ECE 6258 Digital Image Processing

Problem Set 5

Harsh Bhate
Mail: bhate@gatech.edu

Problem 2.

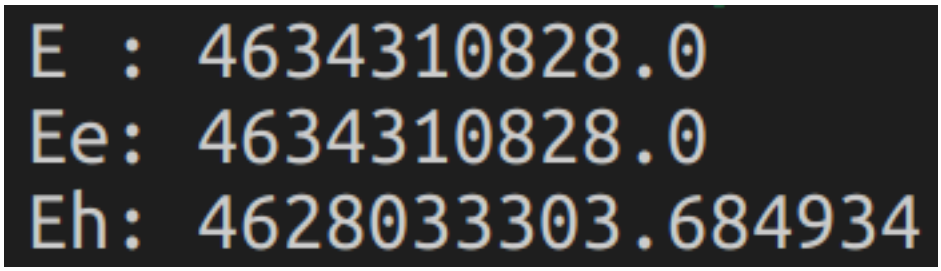
Code

```
import dippykit as dip
import numpy as np
from matplotlib import pyplot as plt
import sys

np.set_printoptions(threshold=np.nan)

#Part 2(a)
def read_image(img_path):
    '''Function to read image, convert to float and normalize'''
    X = dip.image_io.im_read(img_path)
    X = dip.im_to_float(X)
    X *= 255
    return X

#Part 2(b)
def floater(X):
    return dip.float_to_im(X/255)

def show_image(IMG, IMG_DCT, IMG_DIFF, IMG_RECONS):
    '''Function to display image'''
    IMG = floater(IMG)
    IMG_DCT = floater(IMG_DCT)
    IMG_DIFF = floater(IMG_DIFF)
    IMG_RECONS = floater(IMG_RECONS)

    dip.figure()
    dip.subplot(2, 2, 1)
    dip.imshow(IMG, 'gray')
    dip.title('Grayscale Image', fontsize='x-small')

    dip.subplot(2, 2, 2)
    dip.imshow(IMG_DCT, 'gray')
    dip.title('DCT of Image', fontsize='x-small')

    dip.subplot(2, 2, 3)
```

```
37      dip.imshow(IMG_DIFF)
38      dip.colorbar()
39      dip.title('Error image', fontsize='x-small')
40
41      dip.subplot(2, 2, 4)
42      dip.imshow(IMG_RECONS, 'gray')
43      dip.title('Reconstructed Image', fontsize='x-small')
44
45      dip.show()
46
47  #Part 2(c)
48  def energy(X):
49      '''Compute the energy of image'''
50      return np.power(np.linalg.norm(X, 'fro'),2)
51
52  #Part 2(d)
53  def dct(X):
54      '''Compute 2-D DCT of 8x8 non-overlaping blocks'''
55      block_size = (8,8)
56      return dip.block_process(X, dip.dct_2d , block_size)
57
58  #Part 2(e)
59  def inverse_dct(X):
60      '''Compute the inverse 2-D DCT of 8x8 non-overlapping blocks'''
61      block_size = (8,8)
62      return dip.block_process(X, dip.idct_2d , block_size)
63
64  #Part 2(g)
65  def error_image(X,Y):
66      '''Return the difference between the original image and
        difference image'''
67      return np.abs(X-Y)
68
69  #Part 2(h)
70  def masking(coeff):
71      '''Return the inverse DCT with the first 15 coefficients only
        '''
72      block_size = (8,8)
73      mask = np.zeros(block_size)
74      print ("Coeff: "+str(coeff)+"\n")
75      idx = dip.utilities.zigzag_indices(mask.shape, coeff)
76      mask[idx] = 1
77      return mask
78
79  def save(X):
80      '''Save the blocks'''
81      np.save('sub_blocks/saved.npy',X)
82      sys.exit()
83
84  def inverse_dct_reduced_coeff(X, coeff):
85      '''Return the inverse DCT with reduced Coeff'''
86      mask = masking(coeff)
87      print (mask)
88      block_size = (8,8)
89      masker = lambda X: np.multiply(X,mask)
90      dct_recons = dip.block_process(X, masker , block_size)
91      return dip.block_process(dct_recons, dip.idct_2d , block_size)
```

