Digital Imaging Systems Project- 02

- Tanisha Khurana
(Unity ID – tkhuran3
student ID - 200483139)

Problem 1

Write a function to implement g= conv2(f, w, pad), where f is an input image (grey, or RGB), w is a 2-D kernel (e.g., 3 × 3 box filter), and pad represents the 4-padding type clip/zero-padding, wrap around, copy edge, and reflect across edge

First, we import the libraries

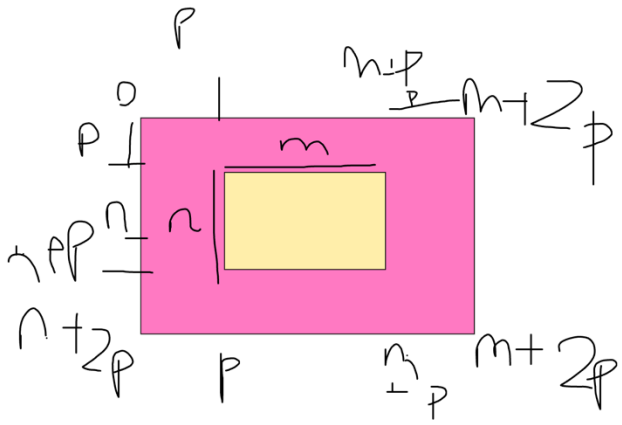
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%config IPCompleter.greedy=True
```

Built in function to read image file

```
img_bgr = cv2.imread("/Users/tanishakhurana/Desktop/tkhuran3_project02/lena.png")
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
plt.imshow(img_rgb)
```

Zero padding:

```
def zero_padding(padding,img):
   if len(img.shape) ==2:
      img_padded= np.zeros((img.shape[0] + 2 *padding, img.shape[1] + 2 *padding))
      #padding the image
      img_padded[padding: img_padded.shape[0]- padding, padding: img_padded.shape[1]-
padding] = img
      cv2.imwrite("img_gray_padded.png",img_padded)

   if len(img.shape) ==3:
      img_padded= np.zeros((img.shape[0] + 2 *padding, img.shape[1] + 2
*padding,img.shape[2]))
      #padding the image
      img_padded[padding: img_padded.shape[0]- padding, padding: img_padded.shape[1]-
padding] = img
      cv2.imwrite("img_bgr_padded.png",img_padded)

   return img_padded
```



Reflection padding

```
def reflection_padding(padding,img):
    img_padded= zero_padding(padding,img)

    right_padding_edge = img[:,:padding]
    left_padded_edge = img[:,-padding:]

    # flip the edges
    flipped_right_padded_edge = fliph(right_padding_edge)
    flipped_left_padded_edge = fliph(left_padded_edge)

    img_padded[padding:-padding,0:padding] = flipped_right_padded_edge
    img_padded[padding:-padding,-padding:] = flipped_left_padded_edge

    #up and down
    up_padded_edge = img_padded[padding:2 *padding,:]
    down_padded_edge = img_padded[img.shape[0]:img.shape[0]+padding,:]

    #flip up and down
    flipped_up_padded_edge = flipv(up_padded_edge)
    flipped_down_padded_edge = flipv(down_padded_edge)

    img_padded[:padding,:] = flipped_up_padded_edge
    img_padded[img.shape[0]+padding:,:] = flipped_down_padded_edge

    return img_padded
```

Flipped functions:

To flip vertically:

```
def flipv(img):
   if len(img.shape) ==2:
      img2= np.zeros([img.shape[0], img.shape[1]], np.uint8)
      for i in range(img.shape[0]):

         img2[i,:]=img[img.shape[0]-i-1,:]
   if len(img.shape) ==3:
      img2= np.zeros([img.shape[0], img.shape[1], img.shape[2]], np.uint8)
      for i in range(img.shape[0]):

         img2[i,:]=img[img.shape[0]-i-1,:]

   return img2
```

To flip horizontally:

```
def fliph(img):
   if len(img.shape) ==2:
      img2= np.zeros([img.shape[0], img.shape[1]], np.uint8)
      for i in range(img.shape[1]):

         img2[:,i]=img[:,img.shape[1]-i-1]
   if len(img.shape) ==3:
      img2= np.zeros([img.shape[0], img.shape[1],img.shape[2]], np.uint8)
      for i in range(img.shape[1]):

         img2[:,i]=img[:,img.shape[1]-i-1]

   return img2
```

Copy edge padding:

