ECE 558 – Project 02

Devadharshini Ayyappan

Question1

(a) Code:

```
import numpy as np
import cv2
import math
image=cv2.imread('Lenna.png')

#convolution function definition
def conv2(f, w, pad):
    if(w==1):
        kernel1 = box_filter
        kernel2 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    if(w==2):
        kernel1= derivative_x
        kernel2 = derivative_y
    elif(w==3):
        kernel1 = prewitt_x
        kernel2 = prewitt_y
    elif(w==4):
        kernel1 = sobel_x
        kernel2 = sobel_y
    elif(w==5):
        kernel1 = roberts_x
        kernel2 = roberts_y
     elif(w==6):
        kernel1 = prewitt_x
        kernel2 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    elif(w==7):
        kernel1 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        kernel2 = prewitt_y
    elif(w==8):
        kernel1 = sobel_x
        kernel2 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    elif(w==9):
        kernel1 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        kernel2 = sobel_y
    elif(w==10):
        kernel1 = roberts_x
        kernel2 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    elif(w==11):
```

```
        kernel1 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        kernel2 = roberts_y

#kernel is flipped since as per convolution – one of the 2 functions has to be flipped
    kernel1 = np.fliplr(np.flipud(kernel1))
    kernel2 = np.fliplr(np.flipud(kernel2))
    #x is the padding width which varies based on kernel size

x=int((max(kernel1.shape[0],kernel1.shape[1],kernel2.shape[1],kernel2.shape[0]))/2)
    #padded image is created based on kernel size
    if(len(f.shape) == 3):
            image_padded
=np.zeros((f.shape[0]+(x*2),f.shape[1]+(x*2),f.shape[2]),dtype=np.uint8)
    else:
            image_padded=np.zeros([f.shape[0]+(x*2),f.shape[1]+(x*2)])
    m = image_padded.shape[0]
    n = image_padded.shape[1]
#selecting padding type:
    #Zero Padding
            image_padded[x: m-x, x: n-x] = f    #Wrap Around Padding
#Wrap Around Padding
    if(pad==2):
        for i in range(1,x+1):
            image_padded[m-i, x: n-x] = image_padded[x-1+i, x: n-x]
            image_padded[i-1, x: n-x] = image_padded[m-x-i, x: n-x]
            image_padded[x: m-x, i-1] = image_padded[x: m-x, n-x-i]
            image_padded[x: m-x, n-i] = image_padded[x: m-x, x-1+i]
            image_padded[0:x, 0:x] = image_padded[m-x-x: m-x, n-x-x: n-x]
            image_padded[m-x:m, n-x:n] = image_padded[x: 2*x, x: 2*x]
            image_padded[0:x, n-x:n] = image_padded[m-x-x: m-x, x: 2*x]
