**实验4 数字图像编码实验**

姓名：张日恒 学号：201050076

本次作业需要提交以下内容：

|  |  |
| --- | --- |
| 提交内容 | 详细要求 |
| 作业文档 | 对算法原理进行简单说明；  展示实验效果（每个实验的测试图片不得少于5张，实验用图像自行选择。）；  对实验结果进行分析。 |
| 程序源代码 | 相关程序的全部源代码，要求能够正常编译和运行。 |

问题3: 实现一维无损预测压缩,计算原图和压缩以后的尺寸，计算压缩率并比较分析.

**作业二: 有损压缩算法实验**

查阅JPEG编码的有关资料，对图像进行JPEG压缩，算法步骤必须包括如下几个部分：图像分块，离散余弦变换，量化，ac和dc系数的Z字形编排。

问题1: 质量因子分别选为20，60，80，对比显示原图与不同质量因子下解码后的图像；

问题2: 记录图像大小、压缩比、均方根误差；对结果进行分析。

结果图的分布说明如下：

左上角第一幅图为原图，右上角第二幅图为质量因子为20下解码后的效果图，左下角第三幅图为质量因子为60下解码后的效果图，右下角第四幅图为质量因子为80下解码后的。下面是五张用于测试的图像的结果图：

|  |  |
| --- | --- |
|  |  |
|  |  |

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

|  |  |
| --- | --- |
|  |  |
|  |  |

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

|  |  |
| --- | --- |
|  |  |
|  |  |

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

|  |  |
| --- | --- |
|  |  |
|  |  |

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

|  |  |
| --- | --- |
|  |  |
|  |  |

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

import cv2  
import numpy as np  
from itertools import groupby  
import sys  
class RLE:  
 def \_\_init\_\_(self):  
 self.path = ''  
  
  
 def matrix2list(self, matirx):  
 *""" 按照行程编码样式将2维数组展开为一维数组 """* mrows, mcols = matirx.shape[:2]  
 mrows -= 1  
 mcols -= 1  
 mlen = min(mrows, mcols)  
  
 rmatrix = []  
 rmatrix.append(matirx[0][0])  
  
 rmatrix.extend(self.first\_encode(matirx, mlen))  
 if mcols > mrows:  
 rmatrix.extend(  
 self.colmore\_middle\_encode(matirx, mlen, mcols, mrows))  
 rmatrix.extend(self.colmore\_last\_encode(matirx, mlen, mcols,  
 mrows))  
  
 else:  
 rmatrix.extend(  
 self.rowmore\_middle\_encode(matirx, mlen, mcols, mrows))  
 rmatrix.extend(self.rowmore\_last\_encode(matirx, mlen, mcols,  
 mrows))  
  
 rmatrix.append(matirx[-1][-1])  
  
 return rmatrix  
  
 def first\_encode(self, matirx, mlen):  
 rmatrix = []  
 for len in range(1, mlen + 1):  
 if (len % 2 == 1):  
 for i in range(0, len + 1):  
 rmatrix.append(matirx[i][len - i])  
 else:  
 for i in range(0, len + 1):  
 rmatrix.append(matirx[len - i][i])  
  
 return rmatrix  
  
 def colmore\_middle\_encode(self, matirx, mlen, mcols, mrows):  
 rmatrix = []  
 if mlen % 2 == 0:  
 for extra in range(mcols - mrows):  
 if extra % 2 == 0:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[i][mlen - i + extra + 1])  
 else:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[mlen - i][i + extra + 1])  
 else:  
 for extra in range(mcols - mrows):  
 if extra % 2 == 1:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[i][mlen - i + extra + 1])  
 else:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[mlen - i][i + extra + 1])  
  
 return rmatrix  
  
 def colmore\_last\_encode(self, matirx, mlen, mcols, mrows):  
 rmatrix = []  
 if mcols % 2 == 0:  
 for len in range(0, mlen - 1):  
 if len % 2 == 0:  
 for i in range(mlen - len):  
 rmatrix.append(  
 matirx[mlen - (mlen - 1 - len - i)][mlen - i +  
 mcols - mrows])  
  
 else:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen -  
 i][mlen - (mlen - 1 - len - i) +  
 mcols - mrows])  
 else:  
 for len in range(0, mlen - 1):  
 if len % 2 == 1:  
 for i in range(mlen - len):  
 rmatrix.append(  
 matirx[mlen - (mlen - 1 - len - i)][mlen - i +  
 mcols - mrows])  
 else:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen -  
 i][mlen - (mlen - 1 - len - i) +  
  