```
def copy_edge_padding(padding,img):
   img_padded= zero_padding(padding,img)

   right_edge = img[:,:1]
   left_edge = img[:,-1:]

   #only the last edge is needed
   img_padded[padding:-padding,0:padding] = right_edge
   img_padded[padding:-padding,-padding:] = left_edge
```

```
    down_edge = img_padded[img.shape[0] + padding -1:-padding,:]
    up_edge = img_padded[padding : padding +1,:]

    img_padded[img.shape[0]+padding:,:] = down_edge
    img_padded[:padding,:] = up_edge

    return img_padded
```

Wrap around padding:

```
def wrap_around(padding,img):
    img_padded= zero_padding(padding,img)

    right_padding_edge = img[:,:padding]
    left_padded_edge = img[:,-padding:]

    #opposite side edges
    img_padded[padding:-padding,0:padding] = left_padded_edge
    img_padded[padding:-padding,-padding:] = right_padding_edge

    #up and down
    up_padded_edge = img_padded[padding:2 *padding,:]
    down_padded_edge = img_padded[img.shape[0]:img.shape[0]+padding,:]

    img_padded[:padding,:] = down_padded_edge
    img_padded[img.shape[0]+padding:,:] = up_padded_edge

    return img_padded
```

Get padding type: whether it is zero padding, reflection padding, wrap around or copy edge

```
def get_padding_type(pad_type,img):
    if pad_type =="zero":
        img_padded = zero_padding(padding,img)
    if pad_type =="wrap around":
        img_padded = wrap_around(padding,img)
    if pad_type =="copy edge":
        img_padded = copy_edge_padding(padding,img)
    if pad_type =="reflect across edge":
        img_padded = reflection_padding(padding,img)
    return img_padded
```

Get new convolved image shape : F +K -1
Where F is image, K is kernel

```
def get_convolved_img_shape(img_padded, kernel):
   #convolved image shape =  M - n +1
   if len(img_padded.shape) == 2:
      new_img = np.zeros((img_padded.shape[0] - kernel.shape[0] + 1 ,img_padded.shape[1] -
kernel.shape[1] +1 ))

   if len(img_padded.shape) == 3:
      new_img = np.zeros((img_padded.shape[0] - kernel.shape[0] + 1 ,img_padded.shape[1] -
kernel.shape[1] +1,img_padded.shape[2]))

   return new_img
```

Convolution Function:

```
def conv(img,kernel,pad):
   img_padded = get_padding_type(pad,img)
   new_img = get_convolved_img_shape(img_padded, kernel)
   if len(img_padded.shape) ==2:
      # convolution grayscale
      for row in range(new_img.shape[0]):
         for col in range(new_img.shape[1]):
            new_img[row,col] = np.sum(img_padded[row:row+kernel.shape[0],col:col +
kernel.shape[1]] * kernel)

   if len(img_padded.shape) ==3:
      # #convolution rgb
      for row in range(new_img.shape[0]):
         for col in range(new_img.shape[1]):
            for channel in range(new_img.shape[2]):
               new_img[row,col,channel] = np.sum(img_padded[row:row+kernel.shape[0],col:col +
kernel.shape[1],channel] * kernel)

