EEL 5820: Digital Image Processing
Fall 2022

Homework # 2

Image Size Reduction

Submitted by:

Your name: Maryna Veksler

PID#: 5848285

Department of Electrical and Computer Engineering

Date: ___12/11/2022_____

Objective

To implement image size reduction algorithm and determine its effect on the image quality

Method

Image size reduction is often required in the digital image processing as a pre-processing step. It can be implemented by averaging 4 neighboring pixels and generating a new reduced image, such that every 4 original pixels correspond to a single reduced pixel.

Results

Example 1:

Figure 1.1 Original Image (256, 256, 3)

Figure 1.2 Resized image (128, 128, 3)

Figure 1.3 Resized image (128, 128, 3); enlarged

Figure 1.4 Resized image (64, 64, 3); enlarged

Figure 1.5 Resized image (64, 64, 3); enlarged

Example 2:

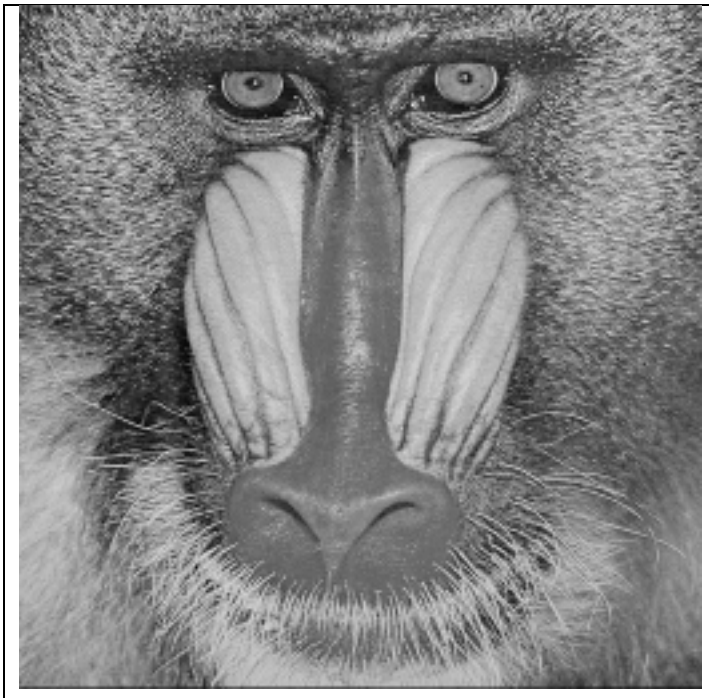


Figure 2.1 Original Image (256, 256, 1)

Figure 2.2 Resized image (128, 128, 1)

Figure 2.3 Resized image (128, 128, 1); enlarged

Figure 2.4 Resized image (64, 64, 1); enlarged

Figure 2.5 Resized image (64, 64, 1); enlarged

Discussion

The image size reduction approach is implemented for both RGB and gray scale images. When implementing for the 3-channel image, it is required to do the reduction operation via pixel averaging for each plane independently before combining it together in the final matrix. Additionally, it evident that the size reduction makes the image more blurry

Program

```
import numpy as np
import cv2

def reduce_plane(im_data, first_x, second_x, first_y, second_y, height, width):
    first_px = im_data[first_x][first_y]
    if second_x < height and second_y < width:
        second_px = im_data[second_x][first_y]
        third_px = im_data[first_x][second_y]
        forth_px = im_data[second_x][second_y]
        average_px = (first_px + second_px + third_px + forth_px)/4

    elif second_x < height:
        second_px = im_data[second_x][first_y]
        average_px = (first_px + second_px + 0 + 0)/4
    elif second_y < width:
        third_px = im_data[first_x][second_y]

        average_px = (first_px + 0 + third_px + 0)/4
    else:
```

```
        average_px = (first_px + 0 + 0 + 0)/4

    return average_px

if __name__ == "__main__":

    image_path =
"/Users/marynavek/Projects/ImageProcessing/reduced_version_2.jpg"

    original_image_data = cv2.imread(image_path)

    original_image_shape = np.shape(original_image_data)

    if len(original_image_shape) < 3:

        gray_image = original_image_data
        gray_image = np.float32(gray_image) / 255.0

        height, width = np.shape(gray_image)
        reduced_image = []

        for row in range(0,height, 2):
            reduced_row = []
            first_x = row
            second_x = row + 1

            for col in range(0, width, 2):
                first_y = col

                second_y = col + 1

                average_px = reduce_plane(gray_image, first_x, second_x,
first_y, second_y, height, width)
                reduced_row.append(average_px)

        if len(reduced_image) < 1:
            reduced_image = np.array(reduced_row)
        else:
            reduced_image = np.vstack((reduced_image, np.array(reduced_row)))

        cv2.imwrite("reduced_version.jpg", reduced_image* 255.0)

    else:
        original_image_data = np.float32(original_image_data) / 255.0

        height, width, channels = np.shape(original_image_data)

        reduced_r_plane = []
```

```
        reduced_g_plane = []
        reduced_b_plane = []

        blue_plane = original_image_data[:,:,0]

        green_plane = original_image_data[:,:,1]

        red_plane = original_image_data[:,:,2]

        reduced_blue = []
        for row in range(0, height,2):
            reduced_row = []
            first_x = row
            second_x = row + 1

            for col in range(0, width, 2):
                first_y = col
                second_y = col + 1

                average_pixel = reduce_plane(blue_plane, first_x, second_x,
first_y, second_y, height, width)

                reduced_row.append(average_pixel)

            if len(reduced_blue) < 1:
                reduced_blue = np.array(reduced_row)
            else:
                reduced_blue = np.vstack((reduced_blue, np.array(reduced_row)))

        reduced_green = []
        for row in range(0, height,2):
            reduced_row = []
            first_x = row
            second_x = row + 1

            for col in range(0, width, 2):
                first_y = col
                second_y = col + 1

                average_pixel = reduce_plane(green_plane, first_x, second_x,
first_y, second_y, height, width)

                reduced_row.append(average_pixel)

            if len(reduced_green) < 1:
                reduced_green = np.array(reduced_row)
            else:
```

```
                reduced_green = np.vstack((reduced_green,
np.array(reduced_row)))


        reduced_red = []
        for row in range(0, height,2):
            reduced_row = []
            first_x = row
            second_x = row + 1

            for col in range(0, width, 2):
                first_y = col
                second_y = col + 1

                average_pixel = reduce_plane(red_plane, first_x, second_x,
first_y, second_y, height, width)

                reduced_row.append(average_pixel)

            if len(reduced_red) < 1:
                reduced_red = np.array(reduced_row)
            else:
                reduced_red = np.vstack((reduced_red, np.array(reduced_row)))

        resize_img = np.zeros((reduced_red.shape[0], reduced_red.shape[1],
channels),np.uint8)
        resize_img[:,:,0] = reduced_blue* 255.0
        resize_img[:,:,1] = reduced_green* 255.0
        resize_img[:,:,2] = reduced_red* 255.0

        cv2.imwrite("reduced_version.jpg", resize_img)
```

EEL 5820: Digital Image Processing
Fall 2022

Homework # 3

Discrete Fourier Transform (DFT)

Submitted by:

Your name: Maryna Veksler

PID#: 584825

Department of Electrical and Computer Engineering

Date: _____

Objective

To Discrete Fourier Transform (DFT) and assess the time and computational power required.

Method

We implement DFT as follows

$$F[k,l]=\frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f[m,n]e^{-j2\pi\left(\frac{k}{M}m+\frac{l}{N}n\right)}$$

The DFT is used in image and signal processing to visualize the signal and determine its spectrum. While DFT has both imaginary and real part, in this work, we only focus on the real part.

Results

Example 1:
Execution time provided in seconds
16x16: Total execution time is: 2.211790208
32x32: Total execution time is: 37.044587666999995
64x64: Total execution time is: 615.21571925
128x128: Total execution time is: 10075.660750041

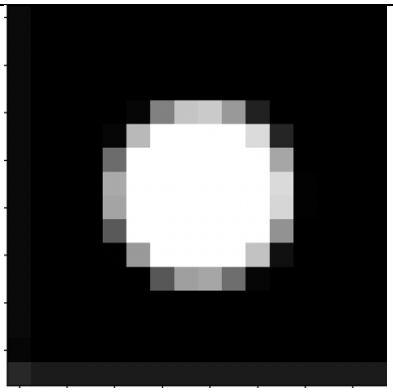


Figure 1.1 Original Image (16x16)

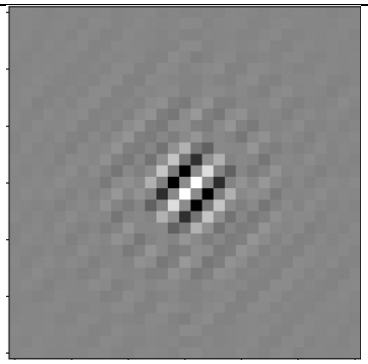


Figure 1.2 DFT

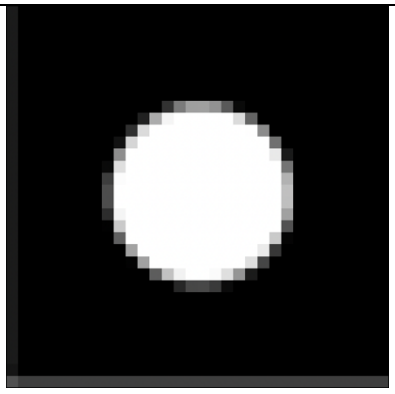


Figure 1.3 Original Image (32x32)

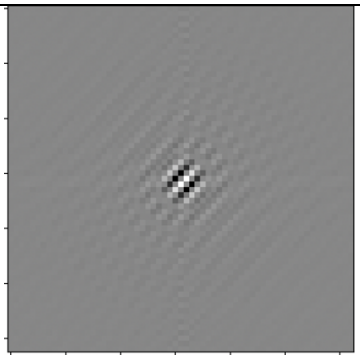


Figure 1.4 DFT

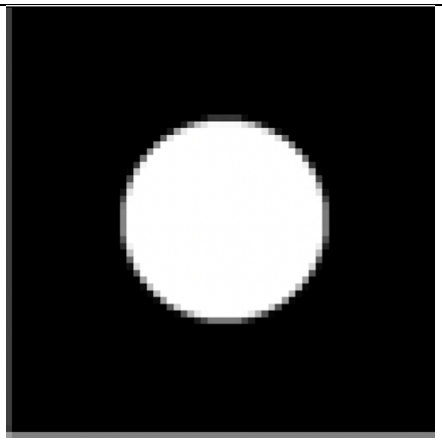


Figure 1.3 Original Image (64x64)

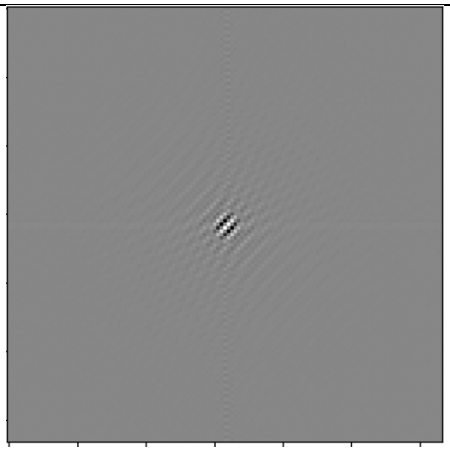


Figure 1.6 DFT

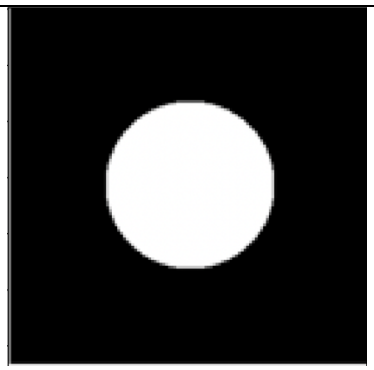


Figure 1.3 Original Image (128x128)

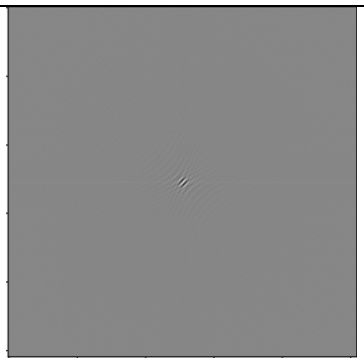


Figure 1.8 Laplacian of Gaussian function filtered Image for $\sigma = 128$

Example 2:

16x16: Total execution time is: 2.1512340420000022
32x32: Total execution time is: 37.413597417000005
64x64: Total execution time is: 616.195894667

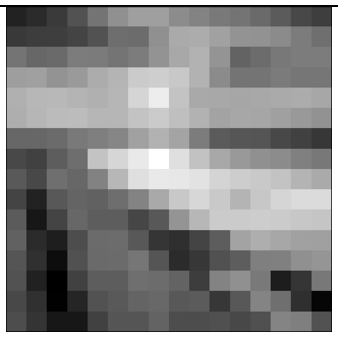


Figure 1.1 Original Image (16x16)

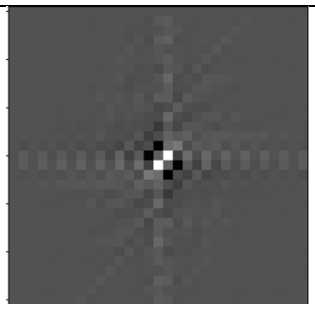


Figure 1.2 DFT

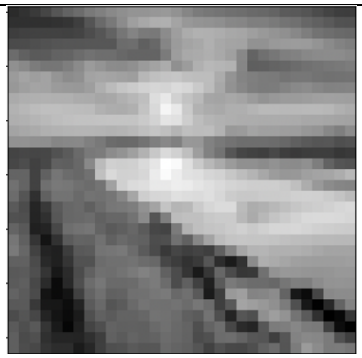


Figure 1.3 Original Image (32x32)

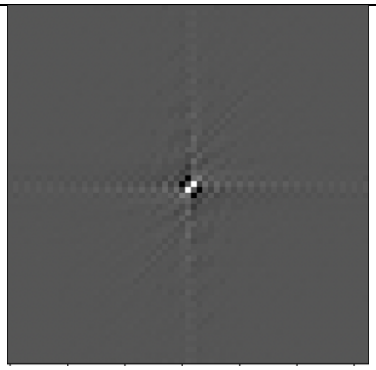


Figure 1.4 DFT

Figure 1.3 Original Image (64x64)

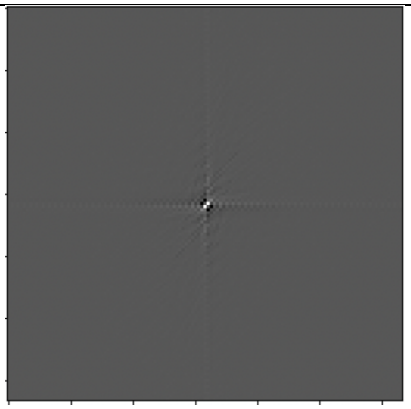


Figure 1.6 DFT

Discussion

In this homework, we were implementing DFT. The experiments demonstrated that regardless of the image content, the transform stores the image in the middle. For the example with a circle, the transform had an interesting shape – while information was the most dense in the center of the transform, it had an ellipsoid-like lines bounded to the middle and opening in the direction to the top left and bottom right.

