Homework 03

0856030, 林正偉, aunbdaunbd@gmail.com

Github: https://github.com/lincw6666/Digital_Image_Processing

I. INTRODUCTION

In homework 03, we are asked to implement image compression with WHT, DCT and DFT. Then, we compress some images and compare the results with different block sizes, transform types and quantization methods.

There are 4 following sections: method review, experiments, discussion and code section. In the method review section, I'll talk about the basic function of WHT, DCT and DFT. In the experiments section, I'll show the results with different transform types, block sizes and quantization methods. In the discussion section, I'll talk about my observation to the experiments. In the code section, I'll paste all my codes.

II. METHOD REVIEW

A. *Walsh-Hadamard Transform*

First, we get the Hadamard matrix by the following equation

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$H_{2^n} = \begin{bmatrix} H_{2^{n-1}} & H_{2^{n-1}} \\ H_{2^{n-1}} & -H_{2^{n-1}} \end{bmatrix}, where\ n \geq 3.$$

Next, we calculate the sign changes of each row and sort them. Now we get the Walsh matrix. Please note that we need to divide Walsh matrix by its block size.

B. *Discrete Cosine Transform*

DCT is defined as below:

$$\mathrm{F(u,v)} = w_x w_y \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left(\frac{(x+0.5)u\pi}{N}\right) \cos\left(\frac{(y+0.5)v\pi}{N}\right)$$
$$, where\ N\ is\ the\ block\ size, w_i = \begin{cases} \frac{1}{\sqrt{N}}, for\ i = 0 \\ \frac{2}{\sqrt{N}}, for\ i > 0 \end{cases}.$$

C. *Discrete Fourier Transform*

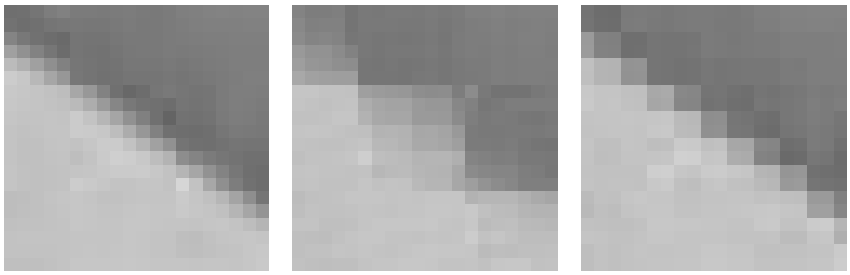
DFT is defined as below:

$$\mathrm{F(u,v)} = \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j\left(\frac{2\pi(ux+vy)}{N}\right)}, \text{where N is the block size.}$$

III. EXPERIMENTS

A. Same block size with different transform type

Figure 1. shows the results of different transform types with block size as 8 and 16 first coefficients quantization. We can see that DCT performs the best and DFT performs the worst.

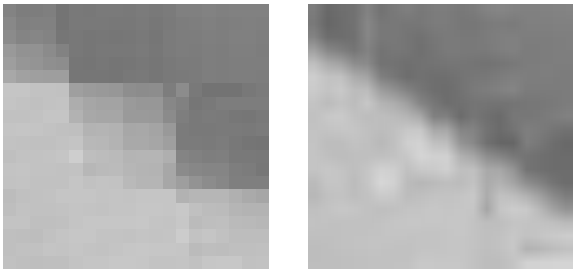


(a) DCT (b) DFT (c) WHT

Figure 1. Results of compressing the famous image – Lena. I used different transform types with block size = 8 and 16 first coefficients quantization. It's a 20 x 20 cropped image.

