Image Processing

Assignment 3 – Report

Tang Wing Ho 鄧永豪 (310551004)

Introduction

This assignment is to experiment with the various components of a JPEG codec. The techniques used in this assignment are implemented in Python 3.8. The code included both an encoder and a decoder without implementation of read/write real JPEG file. We would do the experiment on several images in order to try the effect of different setup and input.

Methodology

1. **Block Transform Coding**
   The transform projects a vector in the source (such as colors of all the pixels in a block) onto another set of basis vectors. One advantage of block transform coding is the easiness to implement in hardware.
2. **Block Discrete Cosine Transform**

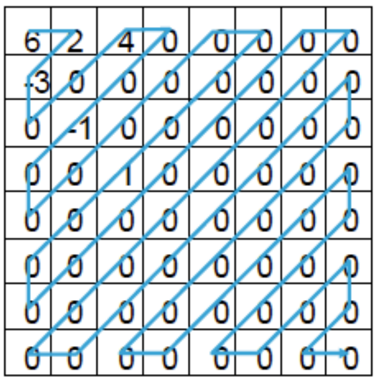
$$r(x,y,u,v) = \alpha(u)\cos\left[\frac{(2x+1)u\pi}{2n}\right]\alpha(v)\cos\left[\frac{(2y+1)v\pi}{2n}\right],$$

$$x,y,u,v = 0,1,\ldots,n-1$$

with the normalization factors:

$$\alpha(u) = \begin{cases} \sqrt{1/n}, & u=0 \\ \sqrt{2/n}, & \text{otherwise} \end{cases} \qquad \text{(similar for } \alpha(v)\text{)}$$

3. **Zigzag Coding**



6 2 -3 004 00 -10 00000 000100 ...

↓

6 (0,2) (0,-3) (2,4) (2,-1) (9,1) EOB

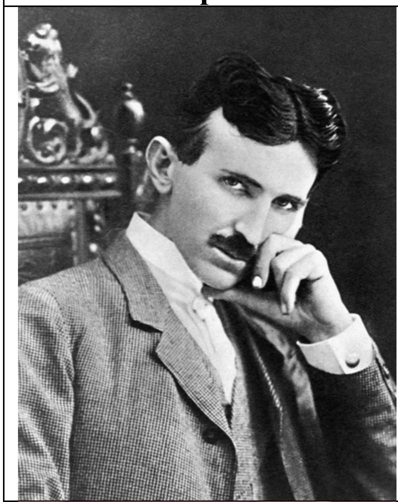
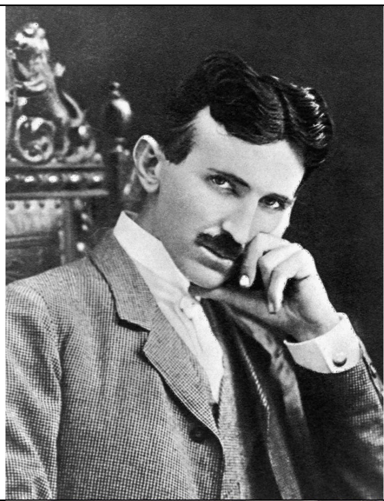
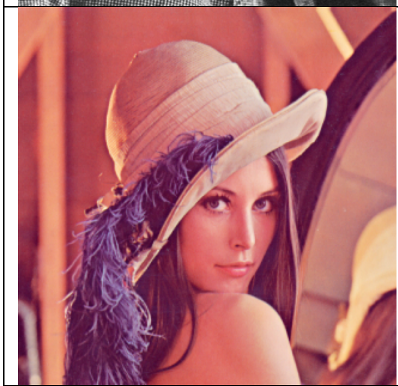
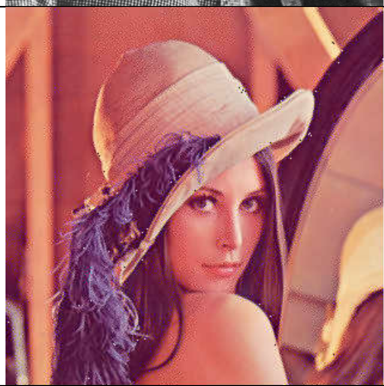
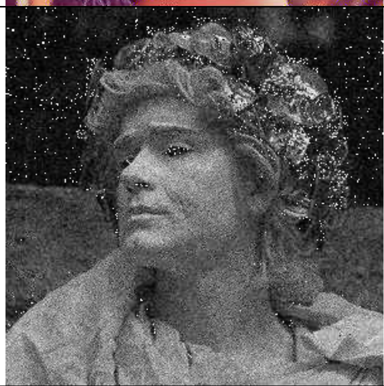
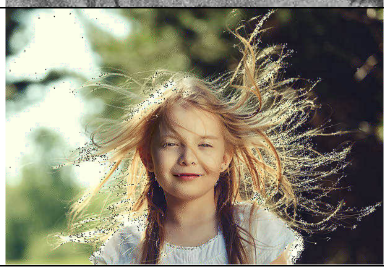
4. **Huffman Coding**
   Huffman coding determine the probability of each symbol in the source.
   Assign a node for each symbol.

Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Original source			Source reduction							
Symbol	Probability	Code	1		2		3		4	
a_2	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
a_6	0.3	00	0.3	00	0.3	00	0.3	00	0.4	1
a_1	0.1	011	0.1	011	0.2	010	0.3	01		
a_4	0.1	0100	0.1	0100	0.1	011				
a_3	0.06	01010	0.1	0101						
a_5	0.04	01011								

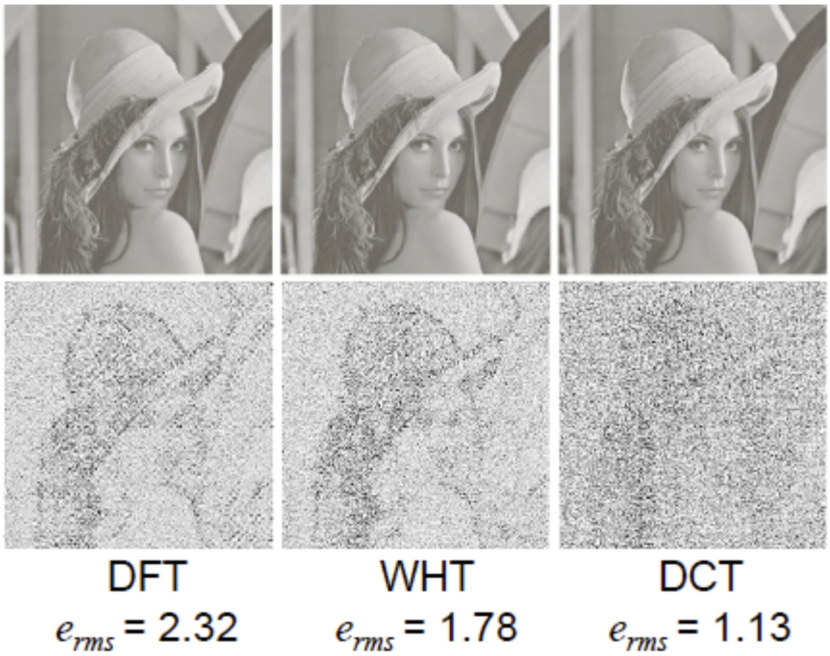
Experiments

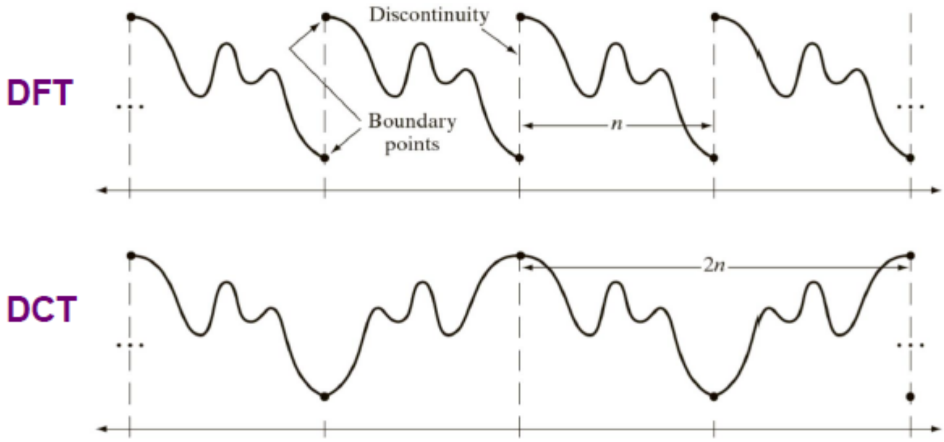
1. **Block Size: 8,**
   Transform: DCT,
   Quantization: Zigzag,
   Run-length Coding: True,
   Chromatic Subsampling: True

Input	Our Detector	MSE
		**27.40**
		**50.28**
		**76.18**
		**46.59**

Discussion

1. **DCT**





The discontinuities in DFT can cause undesired effects (wraparound errors).

2. **Huffman Coding**

Coding efficiency / compression ratio is limited by the requirement that each symbol is coded separately.

Appendix

Huffman Coding(Ref: https://github.com/ghallak/jpeg-python/blob/master/huffman.py)

```
from queue import PriorityQueue


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)
```

```python
        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
```