We also timed the execution of the transform. While it can be used robustly for images of size 16x16 and 32x32, starting from the dimensions of 64x64, the computational time increases drastically to approximately 10 minutes. Moreover, it takes 167 2 hours 48 minutes to compute DFT for 128x128 image. We did not attempt to compute DFT for 256x256 and 512x512 due to limits in the computational power available.

Program

```
from matplotlib import pyplot as plt
import numpy as np
import cv2
import cmath
from PIL import Image
from timeit import default_timer as timer

if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natual_im_1.jpg"

    image = Image.open(image_path).convert("L")

    image = image.resize((64,64))
    plt.imshow(image, cmap='gray')
    plt.show()
    f = np.asarray(image)
    M, N = np.shape(f)

    P,Q = M*2-1,N*2-1
    shape = np.shape(f)

    fp = np.zeros((P, Q))

    fp[:shape[0],:shape[1]] = f

    fpc = np.zeros((P, Q))
    for x in range(P):
        for y in range(Q):
            fpc[x,y]=fp[x,y]*np.power(-1,x+y)


    def DFT2D(padded):
        M,N = np.shape(padded)
        dft2d = np.zeros((M,N),dtype=complex)
        for k in range(M):
            for l in range(N):
```

```
                sum_matrix = 0.0
                for m in range(M):
                    for n in range(N):
                        e = cmath.exp(- 2j * np.pi * (float(k * m) / M +
float(l * n) / N))
                        sum_matrix +=  padded[m,n] * e
                dft2d[k,l] = sum_matrix
        return dft2d

    time_start = timer()
    dft2d = DFT2D(fpc)
    time_end = timer()
    time_elapsed = time_end - time_start
    print(f"Total execution time is: {time_elapsed}")
    plt.imshow(dft2d.real, cmap='gray')
    plt.show()
    cv2.imwrite("dft.jpg", dft2d.real)
```

EEL 5820: Digital Image Processing
Fall 2022

Homework # 4

Image Transforms

Submitted by:

Your name: Maryna Veksler

PID#: 5848285

Department of Electrical and Computer Engineering

Date: _____12/11/2022_____

Objective

To implement Discrete Cosine, Walsh, and Haar Transform and visually compare the differences in information compression.

Method

The Discrete Cosine Transform for the image is implemented as follows

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where $\alpha(u)$ and $\alpha(v)$ are normalizing scale factors to ensure orthogonal transformation. They have fixed values of 1/sqrt(2) if u is 0, and 1 otherwise.

The Walsh (Hadamard) transform for the image is denoted as

$$W(u,\ v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,\ y)\prod_{i=0}^{n-1}(-1)^{[b_i(x)b_{n-1-i}(u)+b_i(y)b_{n-1-i}(v)]}$$

where $b_i(x)$ indicates the ith bit of the binary representation of x.

The Haar transform for the image is denoted as

$$\mathrm{HA}(0,\ 0,\ x) = \frac{1}{\sqrt{N}}$$

$$\mathrm{HA}(r,\ m,\ x) = \begin{cases} \dfrac{2^{r/2}}{\sqrt{N}} & \dfrac{m-1}{2^r} \leqslant x < \dfrac{m-\frac{1}{2}}{2^r} \\ -\dfrac{2^{r/2}}{\sqrt{N}} & \dfrac{m-\frac{1}{2}}{2^r} \leqslant x < \dfrac{m}{2^r} \\ 0 & \text{elsewhere} \end{cases}$$

where x is in range [0,1), r is in range [0, $\log_2(N)$), and m is in range [1, 2^r].
For each case, we implement the forward kernel to visually examine the differences. Next, we apply transforms to the different type of images to examine the difference in data packing.

Results

Example 1: Kernels

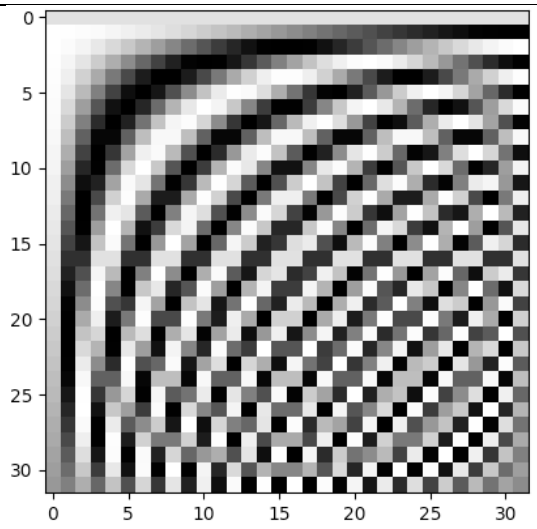


Figure 1.1 DCT 32x32 Kernel

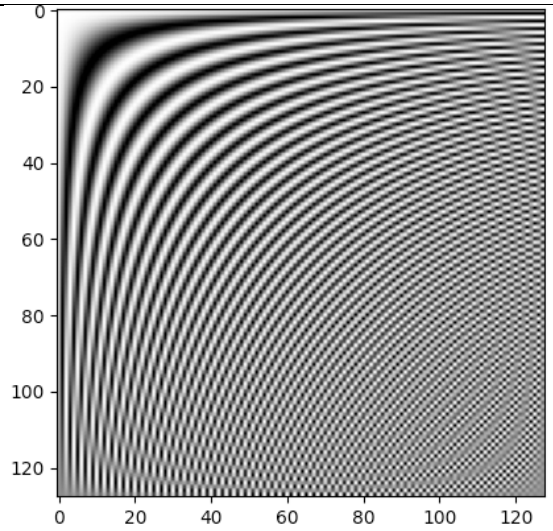


Figure 1.2 DCT 128x128 Kernel

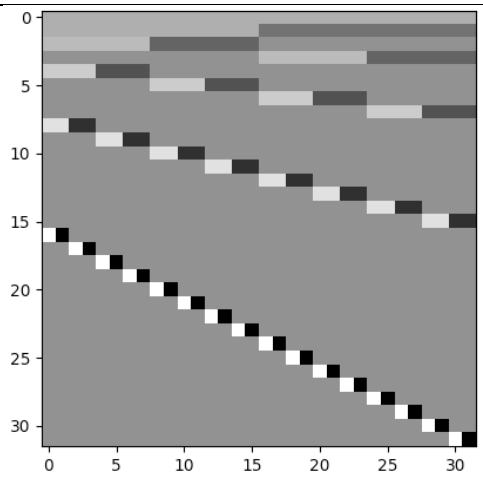


Figure 1.3 Haar 32x32 Kernel

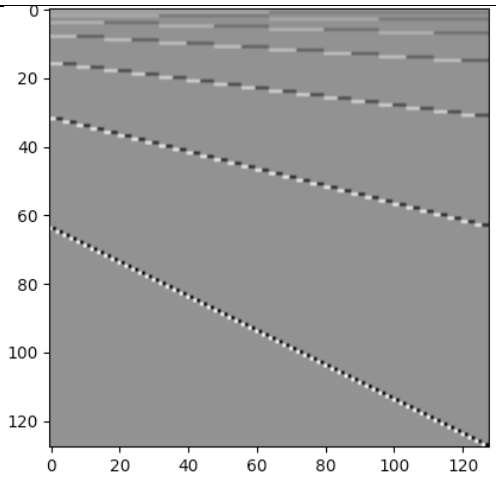


Figure 1.4 Haar 128x128 Kernel

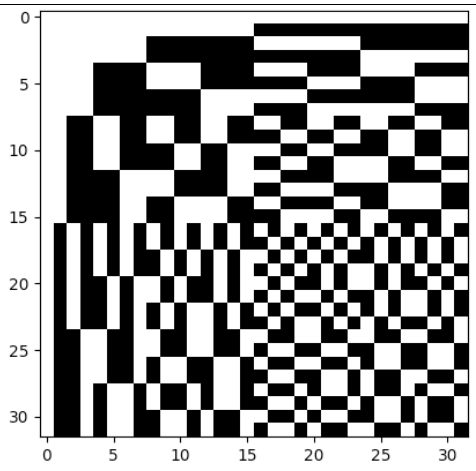


Figure 1.5 Walsh 32x32 Kernel

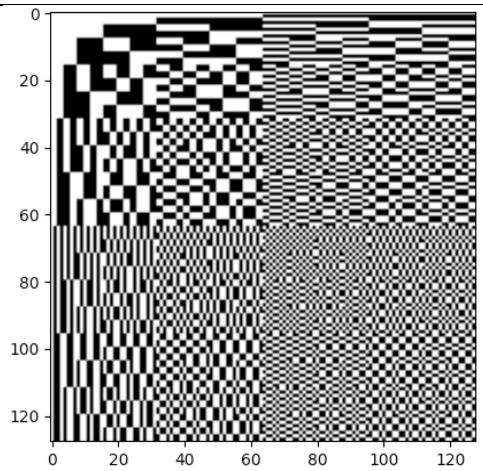


Figure 1.6 Walsh 128x128 Kernel

Example 2: Synthetic Image Texture

Figure 2.1 Original Image (128x128)

