Lab 5: JPEG Image Processing

Shane Kirkley
April 15, 2019

Table of Contents

Introduction

JPEG is a lossy compression algorithm, commonly used on digital images. It is important for the efficient storage and sharing of digital images, and has even been used for stereoscopic 3D image compression[0]. In this lab, we will implement a basic JPEG image compression scheme, and examine the tradeoffs of lossy versus lossless compression. We will describe the details and effects of quantization and variable length coding. Metrics for compression quality will be discussed, and finally an estimation of compression ratio for entropy coding will be calculated.

Background

Our implementation of JPEG compression will use the Discrete Cosine Transform (DCT) to convert blocks of pixels to frequency coefficients. Recall that the DCT expresses a finite sum of data points as the sum of cosine functions with different frequencies. We can calculate the DCT of a 2D matrix by applying the DCT to the rows then columns of the matrix, or vice versa. We will also use the concept of quantization for the compression of image data. Recall that the goal of quantization is to represent a very large or infinite set of values as a finite, and preferably small, set of values.

Part I: DCT-based encoding and Quantization

a. Encoding and Decoding

We will implement the DCT encoding of an image as follows. First divide the image into 8-by-8 blocks of pixels, then apply the forward 2-D DCT to each block. A block now contains the DCT coefficients in a zig-zag orientation, as shown in Figure 1. We note the first DCT coefficient, at the top left of the block, is the DC coefficient, while subsequent coefficients are AC with increasing frequency. These coefficients are arranged into an array following the zig-zag order. Now we have a 2D array of size 64-by-nBlocks, where nBlocks is the number of 8x8 blocks the image divides into. Each column of the array contains the DCT coefficients for a block, in order from left to right and top to bottom of the image.

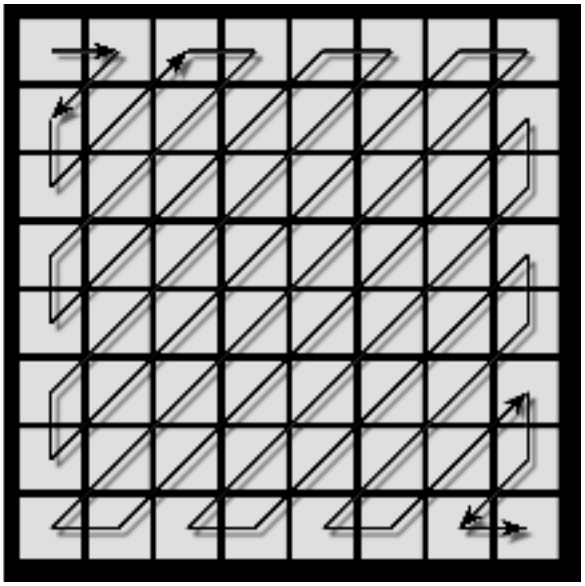


Figure 1 - Zig-zag order of DCT coefficients in block.

We can now differentially encode the DC coefficients of each block. Blocks are predicted by blocks to their left, except for the farthest left blocks of the image, which are predicted by zero. At this point the DCT encoding of an image is complete. Note that we have not implemented any compression. The image can be completely recovered by performing the inverse of the steps as described. The functions dctmgr and idctmgr, viewable in script lab5_funcs.py in the appendix implement the encoding and decoding of image data as described.

b. Quantization

To perform compression on an image, we must introduce quantization. In the JPEG standard, quantization of the AC DCT coefficients is performed as

$$F^q_{u,v} = \texttt{floor}\left(\frac{F_{u,v}}{\texttt{loss-factor} \times Q_{u,v}} + 0.5\right),$$

Where $Q_{u,v}$ is the index (u, v) in the quantization table, which can be seen in useful_arrays.py in the appendix. This quantization function allows us to define a loss factor, the effects of which we will see in the next section. We introduce quantization to the block of DCT coefficients in the encoding function, then in the decoding function we apply inverse quantization. The inverse quantization is given as