 mcols - mrows])  
 return rmatrix  
  
 def rowmore\_middle\_encode(self, matirx, mlen, mcols, mrows):  
 rmatrix = []  
 if mlen % 2 == 0:  
 for extra in range(mrows - mcols):  
 if extra % 2 == 1:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[mlen - i + extra + 1][i])  
 else:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[i + extra + 1][mlen - i])  
 else:  
 for extra in range(mrows - mcols):  
 if extra % 2 == 0:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[mlen - i + extra + 1][i])  
 else:  
 for i in range(mlen + 1):  
 rmatrix.append(matirx[i + extra + 1][mlen - i])  
  
 return rmatrix  
  
  
 def rowmore\_last\_encode(self, matirx, mlen, mcols, mrows):  
 rmatrix = []  
 if mrows % 2 == 0:  
 for len in range(0, mlen - 1):  
 if len % 2 == 0:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +  
 mrows - mcols][mlen - i])  
 else:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen - i + mrows -  
 mcols][mlen -  
 (mlen - 1 - len - i)])  
 else:  
 for len in range(0, mlen - 1):  
 if len % 2 == 1:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +  
 mrows - mcols][mlen - i])  
 else:  
 for i in range(mlen - len):  
 rmatrix.append(matirx[mlen - i + mrows -  
 mcols][mlen -  
 (mlen - 1 - len - i)])  
 return rmatrix  
  
 def encode(self, lst):  
 lst\_encode = np.array([(len(list(group)), name)  
 for name, group in groupby(lst)])  
 return lst\_encode.flatten()  
  
  
 def decode(self, lst\_encode):  
 lst = []  
 for i in range(0, len(lst\_encode), 2):  
 print(lst\_encode[i])  
 length = int(lst\_encode[i])  
 for j in range(length):  
 lst.append(lst\_encode[i + 1])  
 return lst  
  
 def compressimg(self,img):  
 r\_img = self.encode(self.matrix2list(img)).astype(np.uint8)  
 return r\_img  
  
 def compress(self):  
  
  
 for i in range(5):  
 i = i + 1  
 imgpath = 'img' + str(i) + '.jpg'  
 image = cv2.imread(imgpath, 1)  
 image = cv2.resize(image, (200, 200))  
 size = sys.getsizeof((image.flatten()))  
 print("Image {}:".format(i))  
 print("Origin Image's Size is {:.2f} KB.".format(size / 1024))  
 [b, g, r] = cv2.split(image)  
  
 r\_b = self.encode(self.matrix2list(b)).astype(np.uint8)  
 r\_g = self.encode(self.matrix2list(g)).astype(np.uint8)  
 r\_r = self.encode(self.matrix2list(r)).astype(np.uint8)  
  
 r\_size = sys.getsizeof((r\_b)) + sys.getsizeof(  
 (r\_g)) + sys.getsizeof((r\_r))  
  
 print(  
 "After Run Length Encoding Image's Size is {:.2f} KB.\nCompressed Image's size is {:.2%} of Origin Image."  
 .format(r\_size / 1024, r\_size / size))  
  
 print()  
  
rle = RLE()  
rle.compress()

import cv2  
import numpy as np  
import matplotlib.pyplot as plt  
import random  
from skimage import exposure  
import math  
import os ,sys  
from rel import RLE  
class JPEGEncode:  
 def \_\_init\_\_(self):  
 self.path = ''  
 # self.image\_list = [x for x in listdir(path) if is\_image\_file(x)]  
 # self.image\_list = sorted(self.image\_list)  
  
 def compress(self, q\_factor):  
 outdir = './result4/'  
 for index in range(5):  
 index = index + 1  
 imgpath = 'img' + str(index) + '.jpg'  
 image = cv2.imread(imgpath, 1)  
 # image = cv2.resize(image, (200, 200))  
 outimg1 = outdir + 'img\_' + str(index) + ".png"  
 cv2.imwrite(  
 outimg1,  
 image)  
 # Step 1: convert rgb image space tp YCrCb space  
 image = cv2.cvtColor(image, cv2.COLOR\_BGR2YCrCb)  
 # 图像尺寸调整，以适应分块  
 height, width = image.shape[:2]  
 if height % 8 != 0 or width % 8 != 0:  
 image = np.pad(image, ((0, (8 - height % 8) % 8), (0, (8 - width % 8) % 8), (0, 0)),  
 "edge")  
 height, width = image.shape[:2]  
 size = sys.getsizeof((image.flatten()))  
  
 print("Image {}:".format(index))  
 print("Origin Image's Size is {:.2f} KB.".format(size / 1024))  
  