B. Same transform type with different quantization method

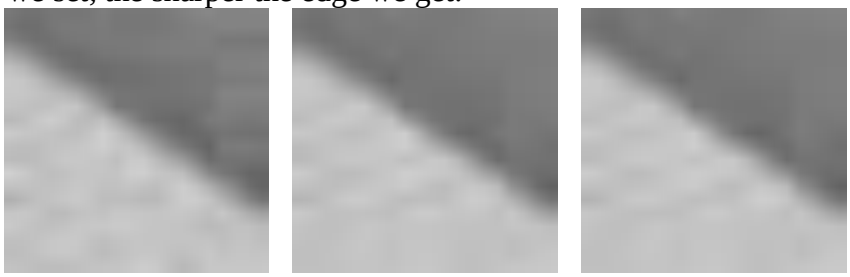
Figure 2. shows the results of different quantization methods with DFT and block size = 8. We can see that K largest gets a better result.



(a) K first (b) K Largest

Figure 2. Results of different quantization methods with DFT and block size = 8.

Figure 3. shows the results of different total coefficient bits with DFT and block size = 8. We can see that the more bits we set, the sharper the edge we get.

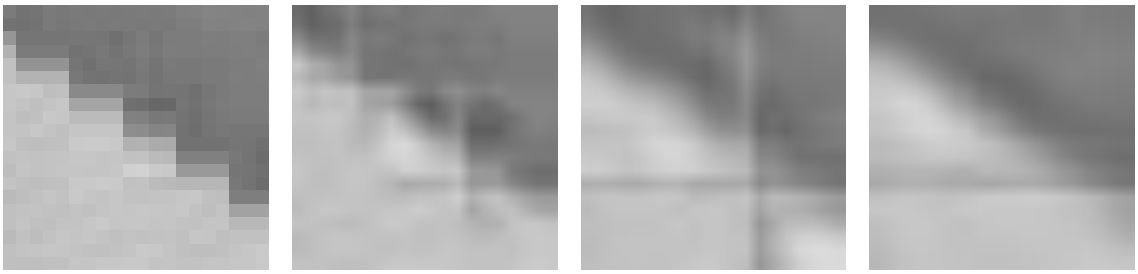


(a) 128 (b) 256 (c) 512

Figure 3. Results of different total coefficient bits with DFT and block size = 8.

C. Same transform type with different quantization method

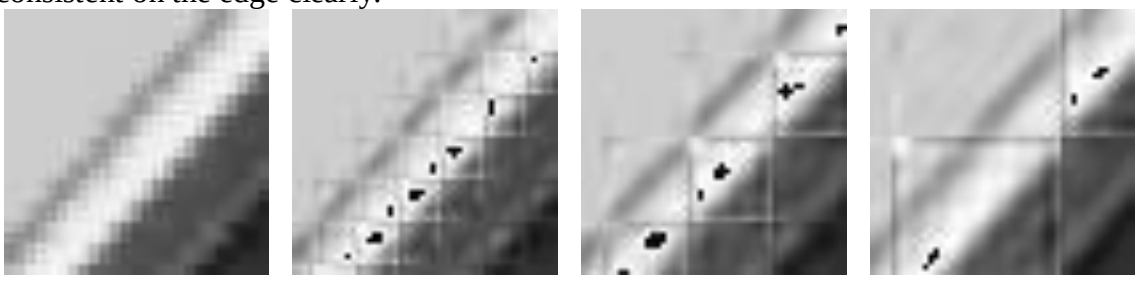
Figure 4. shows the results of different block sizes with DFT and K largest coefficient quantization. We can see the inconsistent on the boundary of the block. This effect occurs on edges.



(a) Block size = 4 (b) Block size = 8 (c) Block size = 16 (d) Block size = 32

Figure 4. Results of different block sizes with DFT and (block size ^ 2 / 2) largest coefficient quantization.

Figure 5. shows the results of compressed house image with the same settings as above. We can see the inconsistent on the edge clearly.



(a) Block size = 4 (b) Block size = 8 (c) Block size = 16 (d) Block size = 32

Figure 5. Results of compressed house image. The settings are the same as figure 3.

IV. DISCUSSION

I think that I explain the results with different transform types, block sizes and quantization methods in the experiment section clearly. I'll focus on the RMS and SNR value in this section.