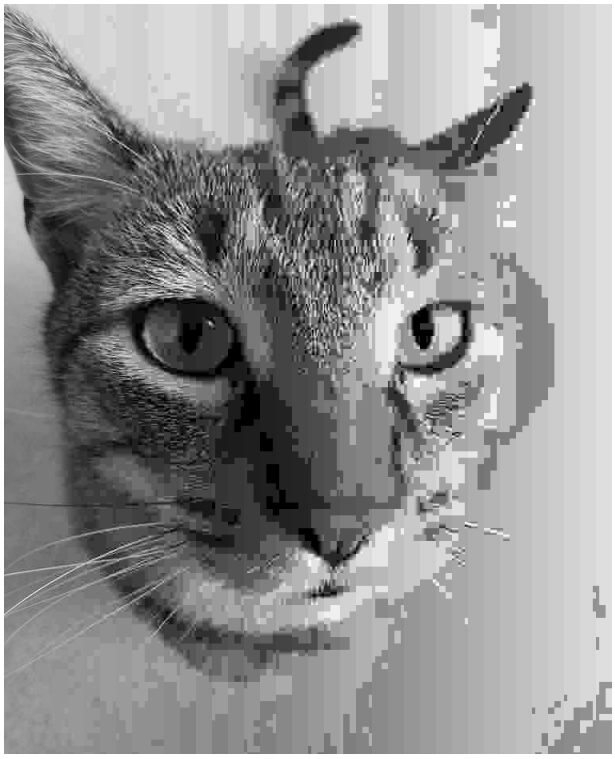
$$\tilde{F}_{u,v} = F^q_{u,v} \times \texttt{loss-factor} \times Q_{u,v}.$$

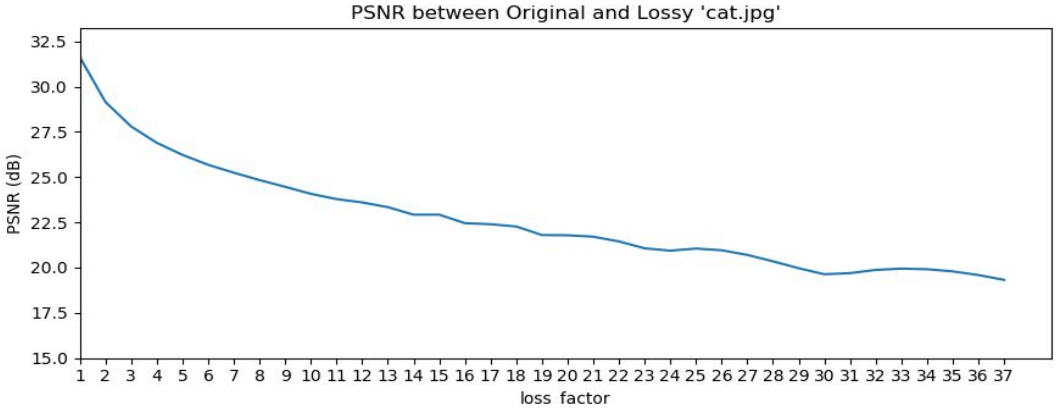
c. Peak Signal to Noise Ratio

Now that we have implemented basic lossy JPEG compression, we can examine the effects of the loss-factor in quantization. One metric that can be used to determine the loss of data in an image is the Peak Signal-to-Noise Ratio (PSNR). The PSNR between two images f and $\sim f$ is

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{255^2}{\frac{1}{N^2}\sum_{i,j=0}^{N-1}|f(i,j)-\widetilde{f}(i,j)|^2}\right).$$

The following three pages show the effects of different loss factors in the compression of different images, along with their PSNR between the original image in dB. As expected, we see a decrease in the PSNR with increased loss factor. This is again confirmed in following image of my cat, which shows a sweep of decreasing loss factor and the resulting PSNR graph. Note at high loss factors the local maxima which correspond to columns of image data with more uniform intensity, versus local minima that correspond to more varied intensity columns. Script `psnr_sweep.py` in the appendix was used to generate this image and graph. Generally, the PSNR metric gives an approximation of human perception of image reconstruction quality. A higher PSNR corresponds to a "better" image reconstruction. There are other metrics that seek to better correlate with human perception of image reconstruction quality, such as PSNR-HVS-M[1], Universal Image Quality Index (UQI)[2] and Noise Quality Measure (NQM)[3].