 [y, cr, cb] = cv2.split(image)  
 # Step 2: DCT decomposition, transform from time-domain to  
 # frequency-domain, and choose 8\*8 block  
 image\_dct = []  
 for img in [y, cr, cb]:  
 f\_patches = []  
 fi\_patches = []  
 # 图像分块  
 h\_patches = np.vsplit(img, height // 8)  
 for i in range(height // 8):  
 wh\_patches = np.hsplit(h\_patches[i], width // 8)  
 f\_patch = []  
 fi\_patch = []  
 for j in range(width // 8):  
 # DCT 变换  
 patch\_dct = cv2.dct(wh\_patches[j].astype(np.float))  
 f\_patch.append(patch\_dct)  
 f\_patchs = np.hstack(f\_patch)  
 f\_patches.append(f\_patchs)  
 img\_dct = np.vstack(f\_patches)  
 image\_dct.append(img\_dct)  
  
 image\_dct = np.moveaxis(image\_dct, 0, 2)  
  
 # Step 3: 量化  
 image\_dct = np.around(image\_dct / q\_factor)  
 # Step 4: 行程编码，转换为一维数组  
 rle = RLE()  
 [d\_y, d\_cr, d\_cb] = cv2.split(image\_dct)  
 image\_rle = []  
 for dct in [d\_y, d\_cr, d\_cb]:  
 dct\_rle = rle.compressimg(dct)  
 image\_rle.append(dct\_rle)  
  
 # 图像大小计算，压缩比计算  
 r\_size = sys.getsizeof((image\_rle))  
 print("quality factor:{:.2f}".format(q\_factor))  
 print("After Run JPEG Compress Image's Size is {:.2f} KB.\  
 \nCompressed Image's size is {:.4%} of Origin Image.".  
 format(r\_size / 1024, r\_size / size))  
  
 image\_iq = image\_dct \* q\_factor  
 [r\_y, r\_cr, r\_cb] = cv2.split(image\_iq)  
 image\_back = []  
 for img in [r\_y, r\_cr, r\_cb]:  
 f\_patches = []  
 # 图像分块  
 h\_patches = np.vsplit(img, height // 8)  
 for i in range(height // 8):  
 wh\_patches = np.hsplit(h\_patches[i], width // 8)  
 f\_patch = []  
 fi\_patch = []  
 for j in range(width // 8):  
 # IDCT 变换  
 patch\_dct = cv2.idct(wh\_patches[j].astype(np.float))  
 f\_patch.append(patch\_dct)  
 f\_patchs = np.hstack(f\_patch)  
 f\_patches.append(f\_patchs)  
 img\_back = np.vstack(f\_patches).astype(np.uint8)  
 image\_back.append(img\_back)  
 image\_back = np.moveaxis(image\_back, 0, 2)  
  
 # YCrCb 空间转换回 RGB 空间  
 image\_back = cv2.cvtColor(image\_back, cv2.COLOR\_YCrCb2BGR)  
 outimg = outdir + 'img\_' + str(index) + '\_' + str(q\_factor) + " IDCT.png"  
 cv2.imwrite(  
 outimg,  
 image\_back)  
  
 mse = ((image - image\_back)\*\*2).mean()  
 print("Compressed Image's MSE is {:.2f}".format(mse))  
  
  
  
j = JPEGEncode()  
j.compress(0.2)  
j.compress(0.6)  
j.compress(0.8)

import os, sys  
import numpy as np  
import cv2  
class HuffmanLetter:  
 def \_\_init\_\_(self, letter, freq):  
 self.letter = letter  
 self.freq = freq  
 self.bitstring = ""  
  
 def \_\_repr\_\_(self):  
 return f"{self.letter}"  
  
  
class HuffmanTreeNode:  
 def \_\_init\_\_(self, freq, left, right):  
 self.freq = freq  
 self.left = left  
 self.right = right  
  
  
class Huffman:  
 def \_\_init\_\_(self):  
 self.path = ""  
 # self.image\_list = [x for x in os.listdir(path) if os.is\_image\_file(x)]  
 # self.image\_list = sort\d(self.image\_list)  
  