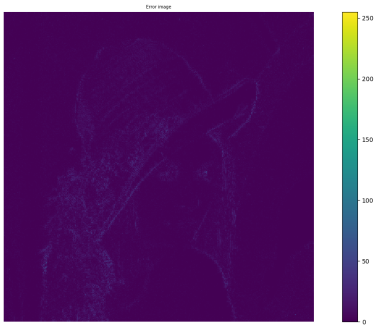
```
92
93 #Part 2(1)
94 def energy_plot(X):
95     '''Plot the engery'''
96     coeffs = np.linspace(1,64,64, dtype=np.int)
97     energies = []
98     for i in coeffs:
99         energies.append(energy(inverse_dct_reduced_coeff(X,i)))
100    plt.plot(coeffs, energies)
101    plt.autoscale(enable=True)
102    plt.title('Energy vs Nos of DCT Coefficients')
103    plt.show()
104
105 if __name__=="__main__":
106    '''Main Function'''
107    image_path = "/home/harshbhate/Codes/DIP/images/lena_gray.png"
108    IMG = read_image(image_path)
109    E = energy(IMG)
110    print ("E : "+str(E))
111    IMG_dct = dct(IMG)
112    IMG_hat_1 = inverse_dct(IMG_dct)
113    E_e = energy(IMG_hat_1)
114    print ("Ee: "+str(E_e))
115    # show_image(IMG_diff)
116    IMG_hat_2 = inverse_dct_reduced_coeff(IMG_dct,6)
117    E_h = energy(IMG_hat_2)
118    IMG_diff = error_image(IMG, IMG_hat_2)
119    print ("Eh: "+str(E_h))
120    #show_image(IMG, IMG_dct, IMG_diff ,IMG_hat_2)
121    energy_plot(IMG_dct)
```