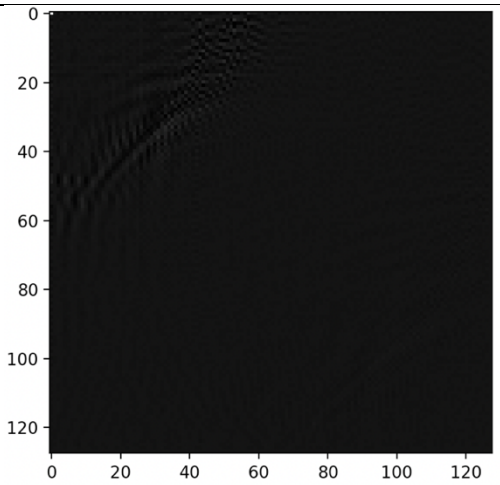


Figure 2.2 DCT transform

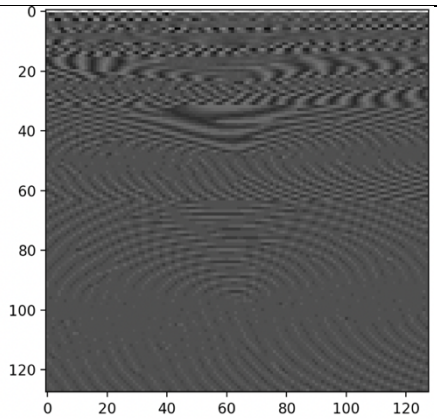


Figure 2.3 Haar Transform

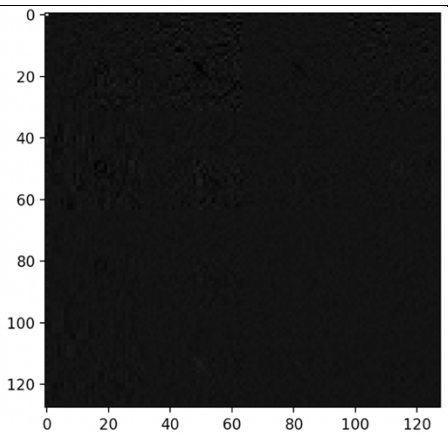


Figure 2.4 Walsh Transform

Example 3: Natural Image

Figure 3.1 Original Image

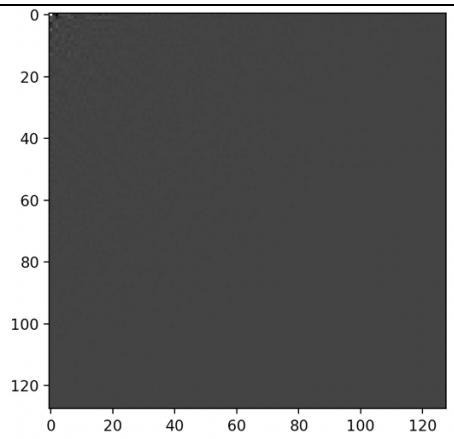


Figure 3.2 DCT Transform

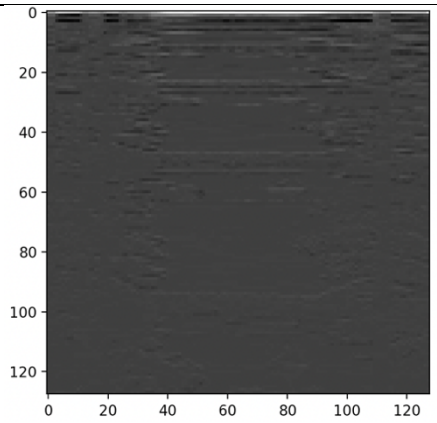


Figure 3.3 Haar Transform

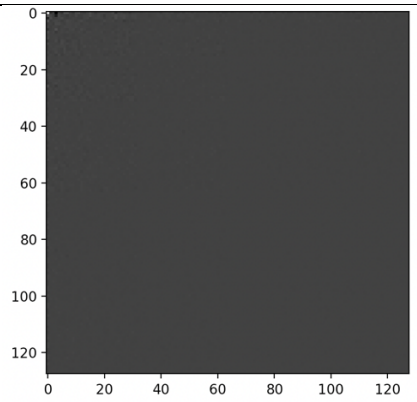


Figure 3.4 Walsh Transform

Example 4: Synthetic Image

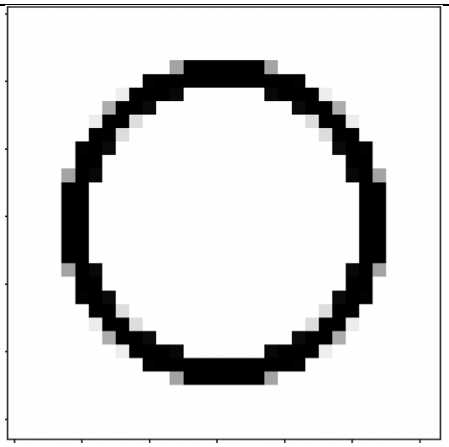


Figure 4.1 Original Image; resized to 32x32

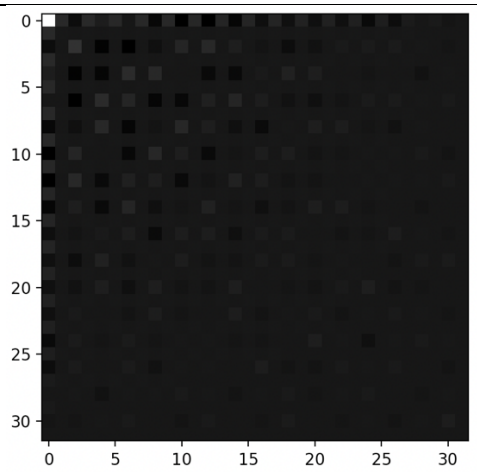


Figure 4.2 DCT Transform

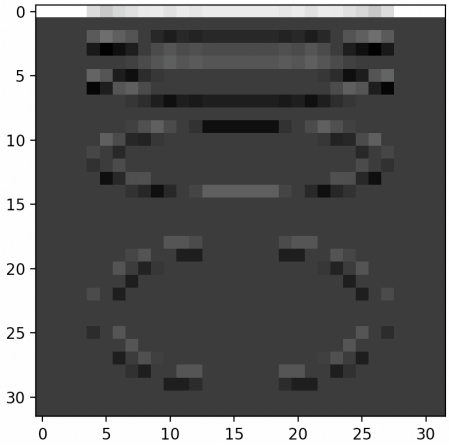


Figure 4.3 Haar Transform

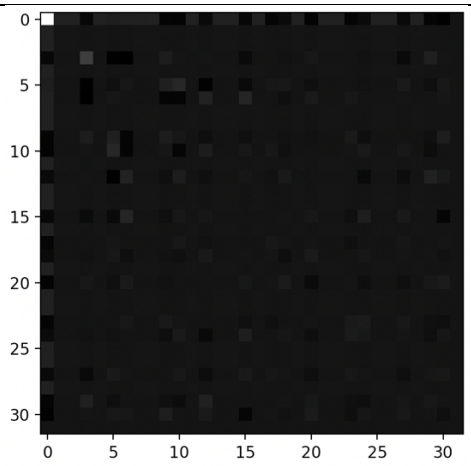


Figure 4.4 Walsh Transform

Discussion

The kernels of each transform, demonstrate a generic property of how the most significant pixels are stored. Specifically, we can see that DCT is the only sinusoidal transform which reflects in smoother transition compared to Haar and Walsh. Nonetheless, for 32x32 kernel, we can observe more pixelized representation of the DCT. The Haar and Walsh kernel appear as a representation of squares. Specifically, Walsh kernel is strictly binomial with values of the kernel being either “0” or “1”, while Haar’s has a different shades of grey since the values vary between “0” and “1”.

From the experiments, we observe that Walsh and DCT transform have a similar structure of storing the important data, as the informative pixels are spread out across the image array without any specific pattern with a most dense area being in the top left corner. On the other hand, Haar transform stores the data in the circle-like symmetric manner with respect to x-axis.

Important to notice that computational time for Walsh and DCT transform increases significantly for the image of larger size when compared to Haar transform.

Program

DCT transform Code

```
from PIL import Image
import numpy as np
import math, cv2
from matplotlib import pyplot as plt
from timeit import default_timer as timer


def dct_2d(img):
        M, N = np.shape(img)
        dct_result = np.zeros((M, N))
        pi = math.pi
        time_start = timer()

        for u in range(M-1):
            for v in range(N-1):
                if u == 0:
                    alpha_u = math.sqrt(1/M)
                else:
                    alpha_u = math.sqrt(2/M)
                if v == 0:
                    alpha_v = math.sqrt(1/M)
                else:
                    alpha_v = math.sqrt(2/M)

                sum = 0
                for x in range(M-1):
                    for y in range(N-1):
                        cos_x = math.cos((2*x+1)*u*pi/(2*M))
                        cos_y = math.cos((2*y+1)*v*pi/(2*N))

                        temp_sum = img[x,y]*cos_x*cos_y

                        sum += temp_sum
```

```
                dct_result[u,v] = alpha_u* alpha_v * sum

        time_end = timer()
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        return dct_result

def get_kernel(N):
    pi = math.pi
    dct = np.zeros((N, N))
    for x in range(N):
        dct[0,x] = math.sqrt(2.0/N) / math.sqrt(2.0)
    for u in range(1,N):
        for x in range(N):
            dct[u,x] = math.sqrt(2.0/N) * math.cos((pi/N) * u * (x + 0.5) )

    return dct


if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/shape_circle.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (32,32))

    plt.imshow(image, cmap='gray')
    plt.show()
    kernel = get_kernel(32)
    plt.imshow(kernel, cmap='gray')
    plt.show()
    transform = dct_2d(image)

    plt.imshow(transform, cmap='gray')
    plt.show()
```

Haar transform Code

```
from PIL import Image
import numpy as np
import math, cv2
from matplotlib import pyplot as plt
from timeit import default_timer as timer

def get_kernel(N):
    haar = np.zeros((N, N))
    x_values = []
```

```
    for i in range(0, N):
        x = (i)/N
        x_values.append(x)

    min_r = 0
    max_r = int(math.log2(N))
    rm_tuples = []

    for r in range(min_r, max_r):
        min_m = 1
        max_m = 2**r
        for m in range(min_m, max_m+1):
            rm_tuples.append((r,m))

    for u in range(N):
        r,m = rm_tuples[u-1]
        min_1_included = (m-1)/(2**r)
        max_1_not_included = (m-0.5)/(2**r)
        min_2_included = (m-0.5)/(2**r)
        max_2_not_included = (m)/(2**r)
        for v in range(N):
            if u == 0:
                haar_value = 1/math.sqrt(N)
            else:
                x = x_values[v]
                if x>= min_1_included and x< max_1_not_included:
                    haar_value = (2**(r/2))/(math.sqrt(N))
                elif x>= min_2_included and x< max_2_not_included:
                    haar_value = -(2**(r/2))/(math.sqrt(N))
                else:
                    haar_value = 0
            haar[u,v] = haar_value*(math.sqrt(N))

    return haar

def fix_binary_to_lenght(binary, lenght):
    while len(binary) < lenght:
        binary = '0'+binary
    return binary


def ordered_kernel(haar_transform):
    skips_number = []
    for row in haar_transform:
        current_value = row[0]
        skips = 0
        for i in row:
```

```
            temp_value = i
            if temp_value != current_value:
                skips += 1
            current_value = temp_value
        skips_number.append(skips)
    ordered_transform = np.zeros((haar_transform.shape))
    for original_position, ordered_position in enumerate(skips_number):
        ordered_transform[ordered_position] = haar_transform[original_position]

    return ordered_transform

def multiply_matrix(A,B):
  global C
  if  A.shape[1] == B.shape[0]:
    C = np.zeros((A.shape[0],B.shape[1]),dtype = int)
    for row in range(A.shape[0]):
        for col in range(B.shape[1]):
            for elt in range(len(B)):
              C[row, col] += A[row, elt] * B[elt, col]
    return C
  else:
    return "Sorry, cannot multiply A and B."

def decimalToBinary(n):
    return bin(n).replace("0b", "")

if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natural_scene.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (128,128))

    plt.imshow(image, cmap='gray')
    plt.show()
    haar = get_kernel(128)
    plt.imshow(haar, cmap='gray')
    plt.show()
    new_image = multiply_matrix(haar,image)
    plt.imshow(new_image, cmap='gray')
    plt.show()
    reverse = multiply_matrix(np.linalg.inv(haar),new_image)
    plt.imshow(reverse, cmap='gray')
    plt.show()
```

Walsh transform Code

```
from PIL import Image
import numpy as np
import math, cv2
from matplotlib import pyplot as plt
from timeit import default_timer as timer


def walsh_transform(img):

        M, N = np.shape(img)

        walsh_results = np.zeros((M, N))

        number_of_bits = int(math.log2(N))

        time_start = timer()

        for u in range(M-1):
            for v in range(N-1):
                u_bits = decimalToBinary(u)
                v_bits = decimalToBinary(v)
                u_final = fix_binary_to_lenght(u_bits, number_of_bits)
                v_final = fix_binary_to_lenght(v_bits, number_of_bits)
                sum = 0
                for x in range(M-1):
                    for y in range(N-1):
                        f_x_y = img[x,y]
                        x_bits = decimalToBinary(x)
                        y_bits = decimalToBinary(y)
                        x_final = fix_binary_to_lenght(x_bits, number_of_bits)
                        y_final = fix_binary_to_lenght(y_bits, number_of_bits)

                        temp_sum = 0
                        product = 1
                        for i in range(number_of_bits):

                            power = int(x_final[i])*int(u_final[number_of_bits-
1-i]) + int(y_final[i])*int(v_final[number_of_bits-1-i])

                            product = product*((-1)**power)

                        temp_sum = (1/N)*(f_x_y*product)
                        sum += temp_sum
                walsh_results[u,v] = sum

        time_end = timer()
        time_elapsed = time_end - time_start
```

```
        print(f"Total execution time is: {time_elapsed}")
        return walsh_results

def get_kernel(N):
    walsh = np.zeros((N, N))
    number_of_bits = int(math.log2(N))
    count_x = 0
    for x in range(N):
        count_x += 1
        count_y = 0
        for u in range(N):
            count_y += 1
            x_bits = decimalToBinary(x)
            u_bits = decimalToBinary(u)
            x_final = fix_binary_to_lenght(x_bits, number_of_bits)
            u_final = fix_binary_to_lenght(u_bits, number_of_bits)

            product = 1
            for i in range(number_of_bits):
                power = int(x_final[i])*int(u_final[number_of_bits-1-i])
                product = product*((-1)**power)

            walsh[x,u] = (1/N)*product
    return walsh

def fix_binary_to_lenght(binary, lenght):
    while len(binary) < lenght:
        binary = '0'+binary
    return binary

def decimalToBinary(n):
    return bin(n).replace("0b", "")

if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/shape_circle.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (32,32))

    plt.imshow(image, cmap='gray')
    plt.show()
    original_kernel = get_kernel(32)
    plt.imshow(original_kernel, cmap='gray')
    plt.show()
    transform = walsh_transform(image)
    plt.imshow(transform, cmap='gray')
    plt.show()
```

EEL 5820: Digital Image Processing
Fall 2022

Homework # 5

Fast Fourier Transform

Submitted by:

Your name: Maryna Veksler

PID#: 5848285

Department of Electrical and Computer Engineering

Date: _____12/11/2022_____

Objective

To Implement the centered Fast Fourier Transform (FFT) and compare its computational time to DFT.

Method

The FFT image transform can be implemented as follows

$$F(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)e^{-j2\pi(x\frac{m}{M}+y\frac{n}{N})}$$

While implementing the FFT, we use the approach called butterfly effect, meaning we only need to compute half of the point at every step of the transformation in case of 1-D transform.

While FFT is a continuation of DFT, when applied to the image we can significantly decrease the processing time as only ¼ of all points needs to be computed at every step.

