
            # 420 block 16 * 16
            self.data = split(16,16)
        else:
            raise NameError('downsampling mode error')

    def discrete_cosine_transform(self):
        def DCT_type2(block):
            #return dctn(block)
```

```
            return dct(dct(block.T, norm='ortho').T, norm='ortho')
        data = self.data
        data -= 128
        data_shape = data.shape
        #print(data[0,0,0])
        for c in range(data_shape[0]):
            for y in range(data_shape[1]):
                for x in range(data_shape[2]):
                    data[c,y,x] = DCT_type2(data[c,y,x])
        self.data = data

    def quantization(self):
        def quan(block,table):
            block=np.round(block/table)
            block.astype(int)
            return block
        table1 = [[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]]

        table2 = [[17, 18, 24, 47, 99, 99, 99, 99],
                  [18, 21, 26, 66, 99, 99, 99, 99],
                  [24, 26, 56, 99, 99, 99, 99, 99],
                  [47, 66, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99],
                  [99, 99, 99, 99, 99, 99, 99, 99]]
        T1 = np.empty((8,8),dtype=int)
        T2 = np.empty((8,8),dtype=int)
        for i in range(8):
            for j in range(8):
                T1[i,j]=table1[i][j]
                T2[i,j]=table2[i][j]
        data = self.data
        data_shape = data.shape
        for c in range(data_shape[0]):
            for y in range(data_shape[1]):
                for x in range(data_shape[2]):
                    data[c,y,x] = quan(data[c,y,x],T1)
```

```
        self.data = data

    def entropy_coding(self):

        def zigzag_iterator(block):
            shape = block.shape
            if shape[0] != shape[1]:
                raise NameError('zigzag block is not square')
            else:
                line = shape[0]
            data_list = []
            for i in range(line):
                x = i
                for k in range(i+1):
                    if i % 2:
                        data_list.append(block[k][x])
                    else:
                        data_list.append(block[x][k])
                    x-=1
            for i in range(1,line):
                x = line - 1
                for k in range(i,line):
                    if i % 2:
                        data_list.append(block[k][x])
                    else:
                        data_list.append(block[x][k])
                    x-=1
            return data_list

        def compute_frequency(data_list):
            EOB = len(data_list)
            for i, element in enumerate(data_list):
                if element != 0:
                    EOB = i + 1
            data_list = data_list[:EOB]
            #data_list.append("E")
            str_list = ''
            for i in data_list:
                str_list += str(i)
            fre = []
            for i in set(data_list):
                fre.append((str(i),data_list.count(i)))
            fre_dict = dict(fre)
            #print(str_list)
            return data_list, fre_dict, str_list
```

```
        data = self.data
        data_shape = data.shape
        new_data = []
        self.block_shape = data_shape[:3]
        for c in range(data_shape[0]):
            for y in range(data_shape[1]):
                for x in range(data_shape[2]):
                    #print(data[c,y,x])
                    data_list = zigzag_iterator(data[c,y,x])
                    data_list, fre_dict, str_list = compute_frequency(data_list)
                    new_data.append(data_list)
                    #new_data[c,y,x](data_list)
        self.data = new_data

    def de_entropy_coding(self):

        def exten_list(data_list):
            zeros_list = [0] * (64 - len(data_list))
            data_list = data_list + zeros_list
            return data_list

        def inverse_zigzag_iterator(data_list, block):
            shape = block.shape
            if shape[0] != shape[1]:
                raise NameError('zigzag block is not square')
            else:
                line = shape[0]

            index = 0
            for i in range(line):
                x = i
                for k in range(i+1):
                    if i % 2:
                        block[k][x] = data_list[index]
                    else:
                        block[x][k] = data_list[index]
                    index += 1
                    x-=1

            for i in range(1,line):
                x = line - 1
                for k in range(i,line):
                    if i % 2:
                        block[k][x] = data_list[index]
```