Energies

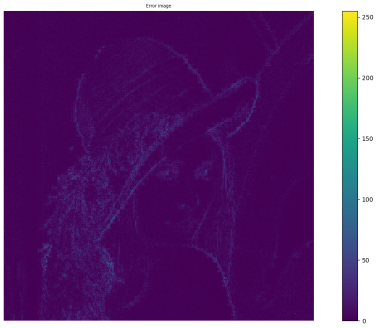


Results

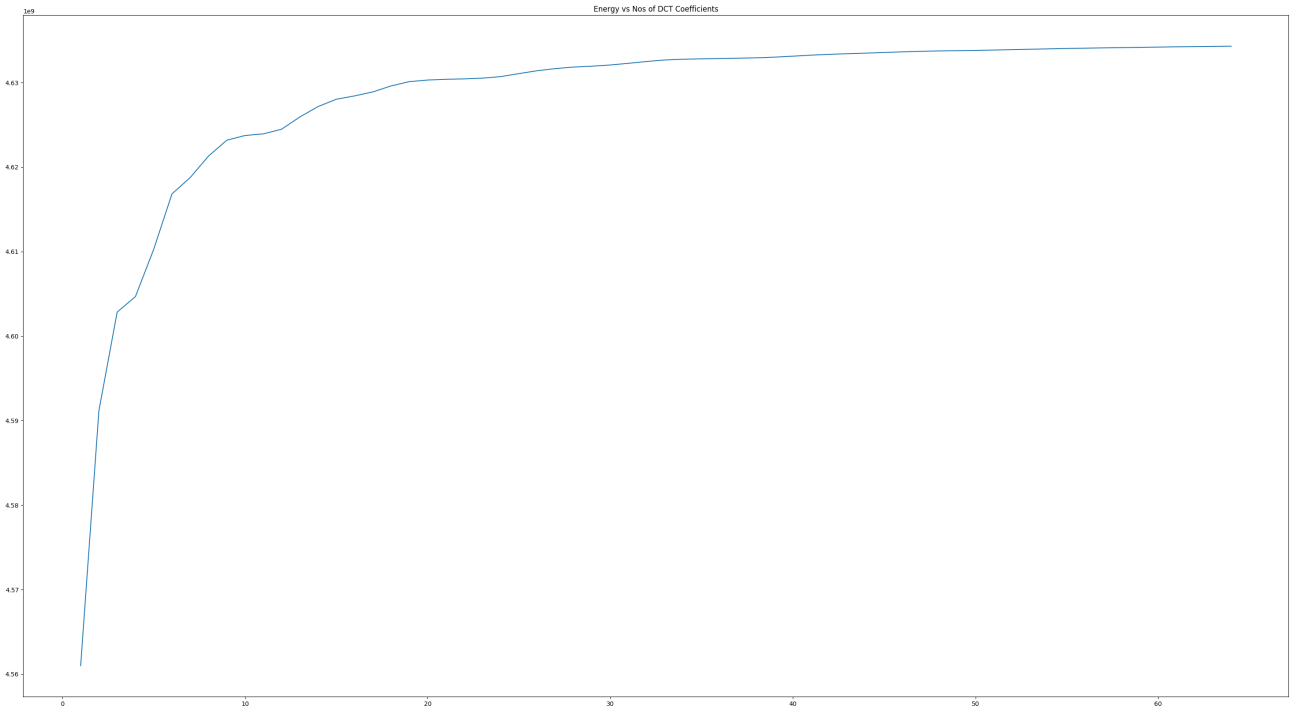
15 Coefficients



6 Coefficients



Energy Plot


Energy vs Nos of DCT Coefficients
1e9
4.63
4.62
4.61
4.60
4.59
4.58
4.57
4.56
0
10
20
30
40
50
60

Problem 4.

(a).

$$\hat{H}_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix}$$

(b). The Haar transform of X is given by:

$$\begin{aligned} Y_{haar} &= \frac{1}{16} H_2 X {H_2}^T \\ &= \frac{1}{16} \begin{bmatrix} 144 & 0 & -8\sqrt{2} & -8\sqrt{2} \\ 0 & 16 & -8\sqrt{2} & -8\sqrt{2} \\ -8\sqrt{2} & -8\sqrt{2} & 16 & 0 \\ -8\sqrt{2} & -8\sqrt{2} & 0 & 16 \end{bmatrix} \end{aligned} \tag{1}$$

(c). The inverse Haar Transform is given by:

$$\begin{aligned} \tilde{X} &= {H_2}^T Y_{haar} H_2 \\ &= \begin{bmatrix} 8 & 8 & 8 & 8 \\ 8 & 16 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 16 \end{bmatrix} \end{aligned} \tag{2}$$

(d). The Hadarmard Transform matrix is given by:

$$\hat{H}_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The Haramard Transform is:

$$\begin{aligned} Y_{hadamard} &= \frac{1}{16} H_2 X {H_2}^T \\ &= \begin{bmatrix} 9 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \end{aligned} \tag{3}$$

Inverse Hadarmard Transform is given by:

$$\begin{aligned}\tilde{X} &= {H_2}^T Y_{hadamard} H_2 \\ &= \begin{bmatrix} 8 & 8 & 8 & 8 \\ 8 & 16 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 16 \end{bmatrix}\end{aligned} \tag{4}$$

(e). The Walsh-Hadarmard Transform matrix is given by:

$$\hat{H_2} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

The Walsh-Haramard Transform is:

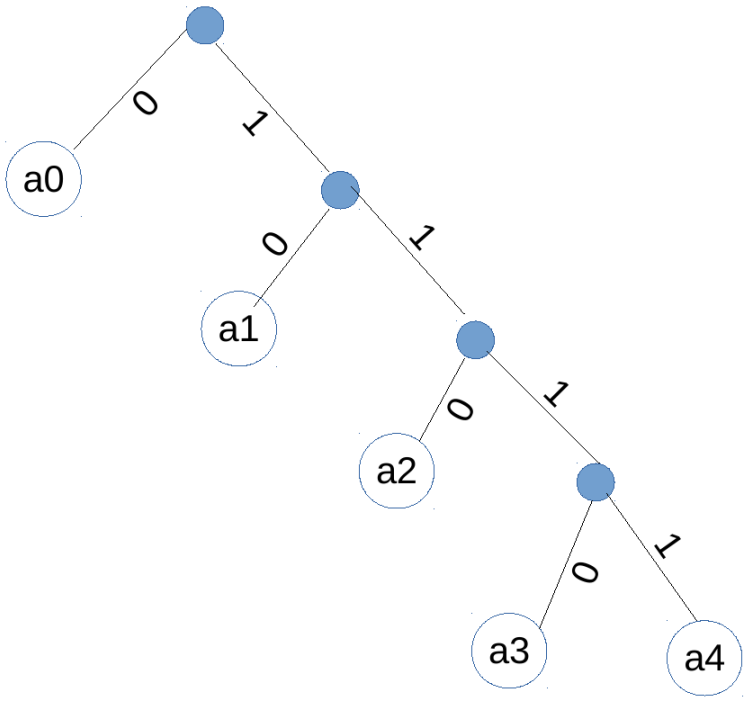
$$\begin{aligned}Y_{hadamard} &= \frac{1}{16} H_2 X {H_2}^T \\ &= \begin{bmatrix} 9 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}\end{aligned} \tag{5}$$

Inverse Walsh-Hadarmard Transform is given by:

$$\begin{aligned}\tilde{X} &= {H_2}^T Y_{hadamard} H_2 \\ &= \begin{bmatrix} 8 & 8 & 8 & 8 \\ 8 & 16 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 16 \end{bmatrix}\end{aligned} \tag{6}$$

Problem 7.

(a).



Symbol	Code Word
a_0	0
a_1	10
a_2	110
a_3	111
a_4	1111

(b).

The average code length of the encoding is given by:

$$\begin{aligned} l &= p(a_0)l_0 + p(a_1)l_1 + p(a_2)l_2 + p(a_3)l_3 + p(a_4)l_4 \\ &= (0.4)(1) + (0.2)(2) + (0.2)(3) + (0.1)(4) + (0.1)(4) \\ &= 0.4 + 0.4 + 0.6 + 0.4 + 0.4 \\ &= 2.2 \text{ bits/symbol} \end{aligned}$$

The Entropy of the encoding is given by:

$$\begin{aligned} H &= -(p(a_0)\log_2 p(a_0) + p(a_1)\log_2 p(a_1) + p(a_2)\log_2 p(a_2) + p(a_3)\log_2 p(a_3) + p(a_4)\log_2 p(a_4)) \\ &= -((0.4)(-1.32) + (0.2)(-2.32) + (0.2)(-2.32) + (0.1)(-3.32) + (0.1)(-3.32)) \\ &= 2.12 \text{ bits/symbol} \end{aligned}$$

Thus, the redundancy of the encoding is given by:

$$\begin{aligned} R &= l - H \\ &= 2.2 - 2.12 \\ &= 0.08 \text{ bits/symbol} \end{aligned}$$

(c).

The given sequence is:

$$a_2a_1a_3a_2a_1a_2 = 11010111011010110$$

(d).

The New Input sequence is 01010111011010110. Thus, this sequence can be decoded as: $(0)(10)(10)(1110)(110)(10)(110) = (a_0)(a_1)(a_1)(a_3)(a_2)(a_1)(a_2)$. Thus, the output of the decoding system will be $a_0a_1a_1a_3a_2a_1a_2$.

Problem 8.

(a).

Pixel	Frequency	Probability
21	12	0.375
243	12	0.375
95	4	0.125
169	4	0.125

The entropy of the system is given by:

$$H = -\sum p_i log_2 p_i$$

From the table above, $H = 1.81$ bits/symbol

(b).

The Huffman Encoding of the Pixels is as follows:

Pixel	Symbol
21	0
243	11
95	101
169	100

(c).