We also implement a centered FFT

Results

Example 1:
16x16: Total execution time is: 0.005862332999999831
32x32: Total execution time is: 0.023367749999998466
64x64: Total execution time is: 0.08349770799999945
128x128: Total execution time is: 0.2945040839999997
256x256: Total execution time is: 1.1517751670000003

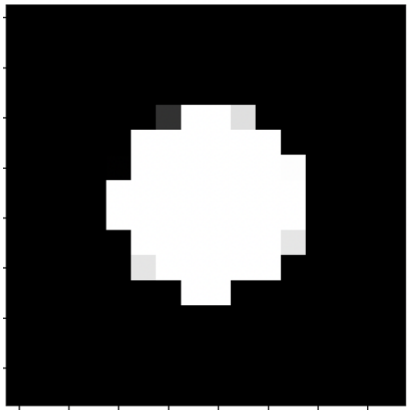


Figure 1.1 Original Image (16x16)

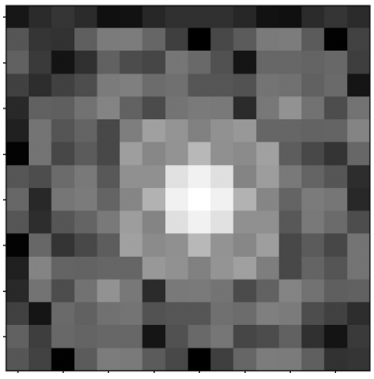


Figure 1.2 FFT

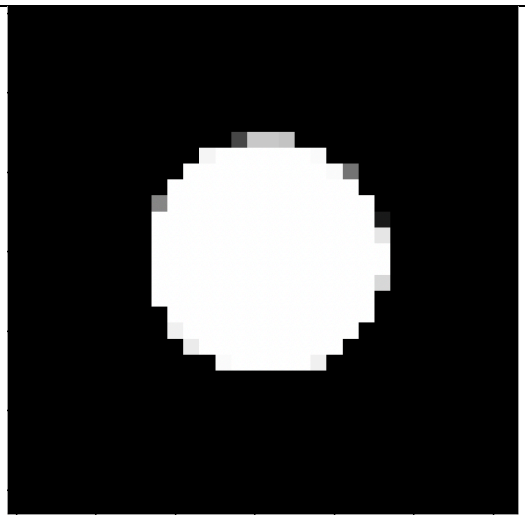


Figure 1.3 Original Image (32x32)

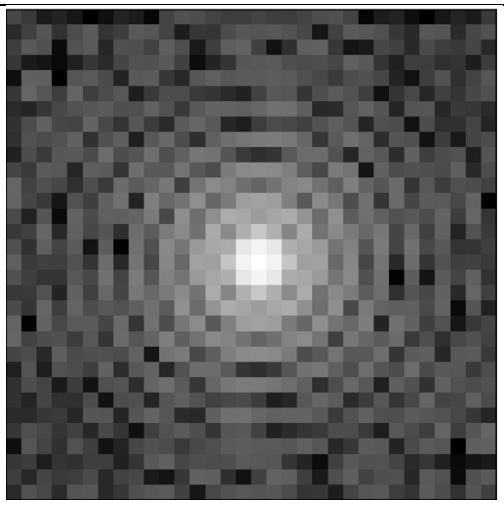


Figure 1.4 FFT

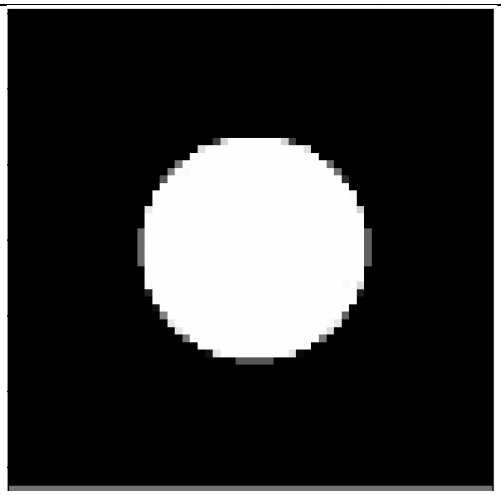


Figure 1.5 Original Image (64x64)

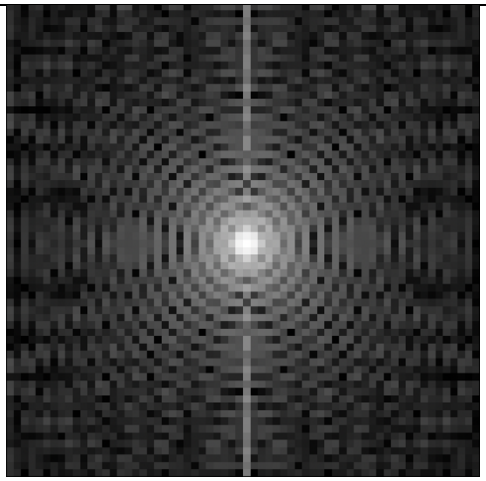


Figure 1.6 FFT

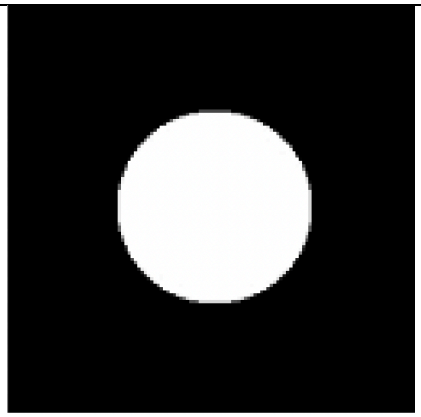


Figure 1.7 Original Image (128x128)

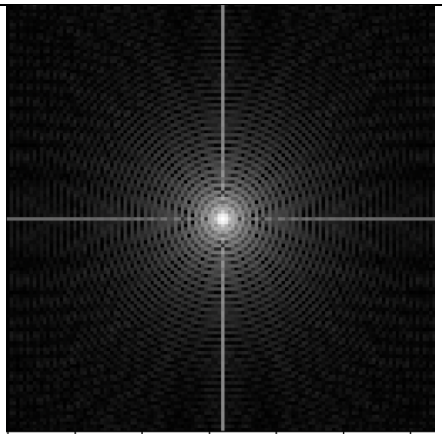


Figure 1.8 FFT

Figure 1.9 Original Image (256x256)

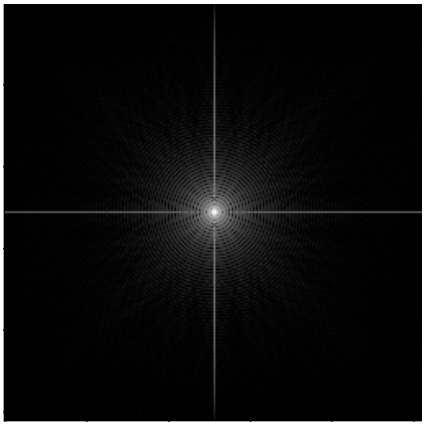


Figure 1.10 FFT

Example 2:

16x16: Total execution time is: 0.006092375000001482
32x32: Total execution time is: 0.02329754200000167
64x64: Total execution time is: 0.0830590839999985
128x128: Total execution time is: 0.29586679200000177
256x256: Total execution time is: 1.1622672919999992

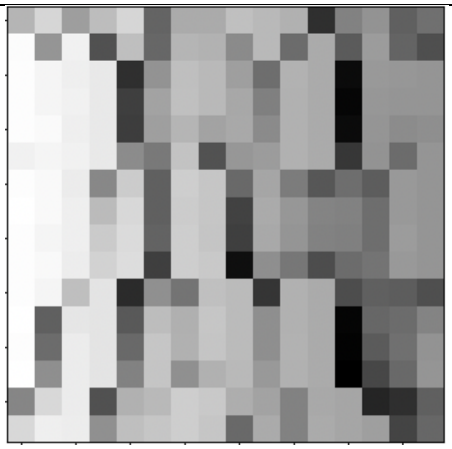


Figure 2.1 Original Image (16x16)

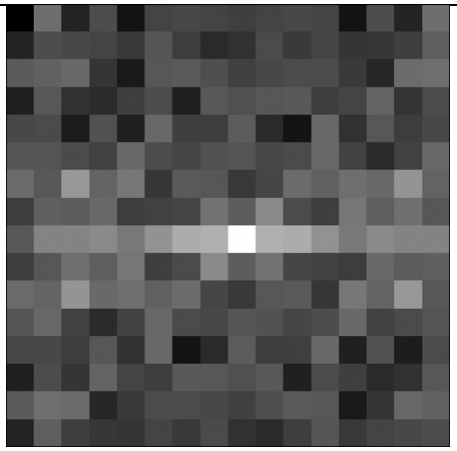


Figure 2.2 FFT

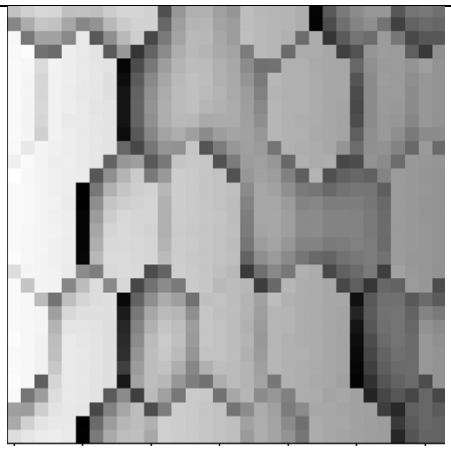


Figure 2.3 Original Image (32x32)

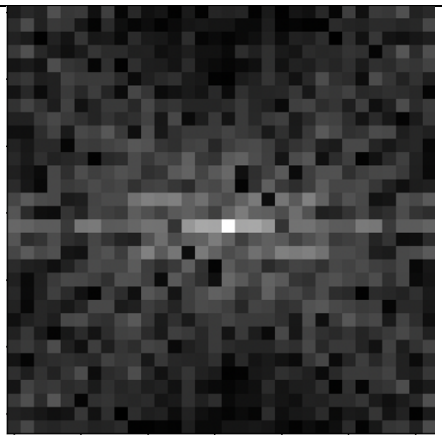


Figure 2.4 FFT

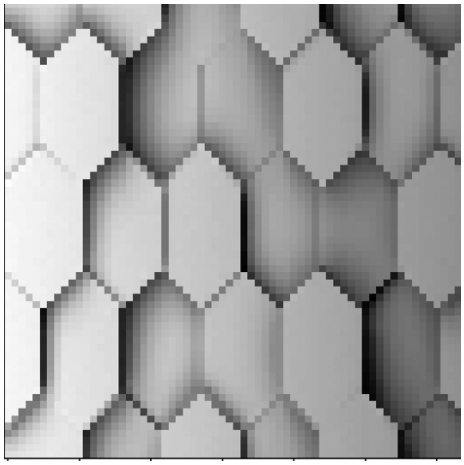


Figure 2.5 Original Image (64x64)

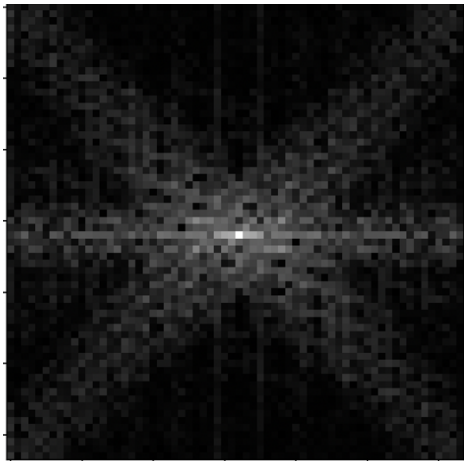


Figure 2.6 FFT

Figure 2.7 Original Image (128x128)

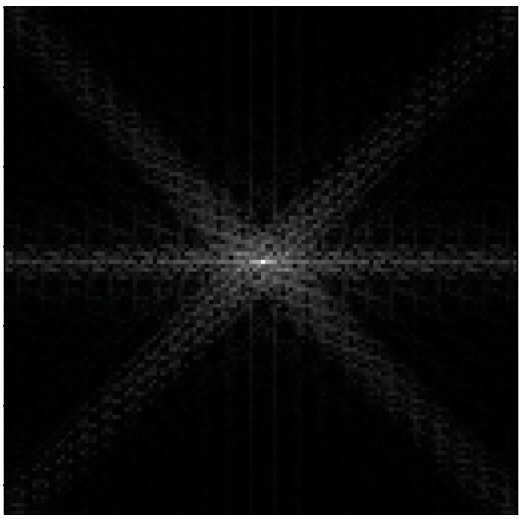


Figure 2.8 FFT

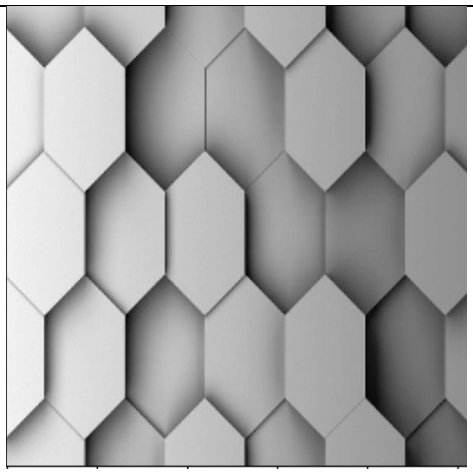


Figure 2.9 Original Image (256x256)

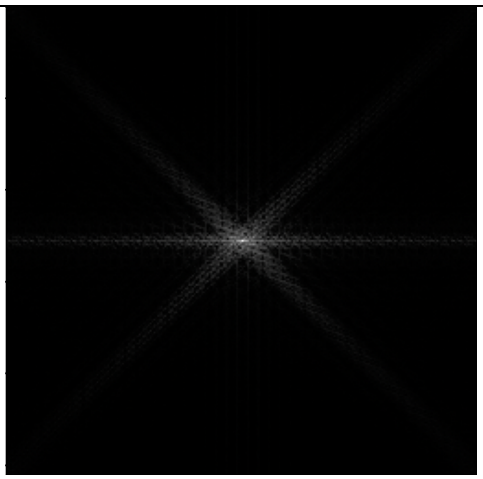


Figure 2.10 FFT

Example 3:
16x16: Total execution time is: 0.006027916999997274
32x32: Total execution time is: 0.023570499999998162
64x64: Total execution time is: 0.08411412500000282
128x128: Total execution time is: 0.29117312500000025
256x256: Total execution time is: 1.1457247499999994

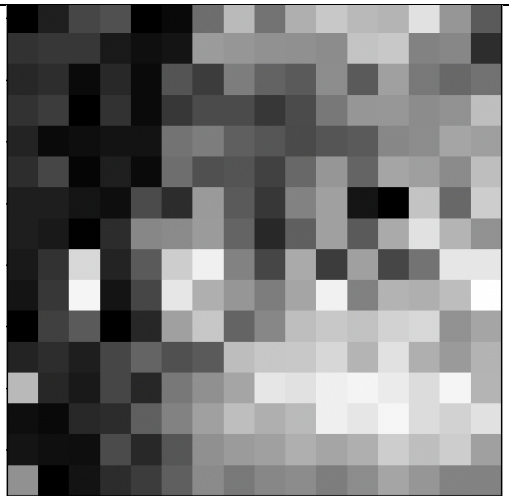


Figure 3.1 Original Image (16x16)