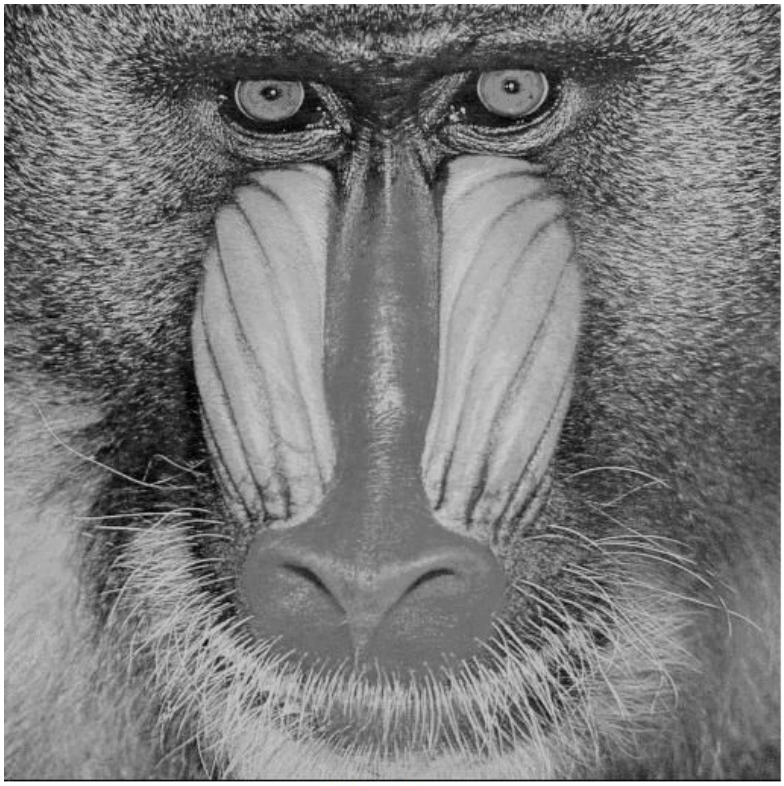


Original

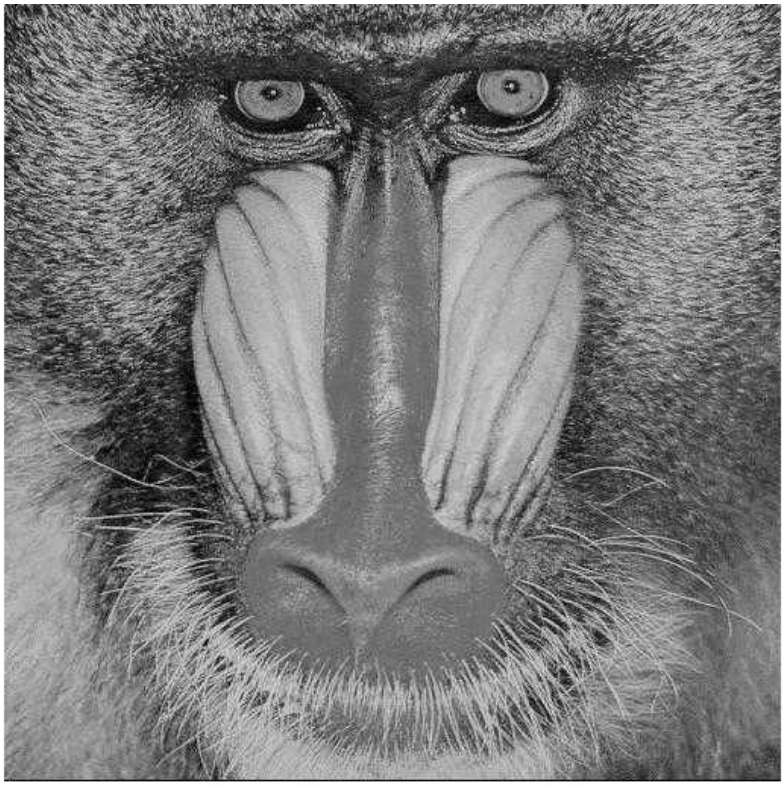
loss-factor = 1, PSNR = 31.99 dB

loss-factor = 10, PSNR = 23.29 dB

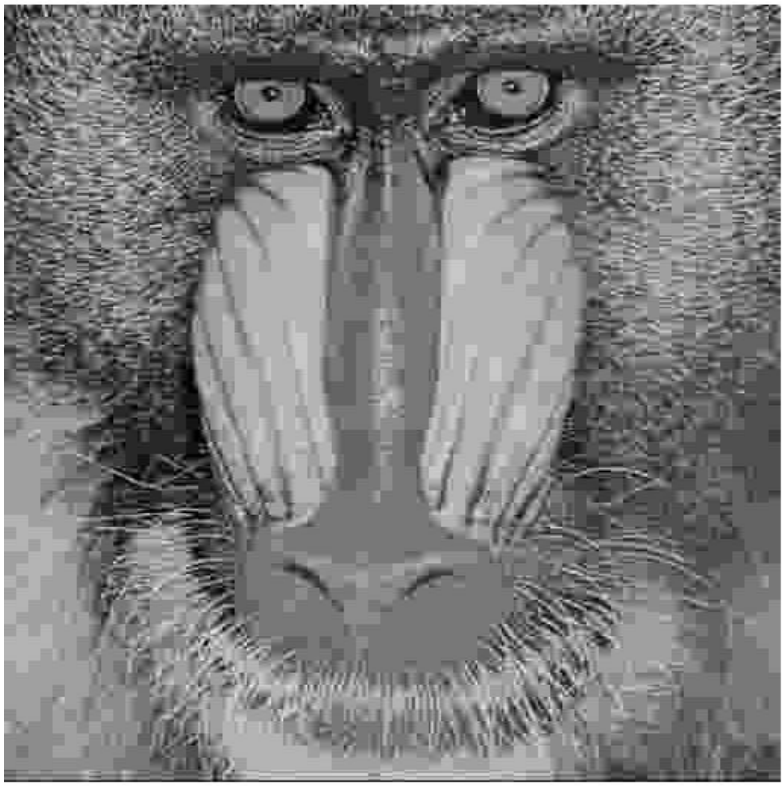
loss-factor = 20, PSNR = 21.28 dB



Original



loss-factor = 1, PSNR = 27.18 dB



loss-factor = 10, PSNR = 21.24 dB

loss-factor = 20, PSNR = 19.37 dB

Original

loss-factor = 1, PSNR = 36.82

loss-factor = 10, PSNR = 26.54

loss-factor = 20, PSNR = 22.79

Part II: Variable Length and Entropy Coding

a. Triplet Pre-processor

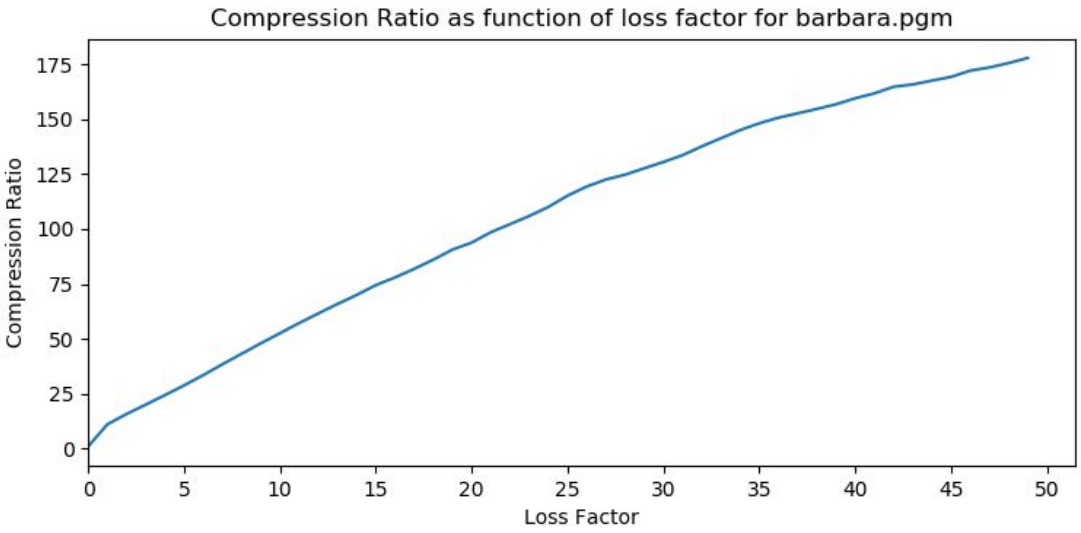
We now expect the quantized coefficients that we have for an image to contain mostly zeros, increasing with the loss factor. To further compress image data, we can introduce variable length encoding. We define a triplet `[nZeros, nBits, value]`, where `nZeros` is the number of zeros prepending a non-zero coefficient, `nBits` is the number of bits needed to represent the coefficient, and `value` is the coefficient itself. Now we can encode a block of quantized coefficients with triplets, and specify an end-of-block marker to separate blocks. The triplet encoder and decoder functions, `run_bits_value` and `irun_bits_value` can be seen in `lab5_funcs.py`. The triplet encoder is called on the quantized coefficients at the end of the image encoding scheme, and the triplet decoder is called immediately in the image decoding scheme.