The average word length of the encoding scheme is $l = 1.875$ bits/symbol. Without the compression, we would require 2 bits/symbol. Thus, a compression ratio of 1.067 was achieved.

(d).

Upon observation, the new clubbed symbols are as follows:

Pixels	Symbol	Frequency	Probability
21,21	a_0	4	0.25
21,95	a_1	4	0.25
169,243	a_2	4	0.25
243,243	a_3	4	0.25

The Entropy of the system is thus $2 bits/symbol$.

(e).

The new image is as follows:

21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0

The frequency distribution of the symbol is as follows:

Pixel	**Symbol**	**Frequency**	**Probability**
0	a_0	16	0.5
74	a_1	12	0.375
25	a_2	4	0.125

From the table, the entropy is 1.405 bits/symbol

(f).

The Huffman encoding algorithm's worst case performance occurs when the distribution of the symbol is equal. Thus, the algorithm is likely to yield better results as the variation in the distribution of symbols increases. This is inherently visible in the entropy of the individual pixel encoded image and the paired pixel encoded image. Both the scenarios have similar symbols but the distribution of the symbol differentiates their entropy.
Any encoding algorithm is bound to be less entropy (distrubance) prone if the number of symbols required for communicating the signal is less. Thus, the entropy of the image by pixel and difference image are different.

Problem 9.

Codes

```
import dippykit as dip
import numpy as np
import cv2
import pdb

img_path = "/home/harshbhate/Codes/DIP/images/lena.png"
# STEP 1: Loading the image
# ============================== EDIT THIS PART ==============================
im = dip.image_io.im_read(img_path)

# STEP 2: Converting to YCrCb
# ============================== EDIT THIS PART ==============================
im = dip.utilities.rgb2ycbcr(im) # HINT: Look into dip.rgb2ycbcr

# STEP 3: Keep only the luminance channel
# ============================== EDIT THIS PART ==============================
im = im[:,:,0]

# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
dip.figure()
dip.subplot(2, 2, 1)
dip.imshow(im, 'gray')
dip.title('Grayscale image', fontsize='x-small')

# STEP 4: Calculating the image entropy
# ============================== EDIT THIS PART ==============================
H = dip.metrics.entropy(im)

# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
print("Entropy of the grayscale image = {:.2f} bits/pixel".format(H))

# STEP 5: Coding of the original image
# ============================== EDIT THIS PART ==============================
byte_seq, _,_,_ = dip.coding.huffman_encode(im.flatten())
l,b = im.shape
im_bit_rate = (len(byte_seq)*8.0)/float(l*b)
# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
print("Bit rate of the original image = {:.2f} bits/pixel"
      .format(im_bit_rate))

# STEP 6: Subtract 127
# ============================== EDIT THIS PART ==============================
# Change im to a float for computations
```

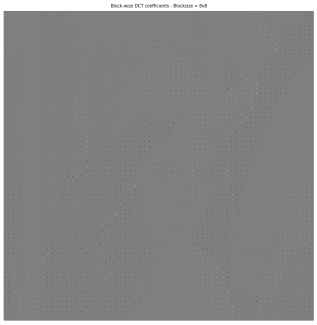
```
im = im.astype(float)
im = im - 127.0
# STEP 7: Block-wise DCT
block_size = (8, 8)
# ================================ EDIT THIS PART ================================
im_DCT = dip.utilities.block_process(im, dip.dct_2d, block_size)
# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
dip.subplot(2, 2, 3)
dip.imshow(im_DCT, 'gray')
dip.title("Block-wise DCT coefficients - Blocksize = {}x{}"
          .format(*block_size), fontsize='x-small')

c = 5
Q_table = dip.JPEG_Q_table_luminance

# STEP 8: Quantization
def step8(X):
      '''Apply X/(cQ), where X is a subblock'''
      c = 5
      Q_table = dip.JPEG_Q_table_luminance
      denominator = c*Q_table
      v = np.round(X/denominator).astype(int)#   v.astype(is
    #   pdb.set_trace()
      return v
# ================================ EDIT THIS PART ================================
im_DCT_quantized = dip.utilities.block_process(im_DCT, step8,
    block_size)
# im_DCT_quantized = dip.float_to_im(np.array(im_DCT_quantized))
# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
dip.subplot(2, 2, 4)
dip.imshow(im_DCT_quantized, 'gray')
dip.title('Quantized DCT coefficients: c={}'.format(c), fontsize='x
    -small')

# STEP 9: Entropy Coding
q_bit_stream, q_bit_stream_length, q_symbol_code_dict, _ = \
        dip.huffman_encode(im_DCT_quantized.flatten())#.reshape(-1)
    )
# ================================ EDIT THIS PART ================================
q_bit_rate = q_bit_stream_length/float(l*b)

# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
print("Bit rate of the compressed image = {:.2f} bits/pixel"
      .format(q_bit_rate))

# STEP 10: Saving the bitstream to a binary file
# ====================!!!!! DO NOT EDIT THIS PART !!!!!============================
bit_stream_file = open("CompressedSunset.bin", "wb")
q_bit_stream.tofile(bit_stream_file)
bit_stream_file.close()
```