```
                else:
                    block[x][k] = data_list[index]
                index += 1
                x-=1


    data = self.data
    data_shape = self.shape
    block_shape = self.block_shape
    ext_block_shape = (block_shape[0], block_shape[1], block_shape[2], 8, 8)
    new_data = np.empty(ext_block_shape)

    index = 0
    for c in range(block_shape[0]):
            for y in range(block_shape[1]):
                for x in range(block_shape[2]):
                    data[index] = exten_list(data[index])
                    inverse_zigzag_iterator(data[index],new_data[c,y,x])
                    #print(new_data[c,y,x])
                    index += 1
    self.data = np.array(new_data,dtype=int)


def de_quantization(self):
    def de_quan(block,table):
        block=np.round(block*table)
        block.astype(int)
        return block
    table1 = [[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]]

    table2 = [[17, 18, 24, 47, 99, 99, 99, 99],
              [18, 21, 26, 66, 99, 99, 99, 99],
              [24, 26, 56, 99, 99, 99, 99, 99],
              [47, 66, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99],
              [99, 99, 99, 99, 99, 99, 99, 99]]
```

```
        T1 = np.empty((8,8),dtype=int)
        T2 = np.empty((8,8),dtype=int)
        for i in range(8):
            for j in range(8):
                T1[i,j]=table1[i][j]
                T2[i,j]=table2[i][j]

        data = self.data
        data_shape = data.shape
        for c in range(data_shape[0]):
            for y in range(data_shape[1]):
                for x in range(data_shape[2]):
                    data[c,y,x] = de_quan(data[c,y,x],T1)
        self.data = data

    def de_discrete_cosine_transform(self):
        def IDCT_type2(block):
            #return idctn(block)
            return idct(idct(block.T, norm='ortho').T, norm='ortho')

        data = self.data
        data_shape = data.shape
        for c in range(data_shape[0]):
            for y in range(data_shape[1]):
                for x in range(data_shape[2]):
                    data[c,y,x] = IDCT_type2(data[c,y,x])
        data += 128
        #print(data)
        self.data = data


    def de_block_splitting(self):
        def merge(data, shape, h, w):
            new_data = np.empty(shape)
            width = w
            height = h
            for c in range(3):
                for j in range(data.shape[1]):
                    for i in range(data.shape[2]):
                        block = data[c,j,i]
                        #print(block)
                        yy = j * height
                        xx = i * width
                        for y in range(height):
                            for x in range(width):
```

```
                                if yy+y >= self.height or xx+x >= self.width:
                                    break
                                new_data[yy+y, xx+x, c] = block[y,
x]
            return new_data

        data = self.data
        #print(data.shape)
        shape = self.shape
        new_shape = (shape[1], shape[0], 3)

        if self.mode == "444":
            # 444 block 8 * 8
            self.data = merge(data, new_shape, 8, 8)
        elif self.mode == "422":
            # 422 block 16 * 8
            self.data = merge(data, new_shape, 16, 8)
        elif self.mode == "420":
            # 420 block 16 * 16
            self.data = merge(data, new_shape, 16, 16)
        else:
            raise NameError('downsampling mode error')


    def de_downsampling(self):
        pass

    def de_color_space_transformation(self):
        YCbCr = self.data
        rgb = []
        for y in range(self.height):
            row = []
            for x in range(self.width):
                Y, Cb, Cr = YCbCr[y,x]
                R = int(1 * Y + 1.402 * (Cr-128))
                G = int(1 * Y - 0.344136 * (Cb-128) - 0.714136 * (Cr-128))
                B = int(1 * Y + 1.772 * (Cb-128) )
                rgb.append((R,G,B))
                #row.append((R,G,B))
            #rgb.append(row)
        self.data = rgb


def show_image(img):
    new_im = Image.new("RGB",img.shape)
```

```
    new_im.putdata(img.data)
    new_im.show()

np.set_printoptions(threshold=sys.maxsize)
img_info = read_image("./image/EdgeDetectors_Original.bmp")
img = Jpeg(img_info, "444")

img.encoding()

img.deconding()

show_image(img)
```