In conclusion, DCT is better than WHT, and WHT is better than DFT. We can see the RMS and SNR with block size = 8 and use 32 largest coefficients. The results are listed below:

	RMS	SNR(dB)
WHT	1. 8851663272238945	14. 707387799145389
DCT	1. 2798886393742304	18. 08833525448537
DFT	2. 359984464432612	12. 769496732900734

Table 1. The RMS and SNR(dB) of compressing Lena with different transform type.

V. CODE SECTION

A. **simple_jpeg.py:** Implement transform types and the quantization methods.

```
import gc   # For garbage collection.
import cv2
from math import ceil
import numpy as np


class Compression:


    class Transform:
```

```
class Zigzag:

    def __init__(self, block_size):
        self.block_size = block_size
        self.i = 0
        self.j = 0
        self.j_end = 0
        self._set_j()

    def _set_j(self):
        j_start = min(self.block_size-1, self.i)
        self.j_end = max(self.i-self.block_size, -1)
        self.j = j_start

    def __iter__(self):
        return self

    def __next__(self):
        if self.j <= self.j_end:
            self.i += 1
            self._set_j()
        if self.i >= 2*self.block_size-1:
            raise StopIteration
        if self.i%2 == 0:
            ret1, ret2 = self.j, self.i-self.j
        else:
            ret1, ret2 = self.i-self.j, self.j
        self.j -= 1
        return ret1, ret2


def __init__(self, method_name):
    method_name_pair = {
        'WHT': self._wht,
        'DCT': self._dct,
        'DFT': self._dft
    }
    self.method = method_name_pair[method_name]
    self.basic_block = None
    self.num_of_block = None

# Wrapper function for all forward transforms.
def transform(self, img, block_size):
    return self.method(img, block_size)

# Wrapper function for all inverse transforms.
def inverse_transform(self, coef, block_size):
    assert self.basic_block is not None, \
```

```
            '[inverse_transform] Need to do forward transform first!!'
        h, w = self.num_of_block
        img = np.zeros((h*block_size, w*block_size))
        for i in range(h):
            for j in range(w):
                y, x = i*block_size, j*block_size
                img[y:(y+block_size), x:(x+block_size)] = \
                    (np.sum(self.basic_block*coef[i*w+j, :, None, None], \
                        axis=0) / block_size).real
        return np.round(img).astype(np.uint8)

    def _do_transform(self, img, block_size):
        h = ceil(img.shape[0]/block_size)
        w = ceil(img.shape[1]/block_size)
        self.num_of_block = (h, w)
        coef = np.zeros((h*w, block_size**2), dtype=self.basic_block.dtype)
        for i in range(h):
            for j in range(w):
                y, x = i*block_size, j*block_size
                coef[i*w+j] = np.sum(
                    self.basic_block * \
                        img[y:(y+block_size), x:(x+block_size)],
                    axis=(1, 2)) / block_size
        return coef

    # Follow the zig-zag order to walk through a matrix.
    def _zigzag_walk(self, func, block_size, **kwargs):
        idx = 0
        for j, k in iter(self.Zigzag(block_size)):
            func(idx, j, k, block_size, **kwargs)
            idx += 1

    # Walsh-Hadamard Transform
    def _wht(self, img, block_size):

        def _permutation(num, width):
            # Transform to gray code.
            num = num ^ (num>>1)
            # Apply bit reverse.
            return int('{:0{width}b}'.format(num, width=width)[::-1], 2)

        def _wht_basic_block(idx, j, k, block_size, mat):
            self.basic_block[idx] = mat[
                j*block_size:(j+1)*block_size,
                k*block_size:(k+1)*block_size]

        # Build the Hadamard matrix.
        size = block_size ** 2
        hadamard_2 = np.array([[1, 1], [1, -1]])
        hadamard = np.ones(1)
```

```
    for _ in range(int(np.log2(size))):
        hadamard = np.kron(hadamard_2, hadamard)
    # Get the permutation of Hadamard matrix, which makes a Hadamard
    # matrix become a Walsh matrix.
    perm = np.array(
        [_permutation(i, int(np.log2(size))) for i in range(size)])
    walsh = hadamard[perm]

    # Build the basic blocks.
    self.basic_block = np.zeros((size, block_size, block_size))
    self._zigzag_walk(_wht_basic_block, block_size, mat=walsh)

    # Apply the transformation.
    return self._do_transform(img, block_size)

# Discrete Cosine Transform
def _dct(self, img, block_size):

    def _dct_basic_block(idx, j, k, block_size, lookup):

        def _update_lookup(x, j):
            tmp_val = np.cos((x+0.5)*j*np.pi/block_size)
            # Set very small number to zero.
            if -1e-14 < tmp_val < 1e-14:
                return 0.0
            return tmp_val

        for x in range(block_size):
            for y in range(block_size):
                # Fill up the lookup table.
                if lookup[x, j] == 0:
                    lookup[x, j] = _update_lookup(x, j)
                if lookup[y, k] == 0:
                    lookup[y, k] = _update_lookup(y, k)
                # Store the value to basic blocks.
                self.basic_block[idx, x, y] = \
                    lookup[x, j] * lookup[y, k]
        w = 1
        if j!=0 and k!=0:
            w = 2
        elif j!=0 or k!=0:
            w = np.sqrt(2)
        if w != 1:
            self.basic_block[idx] = self.basic_block[idx] * w