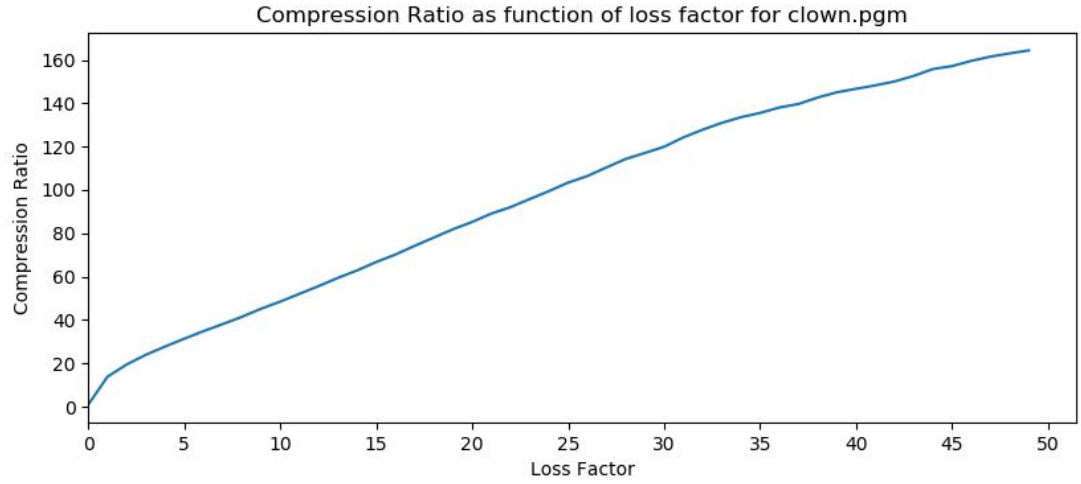
b. Entropy Coding

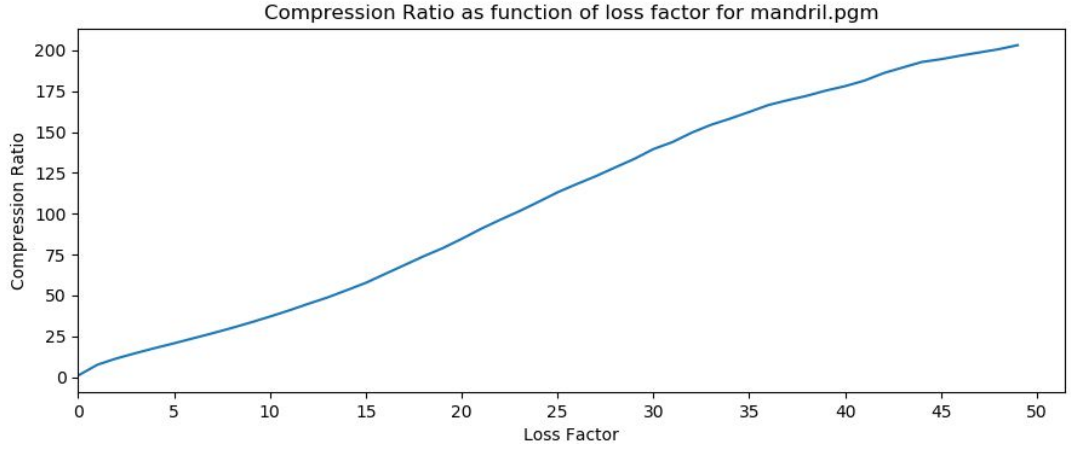
Further compression can be realized by using entropy coding. Frequently appearing symbols in data can be represented with short descriptions, while less frequent outcomes can be represented with longer descriptions. We can estimate the entropy of our encoded triplet values by calculating the entropy of each triplet value for an image. The entropy of a random variable, X, with values $\{x_1, x_2, \ldots, x_N\}$ and probabilities $\{p_1, p_2, \ldots, p_n\}$ is given by

$$H(X) = \sum_{n=1}^{N} p_n \log_2 \left(\frac{1}{p_n} \right)$$

For simplicity, we define $0*\log_2(1/0) = 0$. The sum of entropies of each triplet value gives the estimated number of bits required for a triplet. We find the estimate total number of bits required for a compressed image by multiplying the entropy sum by the number of triplets. Finally, we define the compression ratio as the number of bits needed in the original image divided by the estimated total number of bits with entropy coding. The following plots show the estimate compression ratio for the three sample images shown earlier, sweeping the loss factor from 1 to 50. These plots were generated with script `test_entropy.py`, located in the appendix. The compression ratio increases with the loss factor, and we note the rate of increase is different for different images.