 def byte\_cut(self, image):  
 image\_list = image.flatten()  
 chars = {}  
 for c in image\_list:  
 chars[c] = chars[c] + 1 if c in chars.keys() else 1  
 return sorted([HuffmanLetter(c, f) for c, f in chars.items()], key=lambda l: l.freq)  
  
 def build\_tree(self, letters):  
 while len(letters) > 1:  
 left = letters.pop(0)  
 right = letters.pop(0)  
 total\_freq = left.freq + right.freq  
 node = HuffmanTreeNode(total\_freq, left, right)  
 letters.append(node)  
 letters.sort(key=lambda l: l.freq)  
  
 return letters[0]  
  
 def traverse\_tree(self, root, bitstring):  
 if type(root) is HuffmanLetter:  
 root.bitstring = bitstring  
 return [root]  
 letters = []  
 letters += self.traverse\_tree(root.left, bitstring + "0")  
 letters += self.traverse\_tree(root.right, bitstring + "1")  
  
 return letters  
  
 def test(self):  
 test\_image = np.array(np.random.randint(0, 25, size=[5, 5]))  
 print(test\_image.flatten())  
 letters\_list = self.byte\_cut(test\_image)  
 print(letters\_list)  
 root = self.build\_tree(letters\_list)  
 letters = self.traverse\_tree(root, "")  
 dict = {}  
 for letter in letters:  
 dict[letter.letter] = letter.bitstring  
  
 compress = ""  
  
 for bs in test\_image.flatten():  
 compress += dict[bs]  
  
 print(sys.getsizeof(test\_image.flatten()))  
 print(sys.getsizeof(compress))  
  
 def huffman\_change(self, image):  
 letters\_list = self.byte\_cut(image)  
 root = self.build\_tree(letters\_list)  
 letters = self.traverse\_tree(root, "")  
 dict = {}  
 for letter in letters:  
 dict[letter.letter] = letter.bitstring  
  
 compress = ""  
 for bs in image.flatten():  
 compress += dict[bs]  
  
 return compress, dict  
  
 def compress(self,n):  
 outdir = './result4/'  
 for index in range(5):  
 index= index + 1  
 imgpath = 'img' + str(index) + '.jpg'  
 image = cv2.imread(imgpath, 1)  
 image = cv2.resize(image,(200,200))  
 height, width = image.shape[:2]  
 if height % n != 0 or width % n != 0:  
 image = np.pad(image, ((0, (n - height % n) % n), (0, (n - width % n) % n), (0, 0)),  
 "edge")  
 height, width = image.shape[:2]  
 size = sys.getsizeof((image.flatten()))  
  
 print("Image {}:".format(index))  
 print("Origin Image's Size is {:.2f} KB.".format(size / 1024))  
  
  
 [b, g, r] = cv2.split(image)  
 huff = []  
 dict = []  
 for img in [b, g, r]:  
 # 图像分块  
 h\_patches = np.vsplit(img, height // n)  
 for i in range(height // n):  
 wh\_patches = np.hsplit(h\_patches[i], width // n)  
 for j in range(width // n):  
 huff\_img, huff\_dict = self.huffman\_change(wh\_patches[j])  
 dict.append(huff\_dict)  
 huff.append(huff\_img)  
  
  
  
  
 r\_size = sys.getsizeof(huff)  
 r\_dict\_size = sys.getsizeof(dict)  
 # r\_size = sys.getsizeof(r)  
 # r\_dict\_size = sys.getsizeof(r\_b\_dict) + sys.getsizeof(  
 # r\_g\_dict) + sys.getsizeof(r\_r\_dict)  
 r\_size\_all = r\_size + r\_dict\_size  
  
 print("After Huffman Encoding Image's Size is {:.2f} KB.\  
 \nCompressed Image's Huffman coding size is {:.2f} KB.\  
 \nCompressed Image's Huffman coding dictonary size is {:.2f} KB.\  
 \nCompressed Image's size is {:.2%} of Origin Image.\  
 \nblock size is b {:.2f} ."  
 .format(r\_size\_all / 1024, r\_size / 1024, r\_dict\_size / 1024,  
 r\_size\_all / size, n ))  
  
 print()  
  
h = Huffman()  
h.compress(8)  
h.compress(16)  
h.compress(32)