   return new_img
```

Types of kernels

In [26]:
```
box_kernel = np.array([
    [1/9, 1/9, 1/9],
    [1/9 , 1/9, 1/9],
    [1/9, 1/9, 1/9]])
```

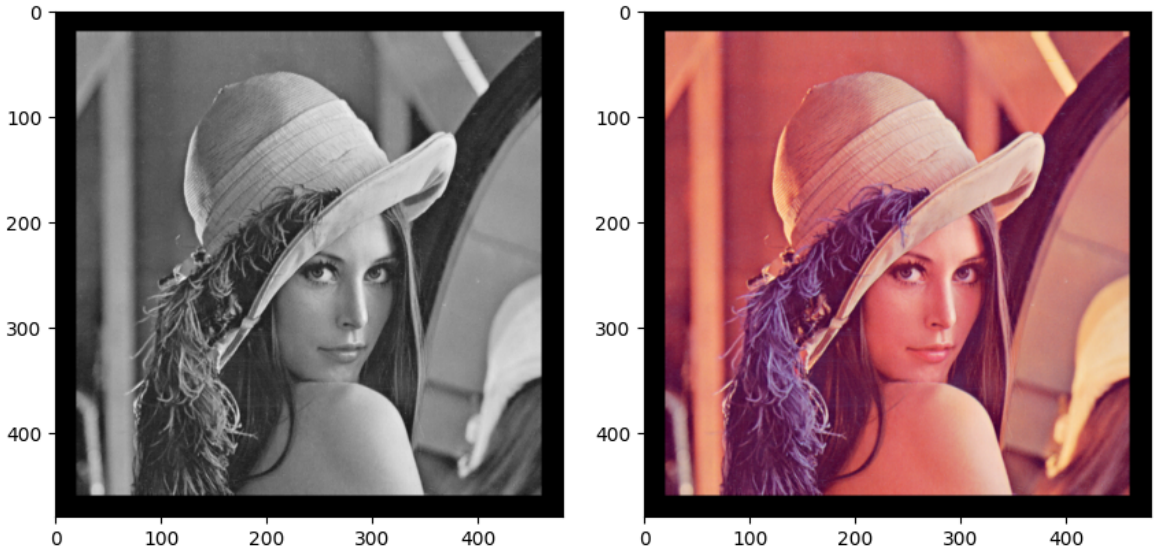
In [27]:
```
first_derivative_filter1 = np.array([[-1,1]])
first_derivative_filter2 = np.array([[-1],[1]])
```

In [28]:
```
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]])
```

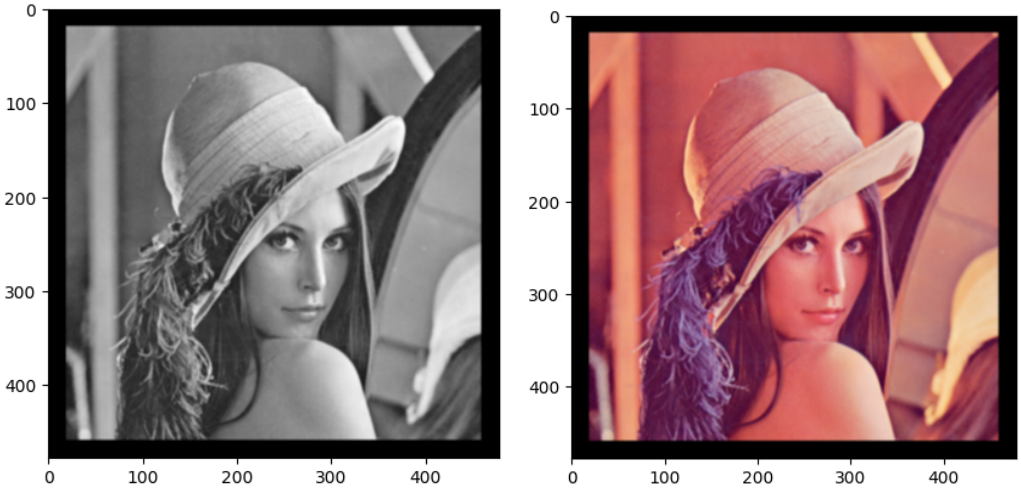
In [29]:
```
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]])

sobel_y = np.array([
    [1, 2, 1],
    [0, 0, 0],
    [-1, -2, -1]])
```

In [30]:
```
roberts_x = np.array([
    [0,1],
    [-1,0]])