            image_padded[m-x:m, 0:x] = image_padded[x: 2*x, n-x-x: n-x]
    #Copy Edge Padding
    elif(pad==3):
        for i in range(1,x+1):
            image_padded[m-x:m, x: n-x] = image_padded[m-x-1, x: n-x]
            image_padded[0:x, x: n-x] = image_padded[x+1, x: n-x]
            image_padded[x: m-x, i-1] = image_padded[x: m-x, x+1]
            image_padded[x: m-x, n-i] = image_padded[x: m-x, n-x-1]
            image_padded[0:x, 0:x] = image_padded[x,x]
            image_padded[m-x:m, n-x:n] = image_padded[m-x-1,n-x-1]
            image_padded[0:x, n-x:n] = image_padded[x,n-x]
            image_padded[m-x:m, 0:x] = image_padded[m-x,x]
    #Reflect Across Edge
    elif(pad==4):
        for i in range(1,x+1):
            image_padded[m-i, 0: n] = image_padded[(m-1)-(2*x)+i, 0: n]
            image_padded[0: m, n-i] = image_padded[0: m, (n-1)-(2*x)+i]
```

```
            image_padded[i-1, 0: n] = image_padded[(2*x)-i, 0: n]
            image_padded[0: m, i-1] = image_padded[0: m, (2*x)-i]

    #output image is initiated with zeros with the same dimensions and type as input image
    output_image = np.zeros_like(f)

    #convolution function is performed here based on the selected color and gray images:
    for x in range(f.shape[0]):
            for y in range(f.shape[1]):
                #For color image
                if(len(f.shape)==3):
                    output_image[x,y,2]=math.sqrt(((kernel1*image_padded[x:x + kernel1.shape[0], y:y + kernel1.shape[1], 2]).sum())**2 + ((kernel2*image_padded[x:x + kernel2.shape[0], y:y + kernel2.shape[1], 2]).sum())**2)
                    output_image[x,y,1]=math.sqrt(((kernel1*image_padded[x:x + kernel1.shape[0], y:y + kernel1.shape[1], 1]).sum())**2 + ((kernel2*image_padded[x:x + kernel2.shape[0], y:y + kernel2.shape[1], 1]).sum())**2)
                    output_image[x,y,0]=math.sqrt(((kernel1*image_padded[x:x + kernel1.shape[0], y:y + kernel1.shape[1], 0]).sum())**2 + ((kernel2*image_padded[x:x + kernel2.shape[0], y:y + kernel2.shape[1], 0]).sum())**2)
                #For gray scale image
                else:
                    output_image[x,y]=math.sqrt(((kernel1*image_padded[x:x + kernel1.shape[0], y:y + kernel1.shape[1]]).sum())**2 + ((kernel2*image_padded[x:x + kernel2.shape[0], y:y + kernel2.shape[1]]).sum())**2)
    return output_image
#convolution function ends

#main function

#Question 1. Part a
#selecting the kernel
kernel_type = ["1. Box filter", "2. First Order Derivative Filter", "3. Prewitt", "4. Sobel", "5. Roberts"]
print("Different Kernel Types: " ,*kernel_type, sep=" \n")