Main Part

```
import math
import numpy as np
import matplotlib.pyplot as plt

from huffman import HuffmanTree
from PIL import Image
from scipy import fftpack

BLOCK_SIZE = 8
LUMINANCE = 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],
    ]
)
CHROMINANCE = np.array(
    [
        [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],
    ]
)

def preprocess_img(img):
    w, h = img.size

    new_w = w
    new_h = h
    if w % 8 != 0:
        new_w = w // 8 * 8 + 8
    if h % 8 != 0:
        new_h = h // 8 * 8 + 8
```

```python
    padded_img = np.zeros((new_h, new_w, 3), dtype=np.uint8)
    padded_img[:h, :w, :] = np.asarray(img)
    padded_img = Image.fromarray(padded_img)
    # padded_img.show()
    return np.array(padded_img.convert("YCbCr")), w, h


def quantize(img, c):
    if c == 0:
        q_mat = LUMINANCE
    else:
        q_mat = CHROMINANCE

    return (img / q_mat).round().astype(np.int32)


def dequantize(img, c):
    if c == 0:
        q_mat = LUMINANCE
    else:
        q_mat = CHROMINANCE

    return img * q_mat


def zigzag_points(rows, cols):
    UP, DOWN, RIGHT, LEFT, UP_RIGHT, DOWN_LEFT = range(6)

    def move(direction, point):
        return {
            UP: lambda point: (point[0] - 1, point[1]),
            DOWN: lambda point: (point[0] + 1, point[1]),
            LEFT: lambda point: (point[0], point[1] - 1),
            RIGHT: lambda point: (point[0], point[1] + 1),
            UP_RIGHT: lambda point: move(UP, move(RIGHT, point)),
            DOWN_LEFT: lambda point: move(DOWN, move(LEFT, point)),
        }[direction](point)

    def inbounds(point):
        return 0 <= point[0] < rows and 0 <= point[1] < cols

    point = (0, 0)

    move_up = True
```

```python
    for i in range(rows * cols):
        yield point
        if move_up:
            if inbounds(move(UP_RIGHT, point)):
                point = move(UP_RIGHT, point)
            else:
                move_up = False
                if inbounds(move(RIGHT, point)):
                    point = move(RIGHT, point)
                else:
                    point = move(DOWN, point)
        else:
            if inbounds(move(DOWN_LEFT, point)):
                point = move(DOWN_LEFT, point)
            else:
                move_up = True
                if inbounds(move(DOWN, point)):
                    point = move(DOWN, point)
                else:
                    point = move(RIGHT, point)


def zigzag(mat):
    return np.array([mat[point] for point in
zigzag_points(*mat.shape)])


def undo_zigzag(zigzag):
    rows = cols = int(math.sqrt(len(zigzag)))

    block = np.empty((rows, cols), np.int32)

    for i, point in enumerate(zigzag_points(rows, cols)):
        block[point] = zigzag[i]

    return block


def run_length_encode(arr):
    last_nonzero = -1
    for i, elem in enumerate(arr):
        if elem != 0:
            last_nonzero = i
```

```python
    symbols = []

    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_required(elem)
            symbols.append((run_length, size))
            values.append(int_to_binstr(elem))