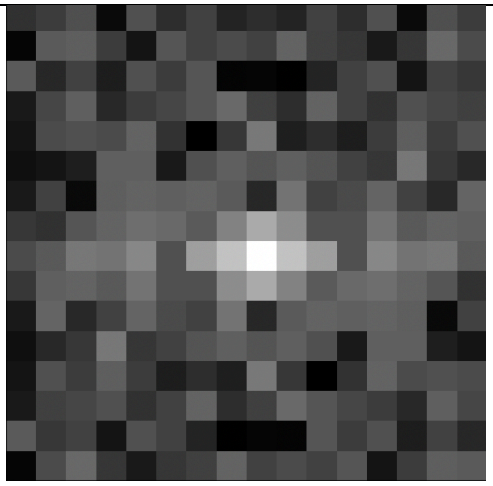


Figure 3.2 FFT

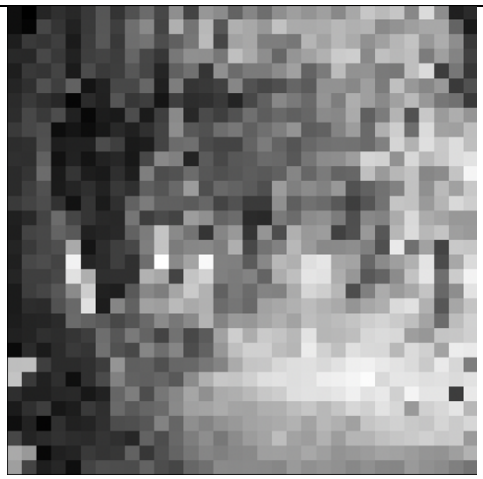


Figure 3.3 Original Image (32x32)

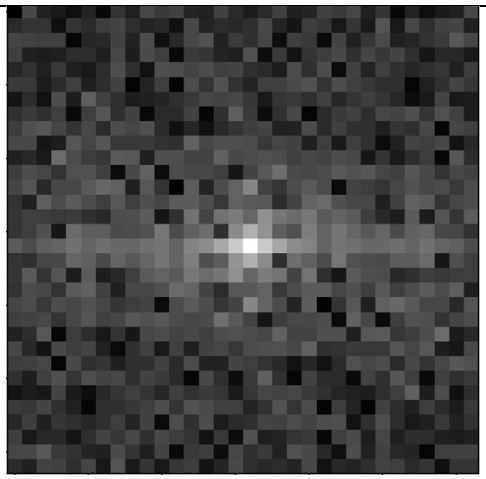


Figure 3.4 FFT

Figure 3.5 Original Image (64x64)

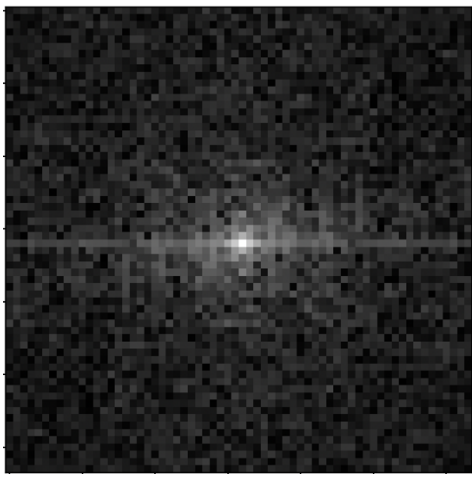


Figure 3.6 FFT

Figure 3.7 Original Image (128x128)

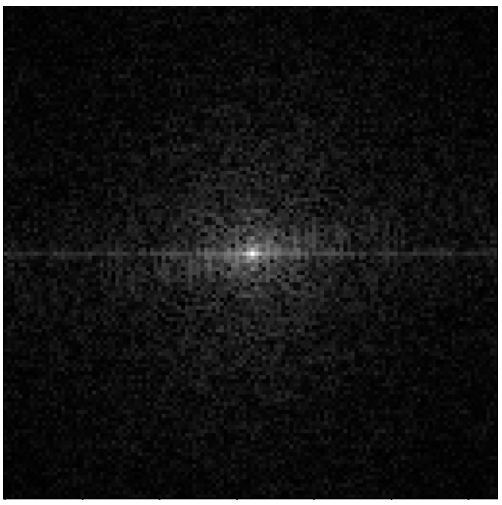


Figure 3.8 FFT

Figure 3.9 Original Image (256x256)

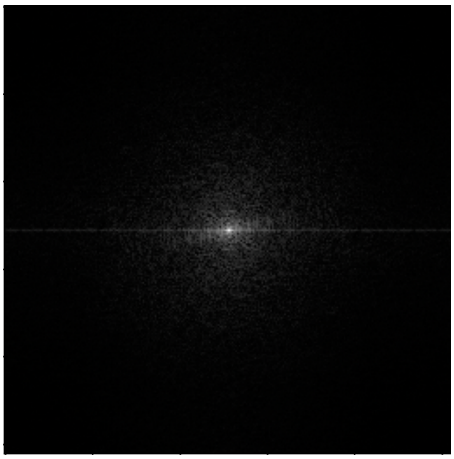


Figure 3.10 FFT

Discussion

In this home work, we applied FFT to 3 different images of different sizes (e.g., 16x16, 32x32, 64x64, 128x128, and 256x256. We can make following conclusions based on the experiments:

1. As predicted, using FFT allows to significantly decrease the processing time compared to DFT. Moreover, as the size increase the time increase insignificantly.
2. For the images of the reduced size, the FFT transform appears pixelized. While we can observe the center value clearly, the rest of the transform values containing data are spread out.
3. As images increase in size and we can clearly see the image data, the centered FFT becomes more visible. Specifically, we can see the majority of the information is contained in the center with a light lines dispersing from the center to the edges.
4. The FFT transform has an interesting behavior based on the original shape. For example, for circular shape (Example 1) the FFT appear as a very bright dot with surrounding aura, and two perpendicular lines crossing in the middle.

   For the structure containing hexagons, the FFT appears as a dot with the aura consisting of 6 lines that intersect in the middle. If we connect the edges of thos lines, we can observe the hexagon shape.

Finally, for the natural image, the informative pixels are spread out circularly around the middle bright dot, and there is a parallel horizontal line crossing the middle.

Program

```
from PIL import Image
import numpy as np
import math, cv2
from matplotlib import pyplot as plt
from timeit import default_timer as timer


def compute_single_2d_FFT(image, u, v, N):
        result = 0 + 0j
        for x in range(N):
            for y in range(N):
                result += (image[x, y] * (math.cos((2*math.pi*(u*x + v*y))/N) -
                                          (1j*math.sin((2*math.pi*(u*x +
v*y))/N))))
        return result

def compute_forward_DFT_no_separation(imge):
        N = imge.shape[0]
        final2DDFT = np.zeros([N, N], dtype=np.complex128)
        for u in range(N):
            for v in range(N):
                final2DDFT[u, v] = compute_single_2d_FFT(imge, u, v, N)
        return ((1.0/(N**2))*final2DDFT)


def compute_single_w(num, denom):
        return math.cos((2*math.pi*num)/denom) -
(1j*math.sin((2*math.pi*num)/denom))

def compute_centered_image(imge):
        M, N = imge.shape
        newImge = np.zeros([M, N], dtype=int)
        for x in range(M):
            for y in range(N):
                newImge[x, y] = imge[x, y] * ((-1)**(x+y))

        return newImge

def compute_w(val, denom, oneD=True):
        val = int(val)
```

```
        if oneD:
            result = np.zeros([val, 1], dtype=np.complex128)
            for i in range(val):
                result[i] = compute_single_w(i, denom)
        else:
            result = np.zeros([val, val], dtype=np.complex128)
            for i in range(val):
                for j in range(val):
                    result[i, j] = compute_single_w((i+j), denom)
        return result

def fft(imge):
        #Compute size of the given image
        N = imge.shape[0]

        #Compute the FFT for the base case (which uses the normal DFT)
        if N == 2:
            return compute_forward_DFT_no_separation(imge)

        #Divide the original image into even and odd
        imgeEE = np.array([[imge[i,j] for i in range(0, N, 2)] for j in
range(0, N, 2)]).T
        imgeEO = np.array([[imge[i,j] for i in range(0, N, 2)] for j in
range(1, N, 2)]).T
        imgeOE = np.array([[imge[i,j] for i in range(1, N, 2)] for j in
range(0, N, 2)]).T
        imgeOO = np.array([[imge[i,j] for i in range(1, N, 2)] for j in
range(1, N, 2)]).T

        #Compute FFT for each of the above divided images
        FeeUV = fft(imgeEE)
        FeoUV = fft(imgeEO)
        FoeUV = fft(imgeOE)
        FooUV = fft(imgeOO)

        #Compute also Ws
        Wu = compute_w(N/2, N)
        Wv = Wu.T #Transpose
        Wuv = compute_w(N/2, N, oneD=False)

        #Compute F(u,v) for u,v = 0,1,2,...,N/2
        imgeFuv = 0.25*(FeeUV + (FeoUV * Wv) + (FoeUV * Wu) + (FooUV * Wuv))

        #Compute F(u, v+M) where M = N/2
        imgeFuMv = 0.25*(FeeUV + (FeoUV * Wv) - (FoeUV * Wu) - (FooUV * Wuv))

        #Compute F(u+M, v) where M = N/2
        imgeFuvM = 0.25*(FeeUV - (FeoUV * Wv) + (FoeUV * Wu) - (FooUV * Wuv))
```

```
        #Compute F(u+M, v+M) where M = N/2
        imgeFuMvM = 0.25*(FeeUV - (FeoUV * Wv) - (FoeUV * Wu) + (FooUV * Wuv))

        imgeF1 = np.hstack((imgeFuv, imgeFuvM))
        imgeF2 = np.hstack((imgeFuMv, imgeFuMvM))
        imgeFFT = np.vstack((imgeF1, imgeF2))

        return imgeFFT

def normalize_by_log(dftImge):
        dftFourierSpect = compute_spectrum(dftImge)

        dftNormFourierSpect = (255.0/ math.log10(255)) * np.log10(1 +
(255.0/(np.max(dftFourierSpect))*dftFourierSpect))

        return dftNormFourierSpect


def compute_spectrum(dftImge):
        N = dftImge.shape[0]

        fourierSpect = np.zeros([N, N], dtype=float)
        for i in range(N):
            for j in range(N):
                v = dftImge[i, j]
                fourierSpect[i, j] = math.sqrt((v.real)**2 + (v.imag)**2)
        return fourierSpect

if __name__ == "__main__":

        image_path =
"/Users/marynavek/Projects/ImageProcessing/natural_im_2.jpg"

        image = cv2.imread(image_path, 0)
        image = cv2.resize(image, (256,256))

        plt.imshow(image, cmap='gray')
        plt.show()
        centeredImge = compute_centered_image(image)
        time_start = timer()
        fft = fft(centeredImge)
        time_end = timer()
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        fftCenteredNormImge = normalize_by_log(fft)
        plt.imshow(fftCenteredNormImge, cmap='gray')
        plt.show()
```

EEL 5820: Digital Image Processing
Fall 2022

Homework # 6

Filtering

Submitted by:

Your name: Maryna Veksler

PID#: 5848285

Department of Electrical and Computer Engineering

Date: _____12/11/2022_____

Objective

To Implement Ideal filter and Butterworth filter using different orders.

Method

In this work, we focus on two types of filters, Ideal and Butterworth. Moreover, we examine both cases for each filter, low pass and high pass.

Ideal Low Pass Filter can be calculated as follows

$$|H(\omega)| = \begin{cases} 1 & \text{for } |\omega| < \omega_c \\ 2 & \text{for } |\omega| > \omega_c \end{cases}$$

And the High Pass as

$$|H(\omega)| = \begin{cases} 0 & \text{for } |\omega| < \omega_c \\ 1 & \text{for } |\omega| > \omega_c \end{cases}$$

In both cases, the phase is calculated as $[\theta(\omega) = -\omega td]$.

Butterworth Low Pass filter calculated as

$$H(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}}$$

And the High Pass as

$$H(u, v) = \frac{1}{1 + [D_0/D(u, v)]^{2n}}$$

In order to successfully apply filter operations, the image should be first converted to the frequency domain before the reconstruction. Thus, for all of the experiments we follow the steps as

1. Apply FFT to the image
2. Apply Filter and see the results
3. Compute inverse after the filter is applied to reconstruct the image