select_kernel = int(input("Enter the type of kernel: "))
#filters are defined
derivative_x = np.array([[-1, 1]])
derivative_y = np.array([[-1],[ 1]])
prewitt_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
prewitt_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
roberts_x = np.array([[0, 1], [-1, 0]])
roberts_y = np.array([[1, 0], [0, -1]])
box_filter = 1/9*(np.ones([3,3]))
```

```
#selecting the padding type
pad_type = ["1. Zero", "2. Wrap Around", "3. Copy Edge", "4. Reflect Across Edge"]
print("Types of padding available: " ,*pad_type, sep=" \n")
select_padding = int(input("Enter the type of padding: "))
#calling the function
output_parta = conv2(image, select_kernel, select_padding)

#Question 1. Part b
#defining unit impulse
unit_impulse = np.zeros([1024,1024])
unit_impulse[512,512] = 255
#selecting the kernel
kernel_type = ["1. Box filter", "2. First Order Derivative Filter", "3. Prewitt", "4.
Sobel", "5. Roberts"]
print("Different Kernel Types: " ,*kernel_type, sep=" \n")
select_kernel = int(input("Enter the type of kernel: "))
#selecting the padding type
pad_type = ["1. Zero", "2. Wrap Around", "3. Copy Edge", "4. Reflect Across Edge"]
print("Types of padding available: " ,*pad_type, sep=" \n")
select_padding = int(input("Enter the type of padding: "))
#calling the function
unit_impulse_output = conv2(unit_impulse, select_kernel, select_padding)
#displaying the convolved sliced matrix at the center
print("Impulse_output", unit_impulse_output[510:515,510:515], sep=" \n")


cv2.imshow("Input Image(a)", image)
cv2.imshow("Result Image(a)", output_parta)
cv2.imshow("Input Image(b)", unit_impulse)
cv2.imshow("Result Image(b)", unit_impulse_output)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Explanation of the convolution function:

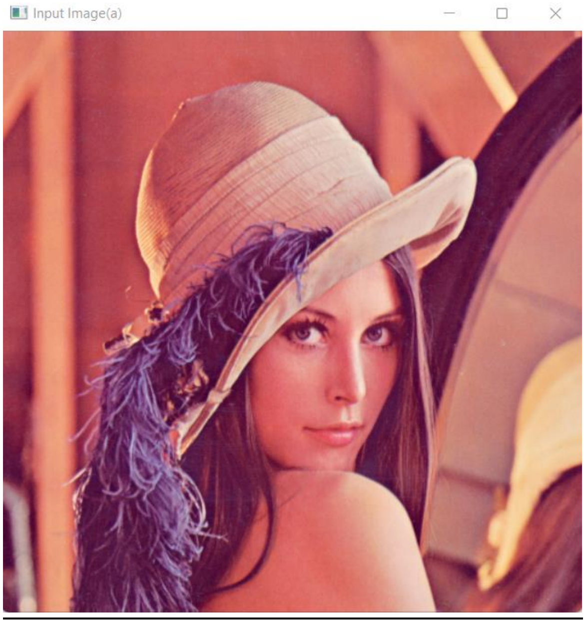
In the convolution function, since first order derivatives are also present, I have implemented using the magnitude of the gradient. Hence, I have taken square root and in order to normalize for box filters and x- or y- only filters, I have assigned the kernel2 as zeros respectively.

$$M(x, y) = \sqrt{g_x^2 + g_y^2}$$

(Refered from the book (DIP by Rafael C. Gonzalez) for 1st order derivatives)

In the python code attached, I have added the individual X and Y filters for prewit, sobel and Roberts.

Input image: Part A

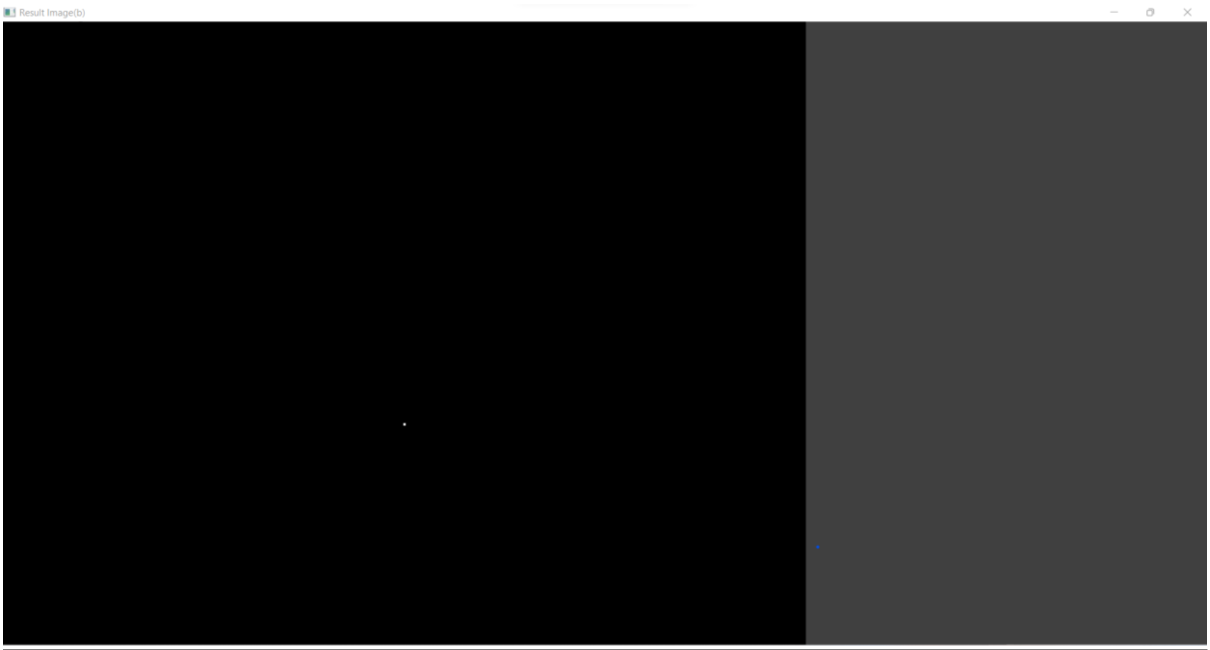


Outputs: Part A

The outputs of PartA are attached as .jpg in the zip folder

Input image: Part B

Output image: Part B (Box Kernel, Zero padding)



Output Matrix:

```
Different Kernel Types:
1. Box filter
2. First Order Derivative Filter-xy
3. Prewitt-xy
4. Sobel-xy
5. Roberts-xy
6. Prewitt-x
7. Prewitt-y
8. Sobel-x
9. Sobel-y
10. Roberts-x
11. Roberts-y
Enter the type of kernel: 1
Types of padding available:
1. Zero
2. Wrap Around
3. Copy Edge
4. Reflect Across Edge
Enter the type of padding: 1
Impulse_output
[[ 0.          0.          0.          0.          0.        ]
 [ 0.         28.33333333 28.33333333 28.33333333  0.        ]
 [ 0.         28.33333333 28.33333333 28.33333333  0.        ]
 [ 0.         28.33333333 28.33333333 28.33333333  0.        ]
 [ 0.          0.          0.          0.          0.        ]]
```

As we can see from the sliced output matrix, the convolution operation is indeed performed on the image since the calculations matches the manually calculated values:

Image $[512, 512] = 255$

image size ⇒ 1024×1024

zero-padding & box-filter is applied

→ Taking the middle pixels:

	510	511	512	513	514
510	0	0	0	0	0
511	0	0	0	0	0
512	0	0	255	0	0
513	0	0	0	0	0
514	0	0	0	0	0

⇒ Box filter ⇒ $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

Performing convolution we get,

$$\text{Image}(511, 511) = \frac{1}{9}\left[0+0+0+0+0+0+0+0+255\right]$$

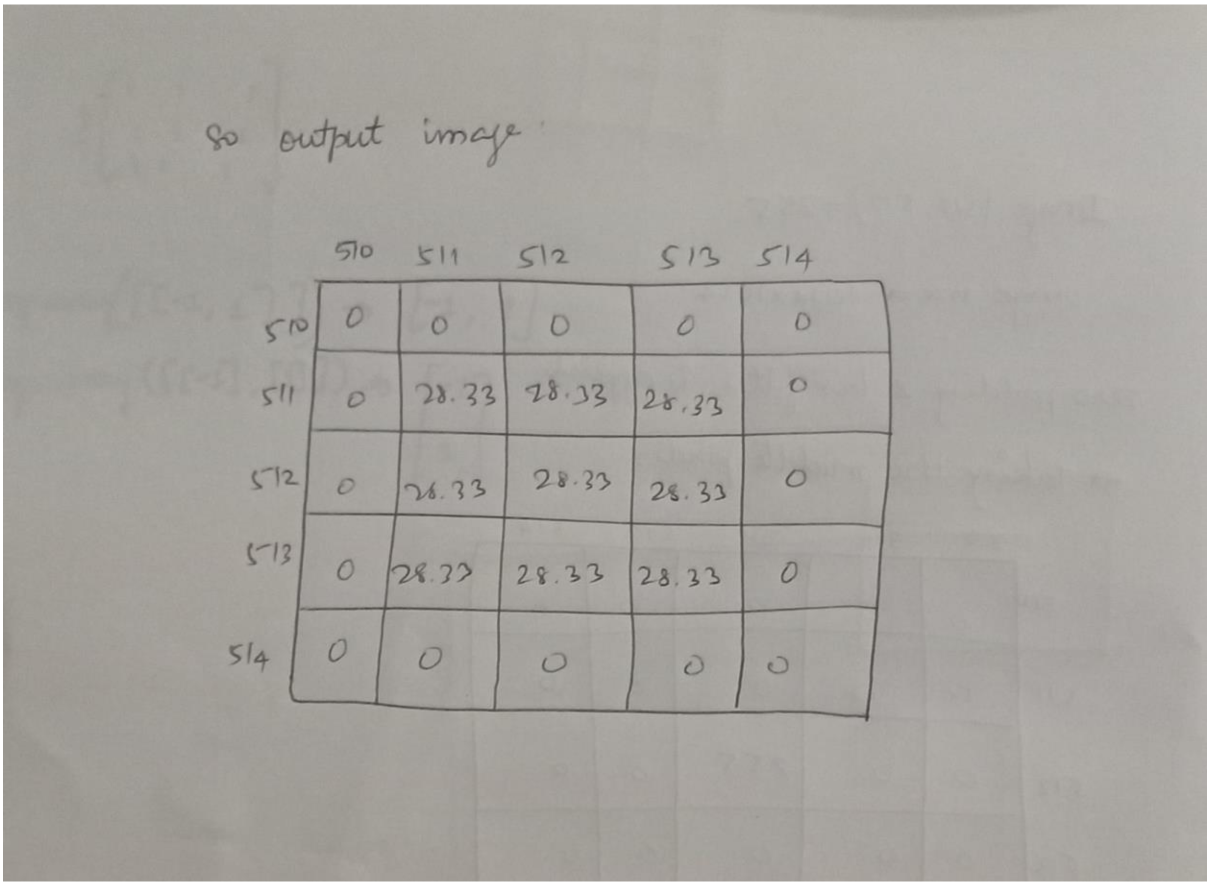
$$= \frac{255}{9}$$

$$= 28.333$$

⇒ Similarly for

$$\text{Image}(511, 512) = (511, 513) = (512, 511) = (512, 512) = (512, 513) =$$

$$(513, 511) = (513, 512) = (513, 513) = \frac{255}{9} = 28.333$$



So output image

	510	511	512	513	514
510	0	0	0	0	0
511	0	28.33	28.33	28.33	0
512	0	28.33	28.33	28.33	0
513	0	28.33	28.33	28.33	0
514	0	0	0	0	0

Also, please find the padded image outputs and the padding function in " Project02_Q1_only_padding.ipynb" attached in the zip file, where I have tested only the different types of padding with pad_width (x) =20 for the Lenna image.

<u>Question2:</u>

In previous HW, L was 100. Here L=2.

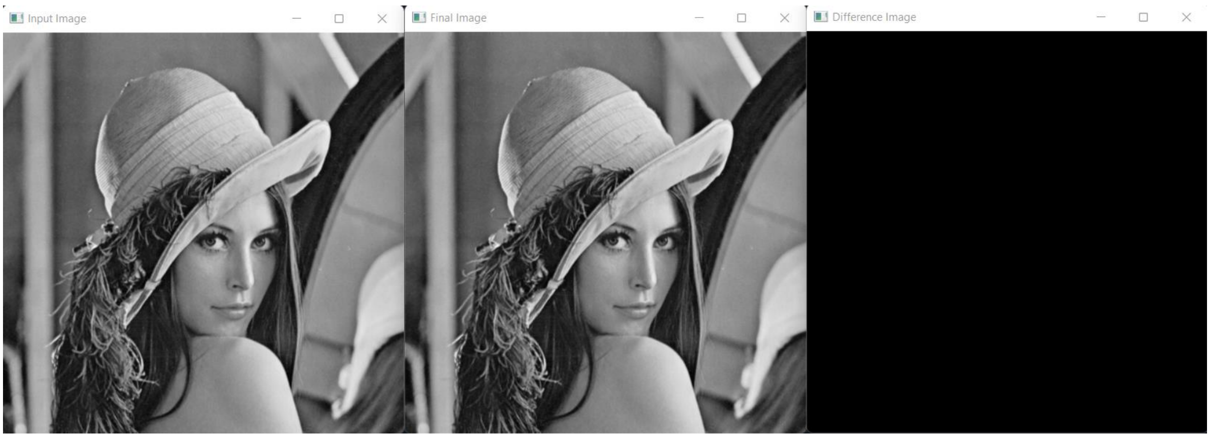
```
import numpy as np