            run_length = 0
    return symbols, values


def bits_required(n):
    n = abs(n)
    result = 0
    while n > 0:
        n >>= 1
        result += 1
    return result


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


def int_to_binstr(n):
    if n == 0:
        return ""

    binstr = bin(abs(n))[2:]

    return binstr if n > 0 else binstr_flip(binstr)


def binstr_flip(binstr):
    if not set(binstr).issubset("01"):
```

```python
        raise ValueError("binstr should have only '0's and '1's")
    return "".join(map(lambda c: "0" if c == "1" else "1", binstr))


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


def cal_MSE(clean, noisy):
    diff = np.subtract(clean, noisy)
    squared = np.square(diff)
    return squared.mean()

# Encoding
original_img = Image.open("input-3.jpeg", "r")
# original_img.show()
img, original_w, original_h = preprocess_img(original_img)

h, w, c = img.shape

img_mat = np.array(img)
block_count = w // BLOCK_SIZE * h // BLOCK_SIZE

dc = np.empty((block_count, 3), dtype=np.int32)
ac = np.empty((block_count, 63, 3), dtype=np.int32)

block_idx = 0

for j in range(0, w, 8):
    for i in range(0, h, 8):
        for k in range(3):
            block = img_mat[i : i + 8, j : j + 8, k] - 128
            dct_mat = fftpack.dct(fftpack.dct(block.T, norm="ortho").T,
norm="ortho")
            quantized_mat = quantize(dct_mat, k)
            zigzag_mat = zigzag(quantized_mat)

            dc[block_idx, k] = zigzag_mat[0]
            ac[block_idx, :, k] = zigzag_mat[1:]

        block_idx += 1

H_DC_Y = HuffmanTree(np.vectorize(bits_required)(dc[:, 0]))
```

```
H_AC_Y = HuffmanTree(flatten(run_length_encode(ac[i, :, 0])[0] for i in
range(block_count)))
if c == 3:
    H_DC_C = HuffmanTree(np.vectorize(bits_required)(dc[:, 1:].flat))
    H_AC_C = HuffmanTree(flatten(run_length_encode(ac[i, :, j])[0] for
i in range(block_count) for j in [1, 2]))

if c == 3:
    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(),
    }
else:
    tables = {
        "dc_y": H_DC_Y.value_to_bitstring_table(),
        "ac_y": H_AC_Y.value_to_bitstring_table(),
    }

# Decoding
width_block = w // BLOCK_SIZE
height_block = h // BLOCK_SIZE
img_mat = np.empty((h, w, 3), dtype=np.uint8)

for block_index in range(block_count):
    j = block_index // min(width_block, height_block) * BLOCK_SIZE if w
> h else block_index // max(width_block, height_block) * BLOCK_SIZE
    i = block_index % min(width_block, height_block) * BLOCK_SIZE if w
> h else block_index % max(width_block, height_block) * BLOCK_SIZE

    for k in range(3):
        zigzag_mat = [dc[block_index, k]] + list(ac[block_index, :, k])
        quant_mat = undo_zigzag(zigzag_mat)
        dct_mat = dequantize(quant_mat, k)
        block = fftpack.idct(fftpack.idct(dct_mat.T, norm="ortho").T,
norm="ortho")
        img_mat[i : i + BLOCK_SIZE, j : j + BLOCK_SIZE, k] = block +
128

unpadded_img = img_mat[:original_h, :original_w, :]
image = Image.fromarray(unpadded_img, "YCbCr").convert("RGB")
image.show()
```

```python
mse = cal_MSE(image, original_img)

print(f"MSE: {mse}")
```