Results

Example 1: Low Pass Filter

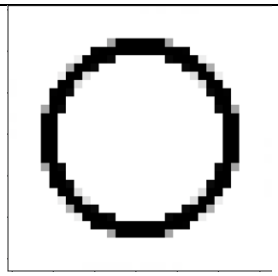


Figure 1.1 Original Image

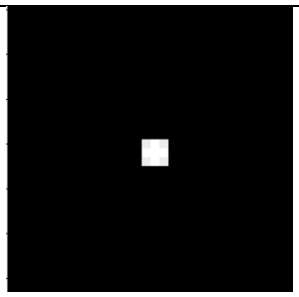


Figure 1.2 Butterworth filtered Image for D=1.5

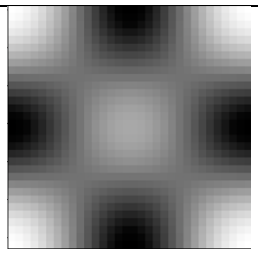


Figure 1.3 Reconstructed Image

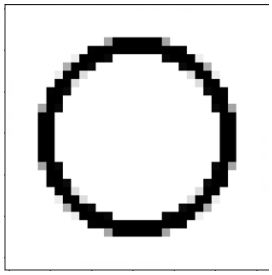


Figure 1.4 Original Image

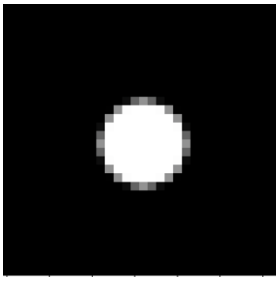


Figure 1.5 Butterworth filtered Image for D=5

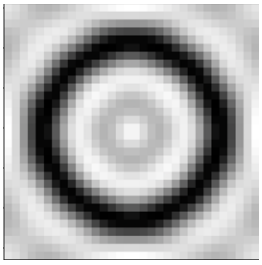


Figure 1.6 Reconstructed Image

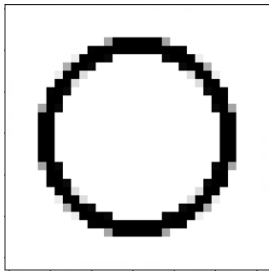


Figure 1.7 Original Image

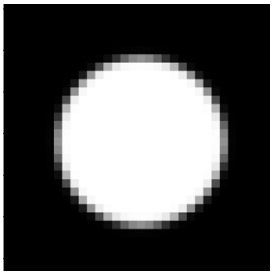


Figure 1.8 Butterworth filtered Image for D=10

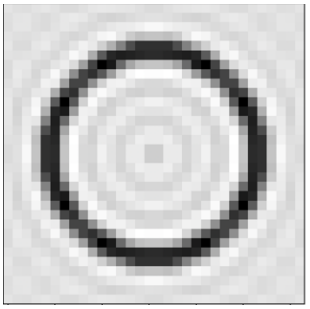


Figure 1.9 Reconstructed Image

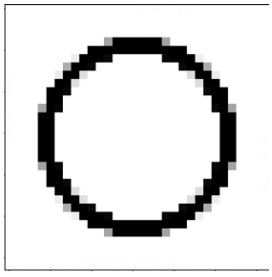


Figure 1.10 Original Image

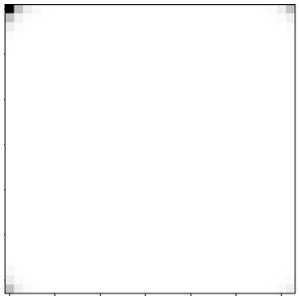


Figure 1.11 Butterworth filtered Image for D=25

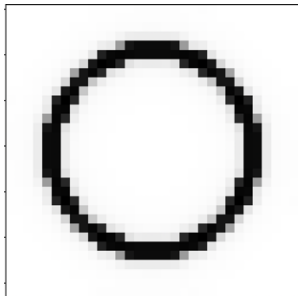
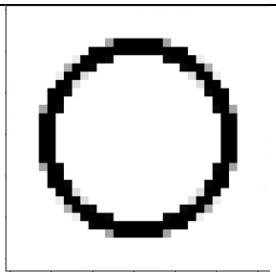
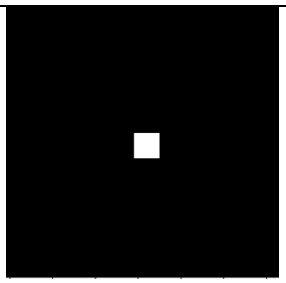
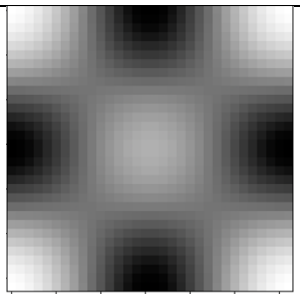
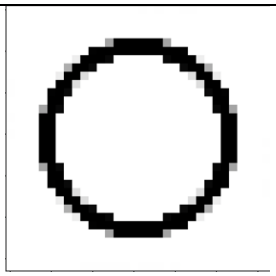
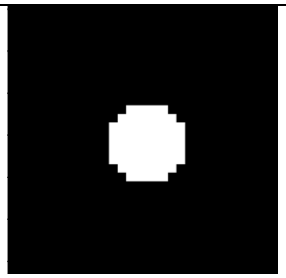
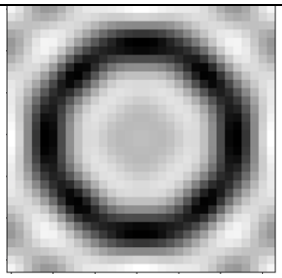
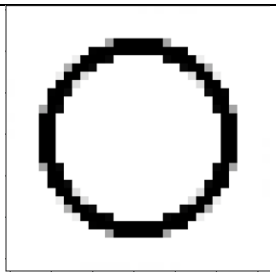
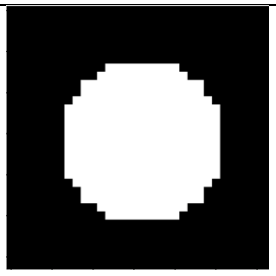
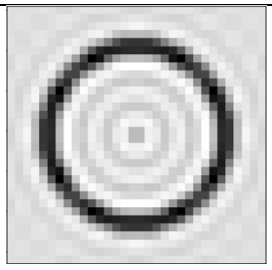
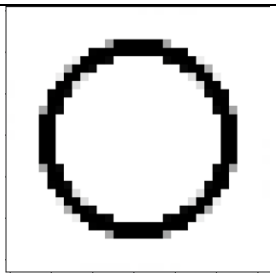
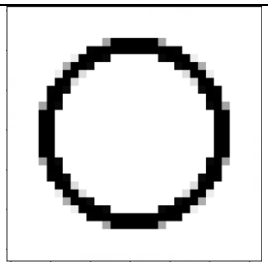
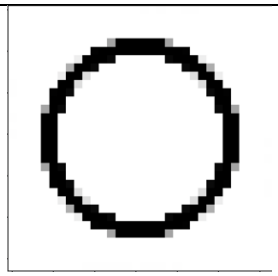
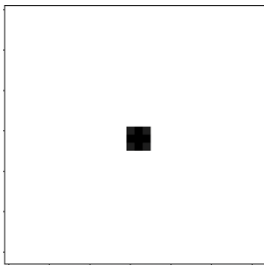
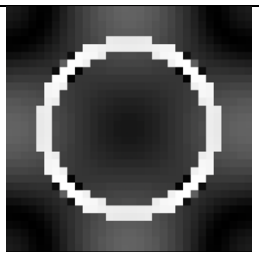
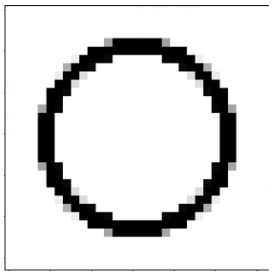
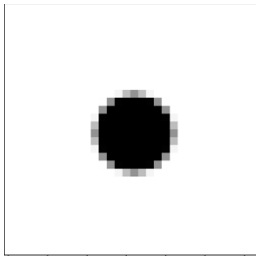
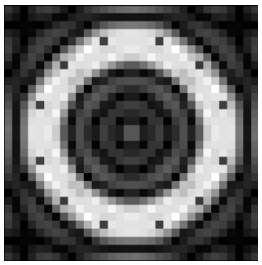
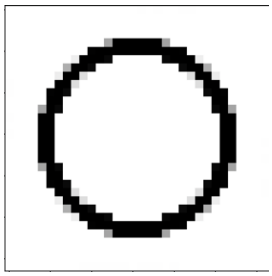
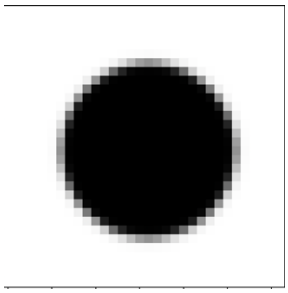
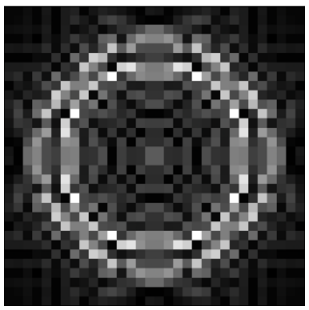
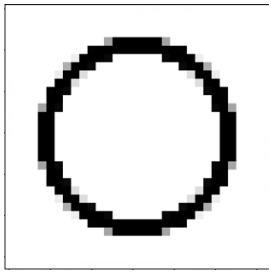
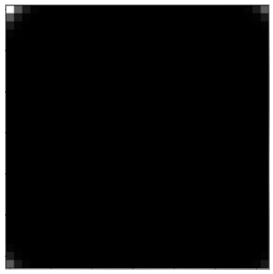
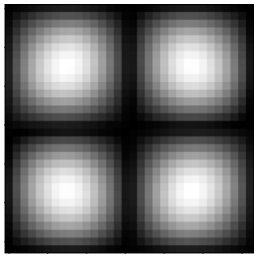
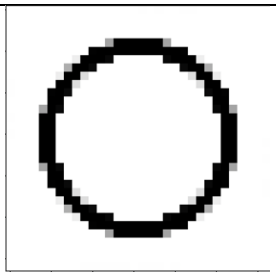
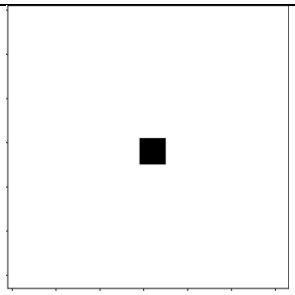
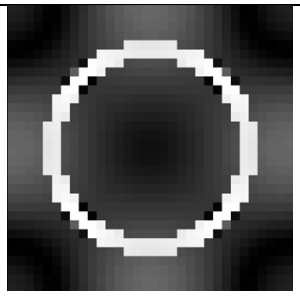
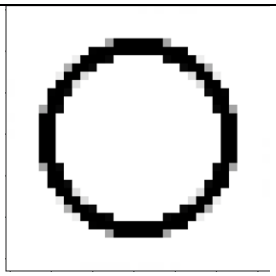
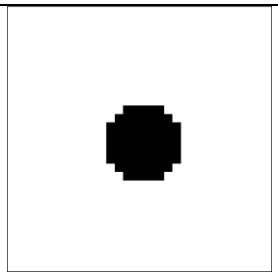
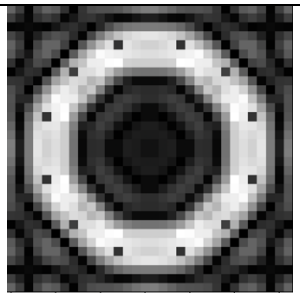
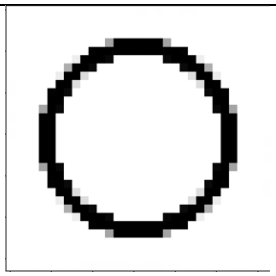
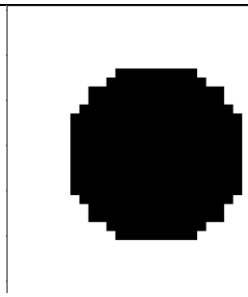
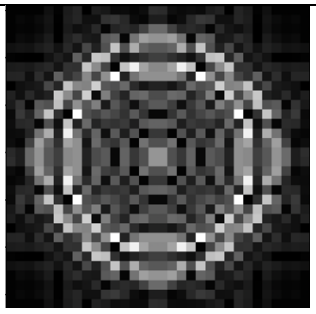
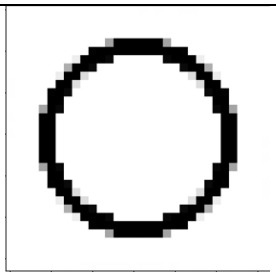


Figure 1.12 Reconstructed Image

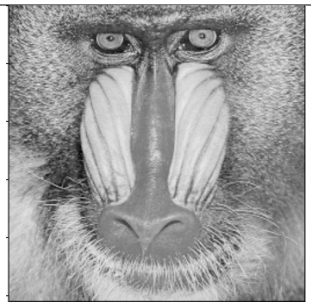
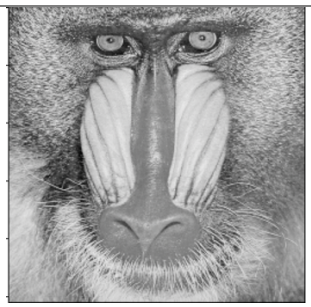
 **Figure 1.13 Original Image**	 **Figure 1.14 Ideal filtered Image for D=1.5**	 **Figure 1.15 Reconstructed Image**
 **Figure 1.16 Original Image**	 **Figure 1.17 Ideal filtered Image for D=5**	 **Figure 1.18 Reconstructed Image**
 **Figure 1.19 Original Image**	 **Figure 1.20 Ideal filtered Image for D=10**	 **Figure 1.21 Reconstructed Image**
 **Figure 1.22 Original Image**	 **Figure 1.23 Ideal filtered Image for D=25**	 **Figure 1.24 Reconstructed Image**

Example 2: High Pass Filter

 **Figure 2.1 Original Image**	 **Figure 2.2 Butterworth filtered Image for D=1.5**	 **Figure 2.3 Reconstructed Image**
 **Figure 2.4 Original Image**	 **Figure 2.5 Butterworth filtered Image for D=5**	 **Figure 2.6 Reconstructed Image**
 **Figure 2.7 Original Image**	 **Figure 2.8 Butterworth filtered Image for D=10**	 **Figure 2.9 Reconstructed Image**
 **Figure 2.10 Original Image**	 **Figure 2.11 Butterworth filtered Image for D=25**	 **Figure 2.12 Reconstructed Image**

 **Figure 2.13 Original Image**	 **Figure 2.14 Ideal filtered Image for D=1.5**	 **Figure 2.15 Reconstructed Image**
 **Figure 2.16 Original Image**	 **Figure 2.17 Ideal filtered Image for D=5**	 **Figure 2.18 Reconstructed Image**
 **Figure 2.19 Original Image**	 **Figure 2.20 Ideal filtered Image for D=10**	 **Figure 2.21 Reconstructed Image**
 **Figure 2.22 Original Image**	 **Figure 2.23 Ideal filtered Image for D=25**	 **Figure 2.24 Reconstructed Image**

Example 3:

 **Figure 3.1 Original Image**	 **Figure 3.2 Butterworth HP (D=5)**	 **Figure 3.3 Butterworth LP (D=25)**
 **Figure 3.1 Original Image**	 **Figure 3.4 Ideal HP (D=5)**	 **Figure 3.3 Ideal LP (D=30)**
 **Figure 3.1 Original Image**	 **Figure 3.2 Butterworth HP (D=5)**	 **Figure 3.3 Butterworth LP (D=15)**
 **Figure 3.1 Original Image**	 **Figure 3.4 Ideal HP (D=15)**	 **Figure 3.3 Ideal LP (D=15)**

Discussion

From the experiments, we can observe that Low-Pass filter for both Ideal and Butterworth filters provides very similar results. However, the LP ideal filter in FFT domain appears with sharper edges.