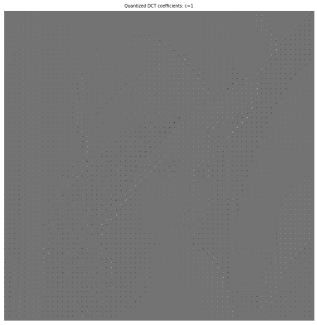
```
91
92 # STEP 11-i: Read the binary file
93 # ====================!!!!! DO NOT EDIT THIS PART
       !!!!============================
94 bit_stream_file = open("CompressedSunset.bin", "rb")
95 q_bit_stream = np.fromfile(bit_stream_file, dtype='uint8')
96 bit_stream_file.close()
97
98 # STEP 11-ii: Decoding
99 # ============================== EDIT THIS PART
       ================================
100 im_DCT_quantized_decoded = dip.huffman_decode(q_bit_stream,
101         q_symbol_code_dict)
102
103 # ====================!!!!! DO NOT EDIT THIS PART
       !!!!============================
104 im_DCT_quantized_decoded = im_DCT_quantized_decoded[:im.size]
105 im_DCT_quantized_reconstructed = im_DCT_quantized_decoded.reshape(
       im.shape)
106
107 # STEP 12: Dequantization
108 # ============================== EDIT THIS PART
       ================================
109 def step12(X):
110       '''Apply X*(cQ), where X is a subblock'''
111       c = 5
112       Q_table = dip.JPEG_Q_table_luminance
113       numerator = c*Q_table
114       return np.round(np.multiply(X,numerator)).astype(int)
115
116 im_DCT_reconstructed = dip.utilities.block_process(
       im_DCT_quantized_reconstructed, step12, block_size)
117
118 # STEP 13: Inverse DCT
119 # ============================== EDIT THIS PART
       ================================
120 im_reconstructed = dip.utilities.block_process(im_DCT_reconstructed
       , dip.transforms.idct_2d, block_size)
121
122 # STEP 14: Add 127 to every pixel
123 # ============================== EDIT THIS PART
       ================================
124 im_reconstructed = im_reconstructed + 127
125
126 # ====================!!!!! DO NOT EDIT THIS PART
       !!!!============================
127 dip.subplot(2, 2, 2)
128 dip.imshow(im_reconstructed, 'gray')
129 dip.title('Reconstructed image', fontsize='x-small')
130
131 # ====================!!!!! DO NOT EDIT THIS PART
       !!!!============================
132 im = im + 127
133
134 # STEP 15: Calculating MSE and PSNR
135 # ============================== EDIT THIS PART
       ================================
```

```
136 MSE = dip.metrics.MSE(im, im_reconstructed)
137 PSNR = dip.metrics.PSNR(im_reconstructed, im, 256)
138
139 # ====================!!!!! DO NOT EDIT THIS PART
        !!!!============================
140 print("MSE = {:.2f}".format(MSE))
141 print("PSNR = {:.2f} dB".format(PSNR))
142
143 dip.show()
```