import cv2

import matplotlib.pyplot as plt

image = cv2.imread('lena.png')

image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#Scaling the grey image to the range [0, 1] – from previous HW

image_gray_flat=image_gray.flatten()

image_gray1 = np.zeros_like(image_gray)

k = min(image_gray_flat)

l = max(image_gray_flat)
```

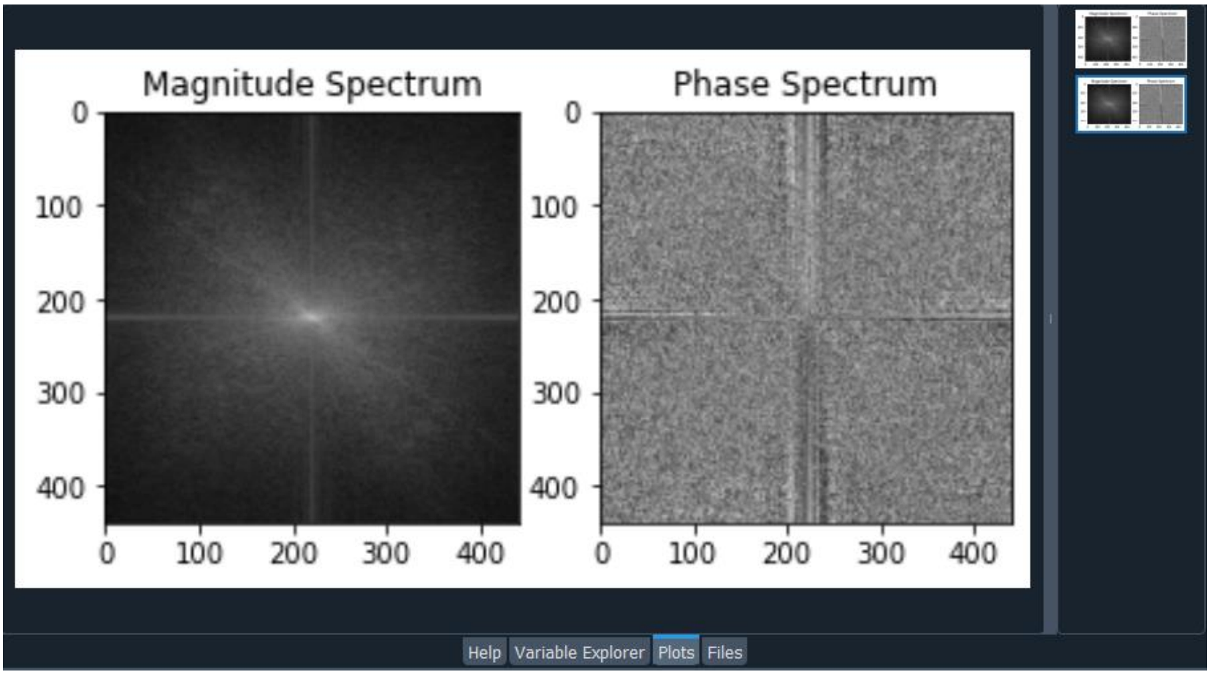
```
L=2
for i in range(image_gray.shape[0]):
        for j in range(image_gray.shape[1]):
                image_gray1[i,j] = (image_gray[i,j] - k)
image_gray_flat1 = image_gray1.flatten()
image_gray2 = ((L-1)*image_gray1)/max(image_gray_flat1)
#Computing fourier transform using the built-in 1-D FFT
def DFT2(f):
   col_fft = np.zeros_like(f, dtype="complex_")
   row_fft = np.zeros_like(f, dtype="complex_")
   for x in range(f.shape[0]):
      row_fft[x, 0:f.shape[1]] = np.fft.fft(f[x, 0:f.shape[1]])
   for y in range(row_fft.shape[1]):
      col_fft[0: row_fft.shape[0], y] = np.fft.fft(row_fft[0: row_fft.shape[0], y])
   return col_fft
#idft
def IDFT2(g):
   col_ifft = np.zeros_like(g, dtype="complex_")
   row_ifft = np.zeros_like(g, dtype="complex_")
   fftShift_b = np.fft.ifftshift(g)
   for y in range(fftShift_b.shape[1]):
      col_ifft[0: fftShift_b.shape[0], y] = np.fft.ifft(fftShift_b[0: fftShift_b.shape[0], y])
   for x in range(col_ifft.shape[0]):
      row_ifft[x, 0: col_ifft.shape[1]] = np.fft.ifft(col_ifft[x, 0: col_ifft.shape[1]])
   return row_ifft

#Question2(a) :
F = DFT2(image_gray2)
#Shifting the zero-frequency component to the center of the spectrum.
fftShift_a = np.fft.fftshift(F)
#Computing the magnitude and phase spectrum of the transform
magnitude = np.log(1 + np.abs(fftShift_a))
```

```
phase = np.angle(fftShift_a)

cv2.imshow('Input Image',image_gray2)

#Displaying the magnitude and phase spectrum of the transform

(fig1, ax) = plt.subplots(1, 2, )

ax[0].imshow(magnitude, cmap="gray")

ax[0].set_title("Magnitude Spectrum")

#Magnitude image

ax[1].imshow(phase, cmap="gray")

ax[1].set_title("Phase Spectrum")

# show our plots

plt.show()
```

```
#Question2(b) :

G = IDFT2(fftShift_a);

#checking difference between the input gray image and the image obtained after applying the inverse fourier transform

difference_image = image_gray2 - G.real

#Final image and the difference image

cv2.imshow('Final Image',G.real)

cv2.imshow('Difference Image', difference_image)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Output:




Magnitude Spectrum
Phase Spectrum
0
100
200
300
400
0
100
200
300
400
Help
Variable Explorer
Plots
Files