Similarly the high-pass (HP) approach yields similar results for both Ideal and Butterworth filters.

One interesting thing to notice, if we set higher D (order), both filters behave very differently and their behavior is unpredictable.

For the LP filtering, as D increases the filtered image appears closer to the original, while for lower D the reconstructed image appears blurry.

For the HP filtering, as D increases the filter may behave unpredictably and the reconstructed image is further for the original, while lower D produces reconstructed images of good quality

Additionally, we do not notice any significant differences in the computational time. In the future, we recommend to carefully select the D (order) value for all filtering operations, as some values may causes misbehavior and yield unexpected results.

Program

```
import math
import numpy as np
from ideal_filter import IdealFilter
from butterworth_filter import ButterworthFilter
from PIL import Image
import numpy as np
import math, cv2
from matplotlib import pyplot as plt



class ButterworthFilter():

    def __init__(self):
        pass

    def __distance(self, point1,point2):
        return math.sqrt((point1[0]-point2[0])**2 + (point1[1]-point2[1])**2)

    def FPLP(self, D0,imgShape,n):
        base = np.zeros(imgShape[:2])
        rows, cols = imgShape[:2]
```

```
        center = (rows/2,cols/2)
        for x in range(cols):
            for y in range(rows):
                base[y,x] = 1/(1+(self.__distance((y,x),center)/D0)**(2*n))
        return base

    def FPHP(self, D0,imgShape,n):
        base = np.zeros(imgShape[:2])
        rows, cols = imgShape[:2]
        center = (rows/2,cols/2)
        for x in range(cols):
            for y in range(rows):
                base[y,x] = 1-1/(1+(self.__distance((y,x),center)/D0)**(2*n))
        return base

class IdealFilter:

    def __init__(self):
        pass

    def __distance(self, point1,point2):
        return math.sqrt((point1[0]-point2[0])**2 + (point1[1]-point2[1])**2)

    def low_pass_filter(self, D0,imgShape):
        base = np.zeros(imgShape[:2])
        rows, cols = imgShape[:2]
        center = (rows/2,cols/2)
        for x in range(cols):
            for y in range(rows):
                if self.__distance((y,x),center) < D0:
                    base[y,x] = 1
        return base

    def high_pass_filter(self, D0,imgShape):
        base = np.ones(imgShape[:2])
        rows, cols = imgShape[:2]
        center = (rows/2,cols/2)
        for x in range(cols):
            for y in range(rows):
                if self.__distance((y,x),center) < D0:
                    base[y,x] = 0
        return base


if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natual_im_1.jpg"
```

```
    image = cv2.imread(image_path, 0)
    plt.imshow(image, cmap='gray')
    plt.show()
    imgTrans = np.fft.fftshift(np.fft.fft2(image))
    filter = ButterworthFilter()
    imgLowPass = filter.FPLP(1.5,image.shape)
    imgProcessLP = imgTrans*imgLowPass

    plt.imshow(imgLowPass, cmap='gray')
    plt.show()

    inversaLP = np.fft.ifftshift(imgProcessLP)
    inversaLP = np.fft.ifft2(inversaLP)
    inversaLP = np.abs(inversaLP)

    plt.imshow(image, cmap='gray')
    plt.show()
    plt.imshow(inversaLP, cmap='gray')
    plt.show()

    imgHighPass = filter.FPHP(10,image.shape, 20)
    imgProcessHighPass = imgTrans*imgHighPass
    inversaHighPass= np.fft.ifftshift(imgProcessHighPass)
    inversaHighPass = np.fft.ifft2(inversaHighPass)
    inversaHighPass = np.abs(inversaHighPass)
```

EEL 5820: Digital Image Processing
Fall 2022

Homework # 7

Analyzing the difference between original and reconstructed images

Submitted by:

Your name: Maryna Veksler

PID#: 5848285

Department of Electrical and Computer Engineering

Date: _____12/12/2022_____

Objective

To analyze the ability of different transforms to pack the most significant information.

Method

To determine the entropy qualities of different transforms, we will first apply the transform directly to the image and then apply the reverse transform.
We will use mean-square error to determine the amount of the same or varying coefficients

Results

Example 1:
MSE FFT: 942047345375.3398
MSE DCT: 32481.846119283746
MSE Walsh: 2118.335373256603
MSE Haar: 5.009765625

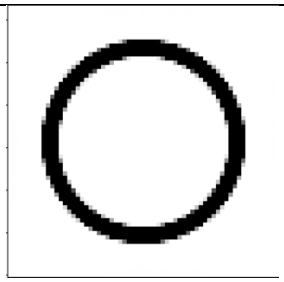


Figure 1.1 Original Image

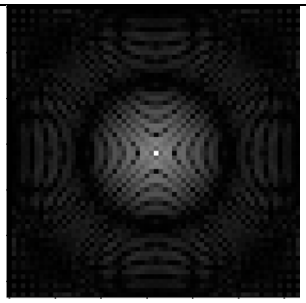


Figure 1.2 FFT Transform

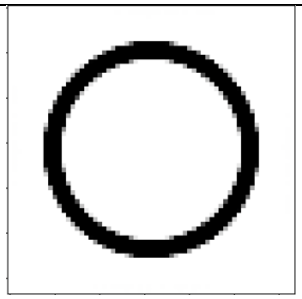


Figure 1.3 Inverse FFT Transform

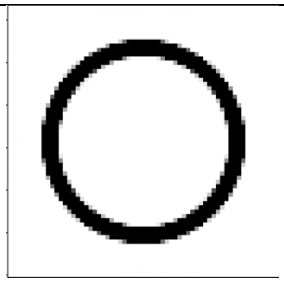


Figure 1.4 Original Image

Figure 1.5 DCT Transform

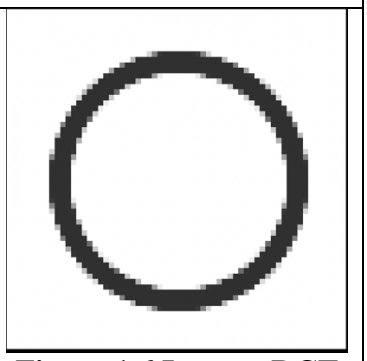


Figure 1.6 Inverse DCT Transform

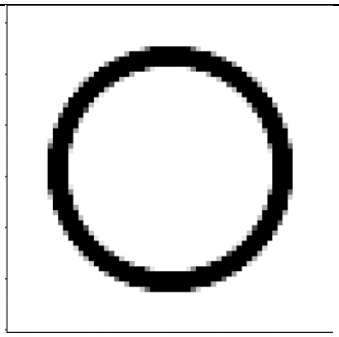


Figure 1.7 Original Image

Figure 1.8 Walsh Transform

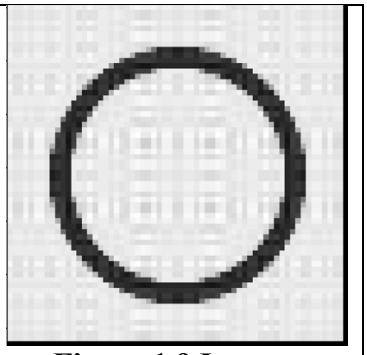


Figure 1.9 Inverse Walsh Transform

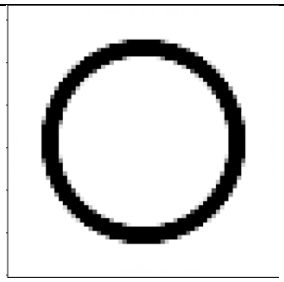


Figure 1.10 Original Image

Figure 1.11 Haar Transform

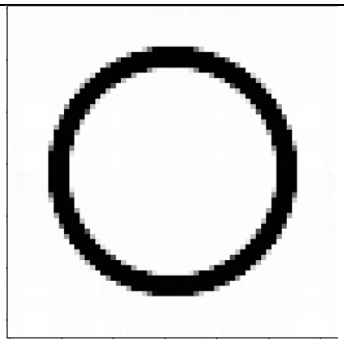


Figure 1.12 Inverse Haar Transform

Example 2:
MSE FFT: 323548035564.0039
MSE DCT: 11089.928965917652
MSE Walsh: 561.3110207612772
MSE Haar: 11.343994140625

Figure 1.1 Original Image

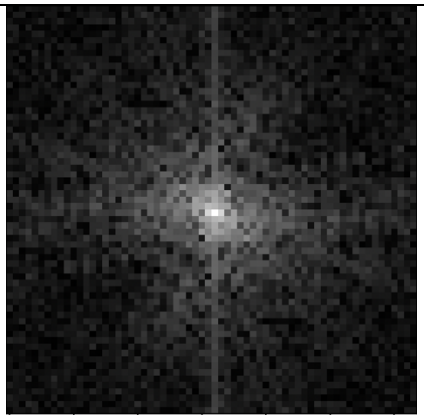


Figure 1.2 FFT Transform

Figure 1.3 Inverse FFT Transform

Figure 1.4 Original Image

Figure 1.5 DCT Transform

Figure 1.6 Inverse DCT Transform

Figure 1.7 Original Image

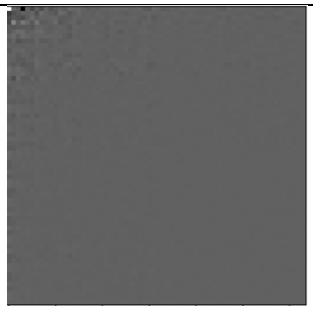


Figure 1.8 Walsh Transform

Figure 1.9 Inverse Walsh Transform

Figure 1.10 Original Image

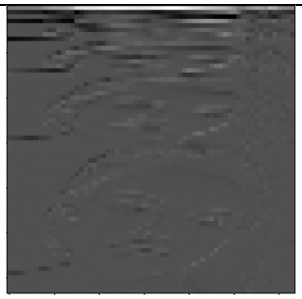
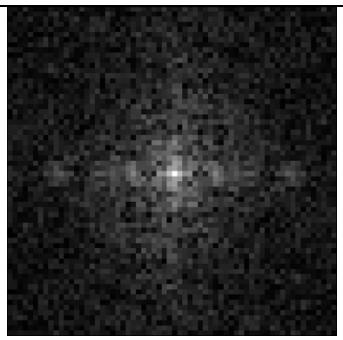
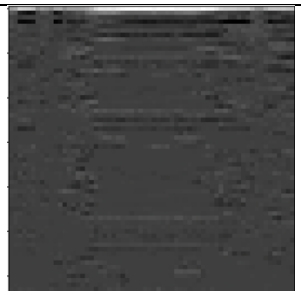


Figure 1.11 Haar Transform

Figure 1.12 Inverse Haar Transform

Example 3:
MSE FFT: 326141171921.324
MSE DCT: 11166.649670464267
MSE Walsh: 536.1491902834387
MSE Haar: 11.680908203125

 **Figure 1.1 Original Image**	 **Figure 1.2 FFT Transform**	 **Figure 1.3 Inverse FFT Transform**
 **Figure 1.4 Original Image**	 **Figure 1.5 DCT Transform**	 **Figure 1.6 Inverse DCT Transform**
 **Figure 1.7 Original Image**	 **Figure 1.8 Walsh Transform**	 **Figure 1.9 Inverse Walsh Transform**
 **Figure 1.10 Original Image**	 **Figure 1.11 Haar Transform**	 **Figure 1.12 Inverse Haar Transform**

Discussion

In this homework, we compared how different transforms store the image information. We used MSE to approximate the entropy. Our experiments demonstrate that Haar transform have the best entropy as it keeps the majority of values as is with the MSE near 11 for complex and natural images, and 5 for simple shapes. The FFT on the other hand has the worst entropy.

Nonetheless, despite a significant differences between MSE values for FFT, DCT, Haar, and Walsh transform, in all cases the images were reconstructed with a good visual accuracy. Additionally, we notice that Haar and FFT are the fastest transforms, while Walsh takes the longest computational time.