Compression Ratio as function of loss factor for barbara.pgm
Compression Ratio
Loss Factor
0
25
50
75
100
125
150
175
0
5
10
15
20
25
30
35
40
45
50


Compression Ratio as function of loss factor for clown.pgm
Compression Ratio
Loss Factor
0
20
40
60
80
100
120
140
160
0
5
10
15
20
25
30
35
40
45
50


Compression Ratio as function of loss factor for mandril.pgm
Compression Ratio
Loss Factor
0
25
50
75
100
125
150
175
200
0
5
10
15
20
25
30
35
40
45
50

Conclusion

In this lab we established a basic JPEG image compression encoder and decoder. We examined how the DCT can be used transform image data, and how that data can be quantized to create lossy compression. The PSNR of image data based on loss factor was determined for several images. Variable length encoding was implemented to further compress data. Finally entropy coding was examined and the estimated compression ratio from implementing entropy coding was calculated.

References

[0] Stereoscopic JPEG implementation: https://web.archive.org/web/20111030182549/http://vrex.com/developer/sterdesc.pdf
[1] PSNR-HVS-M: http://www.ponomarenko.info/psnrhvsm.htm
[2] Universal Quality Index: https://ece.uwaterloo.ca/~z70wang/publications/uqi.html
[3] Noise Quality Measure: https://ieeexplore.ieee.org/abstract/document/841940