    # Build the basic blocks.
    self.basic_block = np.zeros(
        (block_size**2, block_size, block_size))
    lookup = np.zeros((block_size, block_size))
```

```
        self._zigzag_walk(_dct_basic_block, block_size,lookup=lookup)

        # Apply the transformation.
        return self._do_transform(img, block_size)

    # Discrete Fourier Transform
    def _dft(self, img, block_size):

        def _dft_basic_block(idx, j, k, block_size, lookup):
            for x in range(block_size):
                for y in range(block_size):
                    tmp = j*x + k*y
                    # Fill up the lookup table.
                    if lookup[tmp] == 0:
                        tmp_dft = np.exp(-1j*2*np.pi*tmp/block_size)
                        # Set very small number to zero.
                        tmp_real, tmp_imag = tmp_dft.real, tmp_dft.imag
                        if -1e-14 < tmp_real < 1e-14:
                            tmp_real = 0.0
                        if -1e-14 < tmp_imag < 1e-14:
                            tmp_imag = 0.0
                        # Store the correct value to lookup table.
                        lookup[tmp] = tmp_real + 1j*tmp_imag
                    self.basic_block[idx, x, y] = lookup[tmp]

        # Build the basic blocks.
        self.basic_block = np.zeros(
            (block_size**2, block_size, block_size), dtype=np.complex)
        lookup = np.zeros(
            (2*block_size**2-4*block_size+3, ), dtype=np.complex)
        self._zigzag_walk(_dft_basic_block, block_size, lookup=lookup)

        # Apply the transformation.
        coef = self._do_transform(img, block_size)
        # Build the basic blocks for inverse transform.
        self.basic_block.imag *= -1
        return coef


class Quantization:

    # @N_K: it'll be N if we use 'Total N' method. Else, it's K.
    def __init__(self, method_name, N_K):
        method_name_pair = {
            'K First': self._k_first,
            'K Largest': self._k_largest,
            'Total N': self._total_N
        }
        self.method = method_name_pair[method_name]
        self.N_K = N_K
```

```
def quantization(self, coef, block_size):
    self.method(coef, block_size, self.N_K)

# Choose K first coefficient.
def _k_first(self, coef, block_size, K):
    coef[:, K:] = 0

# Choose K largest coefficient.
def _k_largest(self, coef, block_size, K):
    index = np.argsort(np.abs(coef), axis=1)
    np.put_along_axis(coef, index[:, :block_size**2-K], 0, axis=1)

# Total N bits for coefficients. How many bits for a specific coefficient
# depend on its variances over all blocks.
# Assume that the variance of a coefficient is vi, we define qi = log2(vi).
# Then the number of bits for that coefficient is ni = round(N*qi/sum(qi)).
def _total_N(self, coef, block_size, N):
    # Variance of each coefficient.
    variance = np.log2(np.var(coef, axis=0))
    variance = np.where(variance<0, 0, variance)

    # Bits for a specific coefficient.
    bits = np.round(variance*N/np.sum(variance))
    bits = np.where(bits<0, 0, bits)
    idx = np.where(bits!=0)[0][None, :]
    np.put_along_axis(coef, np.where(bits==0)[0][None, :], 0, axis=1)

    # Quantize each coefficient.
    tmp_coef = np.take_along_axis(coef, idx, axis=1)
    min_coef = np.min(tmp_coef, axis=0)
    coef_range = np.max(tmp_coef, axis=0) - min_coef
    if coef_range.dtype != complex:
        bits = 2**bits[idx]
        interval = np.where(coef_range!=0, coef_range/bits, 0)
        factor = np.floor((tmp_coef-min_coef) / interval)
        tmp_coef = \
            (2*min_coef + interval*factor + interval*(factor+1))/2
    else:
        bits = 2**(np.round(bits[idx]/2))
        interval_real = \
            np.where(coef_range.real!=0, coef_range.real/bits, 0)
        interval_imag = \
            np.where(coef_range.imag!=0, coef_range.imag/bits, 0)
        factor_real = np.where(
            interval_real != 0,
            np.floor((tmp_coef-min_coef).real / interval_real),
            0
        )
        factor_imag = np.where(
```

```
                interval_imag != 0.,
                np.floor((tmp_coef-min_coef).imag / interval_imag),
                0
            )
            tmp_coef = (
                2*min_coef + \
                interval_real*factor_real + \
                interval_real*(factor_real+1) + \
                1j*interval_imag*factor_imag + \
                1j*interval_imag*(factor_imag+1))/2
        coef[:, idx[0]] = tmp_coef


    def __init__(self, img_path, block_size, transform_method='WHT',
        quantization_method='K First', N_K=32):
        assert quantization_method=='Total N' or N_K <= block_size**2, \
            '[__init__] N_K must <= block_size^2 !'
        self.img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        self.coef = None
        self.block_size = block_size
        self.transform = self.Transform(transform_method)
        self.quantization = self.Quantization(quantization_method, N_K)

    # Compression
    def compress(self):
        self.coef = self.transform.transform(self.img, self.block_size)
        self.img = None
        gc.collect()
        self.quantization.quantization(self.coef, self.block_size)

    # Reconstruction
    def reconstruct(self):
        self.img = self.transform.inverse_transform(self.coef, self.block_size)
        self.coef = None
    gc.collect()
```