Result

Coefficient = 1

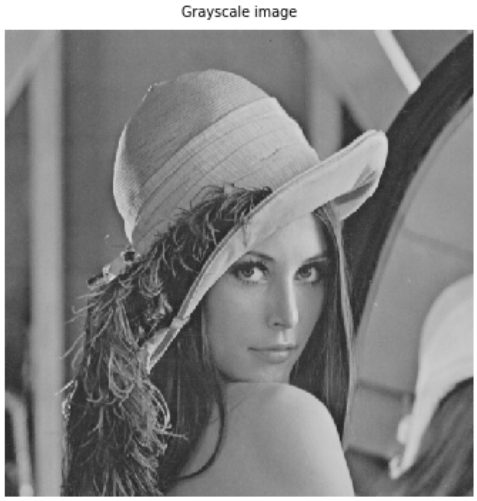


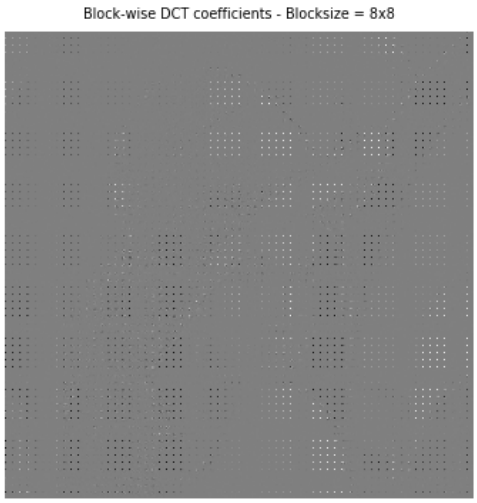


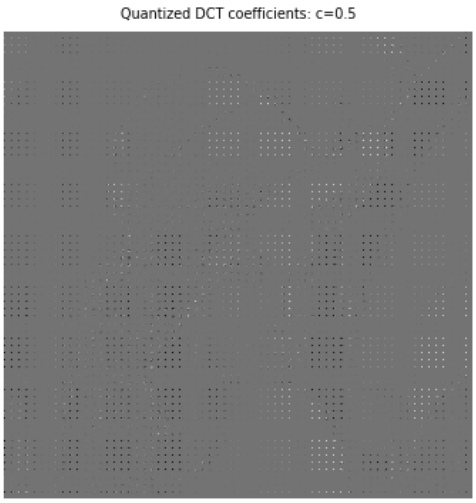

```
Entropy of the grayscale image = 7.23 bits/pixel
Bit rate of the original image = 7.26 bits/pixel
Bit rate of the compressed image = 1.39 bits/pixel
MSE = 14.00
PSNR = 36.70 dB
```

Result

Coefficient = 0.5



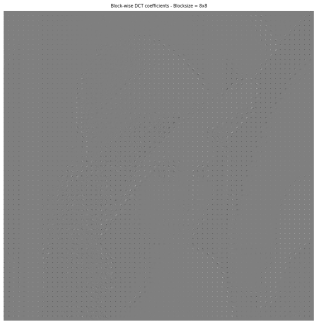


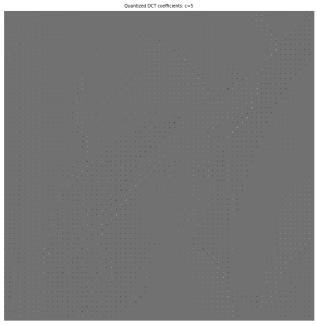


```
Entropy of the grayscale image = 7.23 bits/pixel
Bit rate of the original image = 7.26 bits/pixel
Bit rate of the compressed image = 1.64 bits/pixel
MSE = 8.72
PSNR = 38.76 dB
```

Result

Coefficient = 5






```
Entropy of the grayscale image = 7.23 bits/pixel
Bit rate of the original image = 7.26 bits/pixel
Bit rate of the compressed image = 1.11 bits/pixel
MSE = 50.70
PSNR = 31.11 dB
```