Appendix

lab5_funcs.py

```
#!/usr/local/bin/python3
import numpy as np
import math
import sys
from scipy.fftpack import dct
from scipy.fftpack import idct
import useful_arrays

BLOCK_SIZE = 8

"""
Forward DCT for use on 2D arrays. Applies DCT to rows then columns.
"""
def forward_dct(input):
   return dct(dct(input, norm='ortho').T, norm='ortho').T

"""
Inverse DCT for use on 2D arrays. Applies idct to columns then rows.
"""
def inverse_dct(input):
   return idct(idct(input.T, norm='ortho').T, norm='ortho')

"""
```

```
Helper fcn to return an array of shape (nblocks, rows, cols)
where nblocks * rows * cols == arr.size
"""
def blockshape(arr, rows, cols):
   h, w = arr.shape
   return arr.reshape(h//rows, rows, -1, cols).swapaxes(1,2).reshape(-1, rows, cols)

"""
Helper fcn to return 1d array of size 64 of zig-zag order elements from 8x8 2d array
"""
def zigzag(input):
   zz = useful_arrays.zz
   output = np.zeros(64)
   for i in range(64):
       idx = zz[i]
       output[i] = input[idx[0],idx[1]]
   return output

"""
Helper fcn to reverse the zigzag function (column to 2d-array)
"""
def zigzag_reverse(input):
   zz = useful_arrays.zz
   output = np.zeros((8,8))
   for i in range(64):
       idx = zz[i]
       output[idx[0], idx[1]] = input[i]
   return output

"""
Quantizer function
"""
def quantize(block, loss_factor):
   Q = useful_arrays.Q
   return np.floor(block/(Q * loss_factor) + 0.5)

"""
Inverse quantizer function
"""
def inverse_quantize(block, loss_factor):
   Q = useful_arrays.Q
   return block * Q * loss_factor

"""
Image data DCT function and encoding
Input: 2D array of grayscale image data
Output: Variable length encoded array of quantized DCT coefficients for image
"""
def dctmgr(img_data, loss_factor):
   # divide into non-overlapping 8x8 blocks
   rows = img_data.shape[0]
   cols = img_data.shape[1]
   nblocks = int(rows/BLOCK_SIZE) * int(cols/BLOCK_SIZE)
   chunked_data = blockshape(img_data, BLOCK_SIZE, BLOCK_SIZE)

   coeffs = np.zeros((64, nblocks))
```

```
    encoded = np.zeros(coeffs.shape)
    block_num = 0

    for block in chunked_data:
        # transform each block
        block = forward_dct(block)
        # quantize block
        block = quantize(block, loss_factor)
        # zig-zag data from blocks to coeffecient columns.
        coeffs[:,block_num] = zigzag(block)
        encoded[:,block_num] = coeffs[:,block_num]
        # replace DC coeffs with differentially encoded values
        if (block_num % (cols/BLOCK_SIZE)) != 0: # not at beginning of a row
            encoded[0, block_num] = coeffs[0, block_num] - coeffs[0, block_num - 1]
        else:
            encoded[0, block_num] = coeffs[0, block_num]
        block_num = block_num + 1

    # variable length encoding
    encoded = run_bits_value(encoded)
    return encoded

"""
Image data Inverse DCT and decoding
Input: - coeffs: variable length encoded array of quantized DCT coefficients
       - Image width and height
       - loss_factor in quantization
Output: 2D array of grayscale image data.
"""
def idctmgr(coeffs, img_width, img_height, loss_factor):
    rows = img_height # for my sanity
    cols = img_width
    img_data = np.zeros((rows, cols))

    # variable length decoding
    coeffs = irun_bits_value(coeffs)
    nblocks = coeffs.shape[1]

    for block_num in range(nblocks):
    # for each block b, column coeffs[:,b] is used to reconstruct the block.
        # undo prediction with DC coeffs
        if (block_num % (cols/BLOCK_SIZE)) != 0:
           coeffs[0, block_num] += coeffs[0, block_num - 1]
        # reverse zig-zag into a block.
        block = zigzag_reverse(coeffs[:, block_num])
        # inverse quantization of block.
        block = inverse_quantize(block, loss_factor)
        # inverse 2d dct function on block.
        block = inverse_dct(block)
        # put block into img data.
        # (img_row, img_col) is coords of top left corner of block
        img_row = int((block_num * BLOCK_SIZE) / cols) * BLOCK_SIZE
        img_col = int((block_num * BLOCK_SIZE) % cols)
        img_data[img_row:img_row+8, img_col:img_col+8] = block

    return img_data
```

```
"""
Triplet pre-processor and inverse processor functions
"""
def run_bits_value(coeffs):
   # input: DCT coefficient array
   # output: 2d matrix symb, size nx3, each row storing (nZeros, nBits, value)
   DC_BITS = 12
   AC_BITS = 11
   symb = []

   for block in coeffs.T:
       nZeros = 0
       symb.append([0, DC_BITS, np.float(block[0])])
       for coeff in block[1:]:
           if coeff == 0:
               nZeros += 1
           else:
               # add new entry to symb
               symb.append([nZeros, AC_BITS, np.float(coeff)])
               nZeros = 0
       # EOB
       symb.append([0,0,0])

   # all blocks processed, convert to np array
   return np.asarray(symb)

def irun_bits_value(symb):
   # input: 2d matrix symb
   # output: DCT coefficient array
   coeffs = np.empty((0,64))
   block = np.empty(0)
   for row in symb:
       if (row[0] == 0) and (row[1] == 0) and (row[2] == 0):
           # add trailing zeros
           num_zeros = 64 - block.shape[0]
           zeros = np.zeros(num_zeros)
           block = np.append(block,zeros)
           coeffs = np.vstack((coeffs, block))
           block = np.empty(0)
           continue
       nZeros, nBits, value = row
       if(nZeros > 0):
           zero = np.zeros(int(nZeros))
           block = np.append(block, zero)
       block = np.append(block,value)
   return coeffs.T

"""
psnr gives peak signal to noise ratio in dB between two images.
"""
def psnr(im_1, im_2):
   if im_1.shape != im_2.shape:
       print(f"ERROR in psnr: images are not same size")
       sys.exit(-1)
   denom = np.sum( (im_1 - im_2)**2 ) / (im_1.shape[0]*im_1.shape[1])
```

```
    if denom == 0:
        return 0
    else:
        PSNR = 10 * np.log10( 255**2 / denom )
        return PSNR
```

compare_psnr.py

```
#!/usr/local/bin/python3
import numpy as np
import math
import sys
from PIL import Image
import lab5_funcs as lab5



# open image, prepare name
filename = sys.argv[1]
img_data = np.array(Image.open(filename).convert('L'))
filename = filename[:-4]
img_height, img_width = img_data.shape
# save as jpg for comparison
Image.fromarray(img_data).convert('L').save(filename + ".jpg")
img_data = np.array(Image.open(filename + ".jpg").convert('L'))

# generate images with loss-factors of 1, 10, and 20
coeffs = lab5.dctmgr(img_data, 1)
img_loss1 = lab5.idctmgr(coeffs, img_width, img_height, 1)
Image.fromarray(img_loss1).convert('L').save(filename + "_loss1.jpg")

coeffs = lab5.dctmgr(img_data, 10)
img_loss10 = lab5.idctmgr(coeffs, img_width, img_height, 10)
Image.fromarray(img_loss10).convert('L').save(filename + "_loss10.jpg")

coeffs = lab5.dctmgr(img_data, 20)
img_loss20 = lab5.idctmgr(coeffs, img_width, img_height, 20)
Image.fromarray(img_loss20).convert('L').save(filename + "_loss20.jpg")

# get psnr between original images and lossy images
psnr1 = lab5.psnr(img_data, img_loss1)
psnr10 = lab5.psnr(img_data, img_loss10)
psnr20 = lab5.psnr(img_data, img_loss20)

print("PSNR with loss_factor = 1: " + str(psnr1) + "dB")
print("PSNR with loss_factor = 10: " + str(psnr10) + "dB")
print("PSNR with loss_factor = 20: " + str(psnr20) + "dB")
```