Program

FFT Comparison

```
import numpy as np
import cv2, math
from matplotlib import pyplot as plt
from timeit import default_timer as timer

def mse_between_two_images(original_image, reconstructed_image):
    differences = np.subtract(original_image, reconstructed_image)
    squared_differences = np.square(differences)
    return squared_differences.mean()

def compute_single_2d_FFT(image, u, v, N):
        result = 0 + 0j
        for x in range(N):
            for y in range(N):
                result += (image[x, y] * (math.cos((2*math.pi*(u*x + v*y))/N) -
                                          (1j*math.sin((2*math.pi*(u*x +
v*y))/N))))
        return result

def compute_forward_DFT_no_separation(imge):
        N = imge.shape[0]
        final2DDFT = np.zeros([N, N], dtype=np.complex128)
        for u in range(N):
            for v in range(N):
                final2DDFT[u, v] = compute_single_2d_FFT(imge, u, v, N)
        return ((1.0/(N**2))*final2DDFT)


def compute_single_w(num, denom):
```

```
        return math.cos((2*math.pi*num)/denom) -
(1j*math.sin((2*math.pi*num)/denom))

def compute_centered_image(imge):
        M, N = imge.shape
        newImge = np.zeros([M, N], dtype=int)
        for x in range(M):
            for y in range(N):
                newImge[x, y] = imge[x, y] * ((-1)**(x+y))

        return newImge

def compute_w(val, denom, oneD=True):
        val = int(val)
        if oneD:
            result = np.zeros([val, 1], dtype=np.complex128)
            for i in range(val):
                result[i] = compute_single_w(i, denom)
        else:
            result = np.zeros([val, val], dtype=np.complex128)
            for i in range(val):
                for j in range(val):
                    result[i, j] = compute_single_w((i+j), denom)
        return result

def fft(imge):
        #Compute size of the given image
        N = imge.shape[0]

        #Compute the FFT for the base case (which uses the normal DFT)
        if N == 2:
            return compute_forward_DFT_no_separation(imge)

        #Divide the original image into even and odd
        imgeEE = np.array([[imge[i,j] for i in range(0, N, 2)] for j in
range(0, N, 2)]).T
        imgeEO = np.array([[imge[i,j] for i in range(0, N, 2)] for j in
range(1, N, 2)]).T
        imgeOE = np.array([[imge[i,j] for i in range(1, N, 2)] for j in
range(0, N, 2)]).T
        imgeOO = np.array([[imge[i,j] for i in range(1, N, 2)] for j in
range(1, N, 2)]).T

        #Compute FFT for each of the above divided images
        FeeUV = fft(imgeEE)
        FeoUV = fft(imgeEO)
        FoeUV = fft(imgeOE)
        FooUV = fft(imgeOO)
```

```
        #Compute also Ws
        Wu = compute_w(N/2, N)
        Wv = Wu.T #Transpose
        Wuv = compute_w(N/2, N, oneD=False)

        #Compute F(u,v) for u,v = 0,1,2,...,N/2
        imgeFuv = 0.25*(FeeUV + (FeoUV * Wv) + (FoeUV * Wu) + (FooUV * Wuv))

        #Compute F(u, v+M) where M = N/2
        imgeFuMv = 0.25*(FeeUV + (FeoUV * Wv) - (FoeUV * Wu) - (FooUV * Wuv))

        #Compute F(u+M, v) where M = N/2
        imgeFuvM = 0.25*(FeeUV - (FeoUV * Wv) + (FoeUV * Wu) - (FooUV * Wuv))

        #Compute F(u+M, v+M) where M = N/2
        imgeFuMvM = 0.25*(FeeUV - (FeoUV * Wv) - (FoeUV * Wu) + (FooUV * Wuv))

        imgeF1 = np.hstack((imgeFuv, imgeFuvM))
        imgeF2 = np.hstack((imgeFuMv, imgeFuMvM))
        imgeFFT = np.vstack((imgeF1, imgeF2))

        return imgeFFT

def normalize_by_log(dftImge):
        dftFourierSpect = compute_spectrum(dftImge)

        dftNormFourierSpect = (255.0/ math.log10(255)) * np.log10(1 +
(255.0/(np.max(dftFourierSpect))*dftFourierSpect))

        return dftNormFourierSpect


def compute_spectrum(dftImge):
        N = dftImge.shape[0]

        fourierSpect = np.zeros([N, N], dtype=float)
        for i in range(N):
            for j in range(N):
                v = dftImge[i, j]
                fourierSpect[i, j] = math.sqrt((v.real)**2 + (v.imag)**2)
        return fourierSpect

def inverse_fft(imgeFFT):
        N = imgeFFT.shape[0]
        return np.real(np.conjugate(fft(np.conjugate(imgeFFT)*(N**2)))*(N**2))
```

```
if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natural_scene.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (64,64))
    plt.imshow(image, cmap='gray')
    plt.show()

    centeredImge = compute_centered_image(image)
    fft_im = fft(centeredImge)
    fftCenteredNormImge = normalize_by_log(fft_im)
    plt.imshow(fftCenteredNormImge, cmap='gray')
    plt.show()

    inverse = inverse_fft(fft_im)
    inv_centered = compute_centered_image(inverse)
    plt.imshow(inv_centered, cmap='gray')
    plt.show()

    mse = mse_between_two_images(image,inv_centered)
    print(f"MSE FFT: {mse}")
```

DCT Comparison

```
import numpy as np
import cv2, math
from matplotlib import pyplot as plt
from timeit import default_timer as timer

def mse_between_two_images(original_image, reconstructed_image):
    differences = np.subtract(original_image, reconstructed_image)
    squared_differences = np.square(differences)
    return squared_differences.mean()

def dct_2d(img):
        M, N = np.shape(img)
        dct_result = np.zeros((M, N))
        pi = math.pi
        time_start = timer()

        for u in range(M-1):
            for v in range(N-1):
                if u == 0:
                    alpha_u = math.sqrt(1/M)
                else:
                    alpha_u = math.sqrt(2/M)
```

```
                if v == 0:
                    alpha_v = math.sqrt(1/M)
                else:
                    alpha_v = math.sqrt(2/M)

                sum = 0
                for x in range(M-1):
                    for y in range(N-1):
                        cos_x = math.cos((2*x+1)*u*pi/(2*M))
                        cos_y = math.cos((2*y+1)*v*pi/(2*N))

                        temp_sum = img[x,y]*cos_x*cos_y

                        sum += temp_sum

                dct_result[u,v] = alpha_u* alpha_v * sum

        time_end = timer()
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        return dct_result


def inverse_dct_2d(transformed):
        M, N = np.shape(transformed)
        dct_result_inverse = np.zeros((M, N))

        pi = math.pi
        time_start = timer()


        for x in range(M-1):
            for y in range(N-1):
                sum = 0
                for u in range(M-1):
                    for v in range(N-1):
                        if u == 0:
                            alpha_u = math.sqrt(1/M)
                        else:
                            alpha_u = math.sqrt(2/M)
                        if v == 0:
                            alpha_v = math.sqrt(1/M)
                        else:
                            alpha_v = math.sqrt(2/M)



                        cos_x = math.cos((2*x+1)*u*pi/(2*M))
```

```
                        cos_y = math.cos((2*y+1)*v*pi/(2*N))

                        temp_sum = alpha_u *
alpha_v*transformed[u,v]*cos_x*cos_y

                        sum += temp_sum

                dct_result_inverse[x,y] = 1/4 *sum

        time_end = timer()
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        return dct_result_inverse

def get_kernel(N):
    pi = math.pi
    dct = np.zeros((N, N))
    for x in range(N):
        dct[0,x] = math.sqrt(2.0/N) / math.sqrt(2.0)
    for u in range(1,N):
        for x in range(N):
            dct[u,x] = math.sqrt(2.0/N) * math.cos((pi/N) * u * (x + 0.5) )

    return dct


if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natural_scene.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (64,64))
    plt.imshow(image, cmap='gray')
    plt.show()

    haar = get_kernel(32)
    haar_t = dct_2d(image)
    plt.imshow(haar_t, cmap='gray')
    plt.show()

    reverse = inverse_dct_2d(haar_t)
    plt.imshow(reverse, cmap='gray')
    plt.show()

    mse = mse_between_two_images(image,reverse)
    print(f"MSE DCT: {mse}")
```

Walsh Comparison

```
import numpy as np
import cv2, math
from matplotlib import pyplot as plt
from timeit import default_timer as timer

def mse_between_two_images(original_image, reconstructed_image):
    differences = np.subtract(original_image, reconstructed_image)
    squared_differences = np.square(differences)
    return squared_differences.mean()

def walsh_transform(img):

        M, N = np.shape(img)

        walsh_results = np.zeros((M, N))

        number_of_bits = int(math.log2(N))

        time_start = timer()

        for u in range(M-1):
            for v in range(N-1):
                u_bits = decimalToBinary(u)
                v_bits = decimalToBinary(v)
                u_final = fix_binary_to_lenght(u_bits, number_of_bits)
                v_final = fix_binary_to_lenght(v_bits, number_of_bits)
                sum = 0
                for x in range(M-1):
                    for y in range(N-1):
                        f_x_y = img[x,y]
                        x_bits = decimalToBinary(x)
                        y_bits = decimalToBinary(y)
                        x_final = fix_binary_to_lenght(x_bits, number_of_bits)
                        y_final = fix_binary_to_lenght(y_bits, number_of_bits)

                        temp_sum = 0
                        product = 1
                        for i in range(number_of_bits):
                            # print(i)
                            power = int(x_final[i])*int(u_final[number_of_bits-
1-i]) + int(y_final[i])*int(v_final[number_of_bits-1-i])
                            # print(f'x is {x_final[i]} and u is {u_final[n-
i]}')
                            product = product*((-1)**power)
```

```
                        temp_sum = (1/N)*(f_x_y*product)
                        sum += temp_sum
                walsh_results[u,v] = sum

        time_end = timer()
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        # print(dct_result)
        return walsh_results


def inverse_walsh_transform(transform):

        M, N = np.shape(transform)

        invese_walsh_results = np.zeros((M, N))

        number_of_bits = int(math.log2(N))

        time_start = timer()

        for u in range(M-1):
            for v in range(N-1):
                u_bits = decimalToBinary(u)
                v_bits = decimalToBinary(v)
                u_final = fix_binary_to_lenght(u_bits, number_of_bits)
                v_final = fix_binary_to_lenght(v_bits, number_of_bits)
                sum = 0
                for x in range(M-1):
                    for y in range(N-1):
                        f_x_y = transform[x,y]
                        x_bits = decimalToBinary(x)
                        y_bits = decimalToBinary(y)
                        x_final = fix_binary_to_lenght(x_bits, number_of_bits)
                        y_final = fix_binary_to_lenght(y_bits, number_of_bits)

                        temp_sum = 0
                        product = 1
                        for i in range(number_of_bits):
                            power = int(x_final[i])*int(u_final[number_of_bits-
1-i]) + int(y_final[i])*int(v_final[number_of_bits-1-i])
                            product = product*((-1)**power)

                        temp_sum = (1/N)*(f_x_y*product)
                        sum += temp_sum
                invese_walsh_results[u,v] = sum

        time_end = timer()
```

```
        time_elapsed = time_end - time_start
        print(f"Total execution time is: {time_elapsed}")
        # print(dct_result)
        return invese_walsh_results

def get_kernel(N):
    walsh = np.zeros((N, N))
    number_of_bits = int(math.log2(N))
    count_x = 0
    for x in range(N):
        count_x += 1
        count_y = 0
        for u in range(N):
            count_y += 1
            x_bits = decimalToBinary(x)
            u_bits = decimalToBinary(u)
            x_final = fix_binary_to_lenght(x_bits, number_of_bits)
            u_final = fix_binary_to_lenght(u_bits, number_of_bits)

            product = 1
            for i in range(number_of_bits):
                power = int(x_final[i])*int(u_final[number_of_bits-1-i])
                product = product*((-1)**power)

            walsh[x,u] = (1/N)*product
    return walsh

def fix_binary_to_lenght(binary, lenght):
    while len(binary) < lenght:
        binary = '0'+binary
    return binary

def decimalToBinary(n):
    return bin(n).replace("0b", "")


if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natural_scene.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (64,64))
    plt.imshow(image, cmap='gray')
    plt.show()

    walsh = walsh_transform(image)
    plt.imshow(walsh, cmap='gray')
    plt.show()
```

```
    reverse = inverse_walsh_transform(walsh)
    plt.imshow(reverse, cmap='gray')
    plt.show()

    mse = mse_between_two_images(image,reverse)
    print(f"MSE Walsh: {mse}")
```

Haar Comparison

```
import numpy as np
import cv2, math
from matplotlib import pyplot as plt
from timeit import default_timer as timer

def mse_between_two_images(original_image, reconstructed_image):
    differences = np.subtract(original_image, reconstructed_image)
    squared_differences = np.square(differences)
    return squared_differences.mean()

def get_kernel(N):
    haar = np.zeros((N, N))
    x_values = []

    for i in range(0, N):
        x = (i)/N
        x_values.append(x)

    min_r = 0
    max_r = int(math.log2(N))
    rm_tuples = []

    for r in range(min_r, max_r):
        min_m = 1
        max_m = 2**r
        for m in range(min_m, max_m+1):
            rm_tuples.append((r,m))

    for u in range(N):
        r,m = rm_tuples[u-1]
        min_1_included = (m-1)/(2**r)
        max_1_not_included = (m-0.5)/(2**r)
        min_2_included = (m-0.5)/(2**r)
        max_2_not_included = (m)/(2**r)
        for v in range(N):
            if u == 0:
                haar_value = 1/math.sqrt(N)
```

```
            else:
                x = x_values[v]
                if x>= min_1_included and x< max_1_not_included:
                    haar_value = (2**(r/2))/(math.sqrt(N))
                elif x>= min_2_included and x< max_2_not_included:
                    haar_value = -(2**(r/2))/(math.sqrt(N))
                else:
                    haar_value = 0
            haar[u,v] = haar_value*(math.sqrt(N))

    return haar

def fix_binary_to_lenght(binary, lenght):
    while len(binary) < lenght:
        binary = '0'+binary
    return binary


def ordered_kernel(haar_transform):
    skips_number = []
    for row in haar_transform:
        current_value = row[0]
        skips = 0
        for i in row:
            temp_value = i
            if temp_value != current_value:
                skips += 1
            current_value = temp_value
        skips_number.append(skips)
    ordered_transform = np.zeros((haar_transform.shape))
    for original_position, ordered_position in enumerate(skips_number):
        ordered_transform[ordered_position] = haar_transform[original_position]

    return ordered_transform

def multiply_matrix(A,B):
  global C
  if  A.shape[1] == B.shape[0]:
    C = np.zeros((A.shape[0],B.shape[1]),dtype = int)
    for row in range(A.shape[0]):
        for col in range(B.shape[1]):
            for elt in range(len(B)):
              C[row, col] += A[row, elt] * B[elt, col]
    return C
  else:
    return "Sorry, cannot multiply A and B."

def decimalToBinary(n):
```

```
    return bin(n).replace("0b", "")


if __name__ == "__main__":

    image_path = "/Users/marynavek/Projects/ImageProcessing/natural_scene.png"

    image = cv2.imread(image_path, 0)
    image = cv2.resize(image, (64,64))
    plt.imshow(image, cmap='gray')
    plt.show()

    haar = get_kernel(64)
    haar_t = multiply_matrix(haar,image)
    plt.imshow(haar_t, cmap='gray')
    plt.show()

    reverse = multiply_matrix(np.linalg.inv(haar),haar_t)
    plt.imshow(reverse, cmap='gray')
    plt.show()

    mse = mse_between_two_images(image,reverse)
    print(f"MSE Haar: {mse}")
```