B. **evaluation.py**: Implement RMS and SNR(dB).

```
import numpy as np


def RMS(img, ref_img):
    assert img.shape == ref_img.shape, \
        '[RMS] The shape of img and ref_img must be consistant!'
    total_pixels = img.shape[0] * img.shape[1]
    return np.sqrt(np.sum((img-ref_img)**2)/total_pixels)


def SNR(img, ref_img):
    assert img.shape == ref_img.shape, \
```

```
        '[RMS] The shape of img and ref_img must be consistant!'
    return 10 * np.log10(np.sum(img**2)/np.sum((img-ref_img)**2))
```

C. **main.py**

```
import cv2
from simple_jpeg import Compression
from evaluation import RMS, SNR

if __name__ == '__main__':
    img_path = 'images/lena.bmp'
    origin_img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    methods = ['WHT','DCT', 'DFT']
    block_size = [4, 4, 4]
    N_K = [512, 512, 512]
    for i in range(len(methods)):
        compress = Compression(
            img_path, block_size[i],
            transform_method=methods[i],
            quantization_method='Total N',
            N_K=N_K[i])

        # Apply compression and reconstruct it.
        compress.compress()
        compress.reconstruct()
        cv2.imwrite(f'outputs/{methods[i]}.bmp', compress.img)

        # Evaluate by RMS and SNR.
        rms = RMS(compress.img, origin_img)
        snr = SNR(compress.img, origin_img)
        print(f'RMS: {rms}, SNR(dB): {snr}')
```