psnr_sweep.py

```
#!/usr/local/bin/python3
import numpy as np
import math
import sys
from PIL import Image
import lab5_funcs as lab5
import matplotlib.pyplot as plt

SWEEP_LENGTH = 38
GEN_SWEEP_IMG = True

filename = sys.argv[1]
img_data = np.array(Image.open(filename).convert('L'))
img_height, img_width = img_data.shape
filename = filename[:-4]
img_height, img_width = img_data.shape
# save as jpg for comparison
Image.fromarray(img_data).convert('L').save(filename + ".jpg")
img_data = np.array(Image.open(filename + ".jpg").convert('L'))
psnr = np.zeros(SWEEP_LENGTH)

if(GEN_SWEEP_IMG):
   step = int(img_width/SWEEP_LENGTH)
   img_out = np.zeros(img_data.shape)
   img_out[:,:1*step] = img_data[:,:1*step]

for loss_factor in range(1,SWEEP_LENGTH):
   coeffs = lab5.dctmgr(img_data, loss_factor)
   img_loss = lab5.idctmgr(coeffs, img_width, img_height, loss_factor)
   psnr[loss_factor] = lab5.psnr(img_data, img_loss)
   if(GEN_SWEEP_IMG):
       step_idx = loss_factor*step
       img_out[:,step_idx:step_idx+step] = img_loss[:,step_idx:step_idx+step]

if(GEN_SWEEP_IMG):
   Image.fromarray(img_out).convert('L').save("sweep.jpg")

plt.plot(psnr)
plt.xlim(left=1)
plt.xlabel("loss_factor")
plt.ylabel("PSNR (dB)")
plt.ylim(bottom = 15)
plt.xticks(np.arange(1,SWEEP_LENGTH))
plt.title("PSNR between Original and Lossy '%s" % filename + ".jpg'")
plt.show()
```

test_entropy.py

```
#!/usr/local/bin/python3
import numpy as np
import math
import sys
from scipy.fftpack import dct
from scipy.fftpack import idct
from PIL import Image
import matplotlib.pyplot as plt
import lab5_funcs as lab5

def entropy(p):
   sum = 0
   for i in range(p.shape[0]):
       if p[i] > 0:
           sum += p[i] * np.log2(1/p[i])
   return sum

def nZero_entropy(nZero):
   n_zeros = np.zeros(64)
   for n in nZero:
       n_zeros[int(n)] += 1
   p_zeros = n_zeros / nZero.shape[0]
   return entropy(p_zeros)

def nBit_entropy(nBits):
   n_bits = np.zeros(15)
   for n in nBits:
       n_bits[int(n)] += 1
   p_bits = n_bits / nBits.shape[0]
   return entropy(p_bits)

def value_entropy(value):
   n_values = np.zeros(200)
   for n in value:
       if(n > 0):
           n_values[int(n)] += 1
   p_values = n_values / value.shape[0]
   return entropy(p_values)

img_data = np.array(Image.open(sys.argv[1]).convert('L'))
img_rows = img_data.shape[0]
img_cols = img_data.shape[1]

compression_ratio = np.zeros(50)
compression_ratio[0] = 1

for loss_factor in range(1,50):
   symb = lab5.dctmgr(img_data, loss_factor)
   nZero_e = nZero_entropy(symb[:,0])
```

```
    nBit_e = nBit_entropy(symb[:,1])
    value_e = value_entropy(symb[:,2])

    bit_estimate = nZero_e + nBit_e + value_e
    total_bit_estimate = bit_estimate * symb.shape[0]

    cr = (8 * img_rows * img_cols) / total_bit_estimate
    compression_ratio[loss_factor] = cr

plt.plot(compression_ratio)
plt.xlim(left=1)
plt.xlabel("Loss Factor")
plt.xticks(np.arange(0,51,5))
plt.ylabel("Compression Ratio")
plt.title("Compression Ratio as function of loss factor for %s" %
sys.argv[1])
plt.show()
```