roberts_y = np.array([
    [1,0],
    [0,-1]])
```

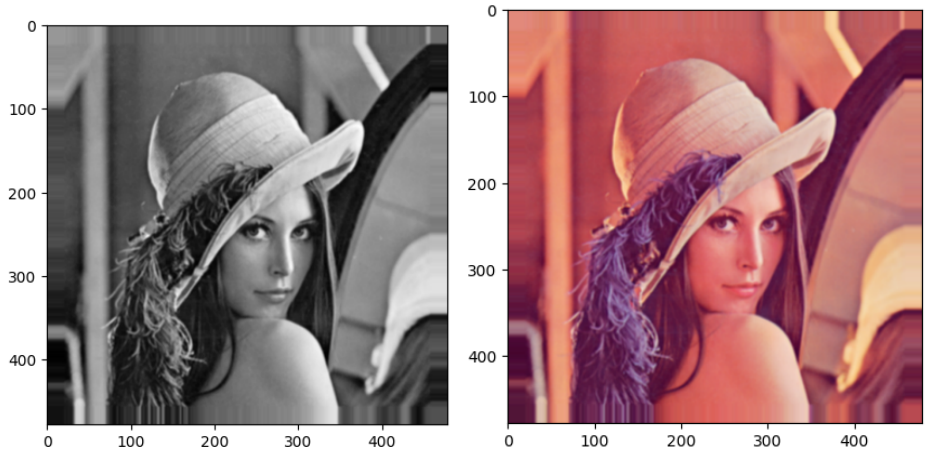
Let padding = 20
So if original shape = (440,440) , padded image will be (480,480)
Zero padding original image of lena for Grayscale and RGB



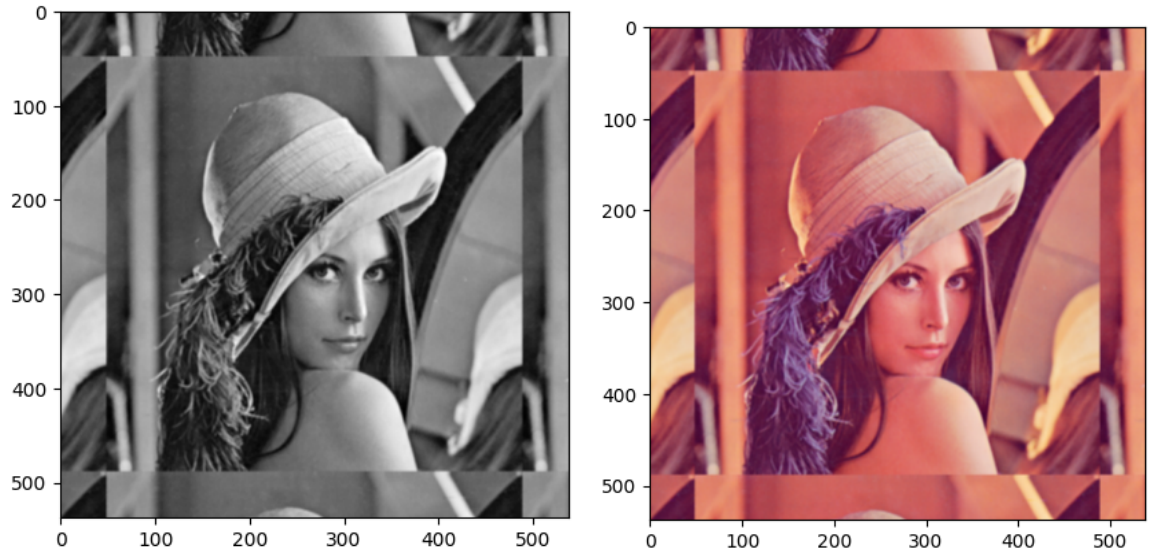
Convolved Image for box filter is:



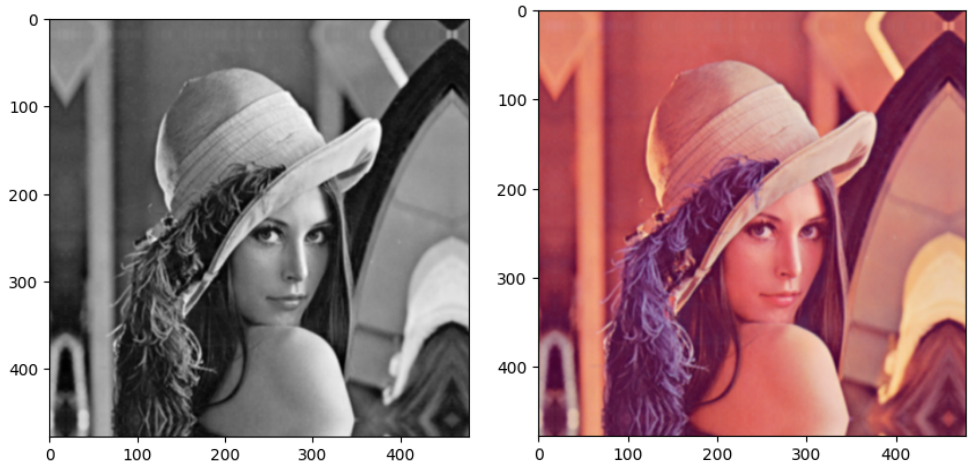
Convoluted image for box kernel for copy edge padding is:



Convoluted image for box kernel for wrap around padding is:



Convoluted image for box kernel for reflection padding is:

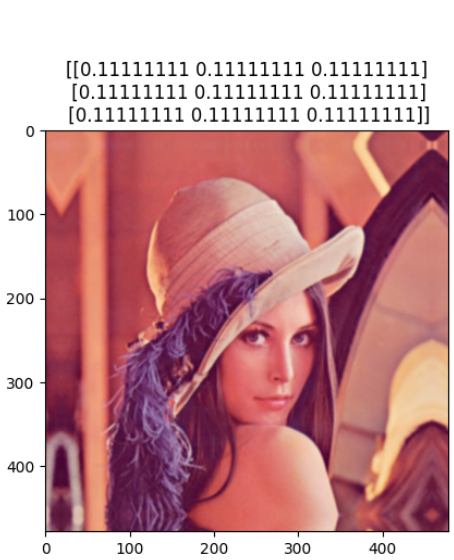


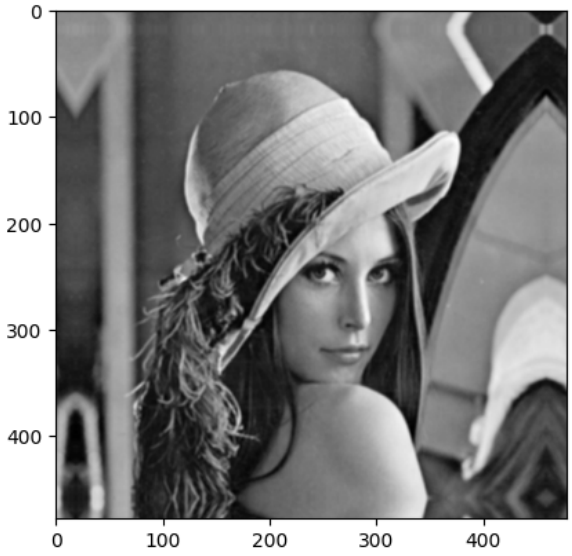
For reflection padding and lena image:

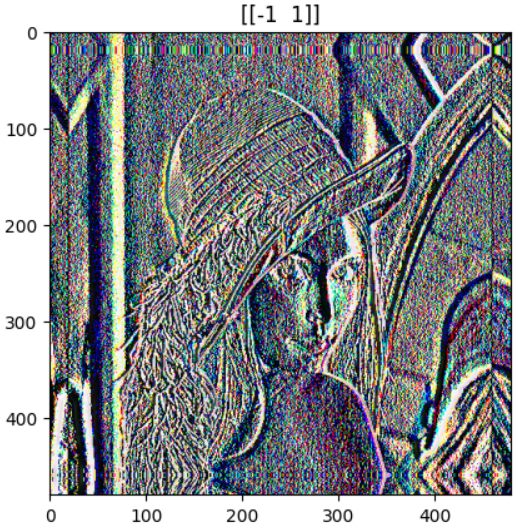
Changing different kernels

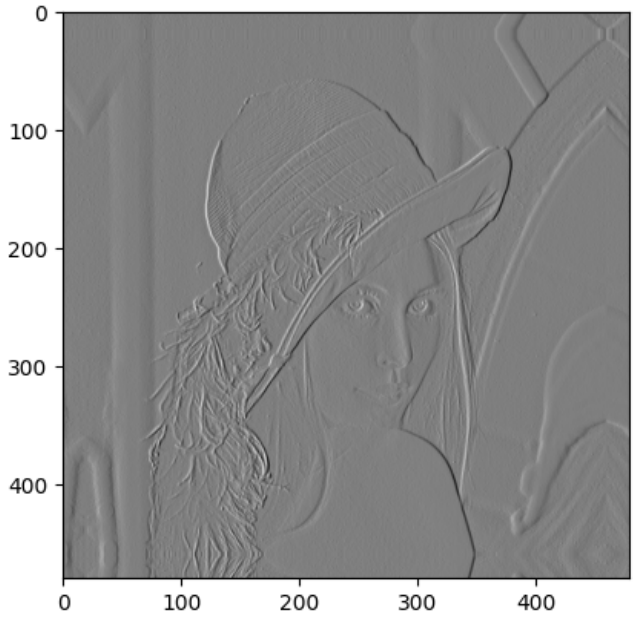
```
In order of list:

kernel_list = [box_kernel,first_derivative_filter1 ,
first_derivative_filter2,prewitt_x,prewitt_y, sobel_x, sobel_y,
roberts_x,roberts_y]
```

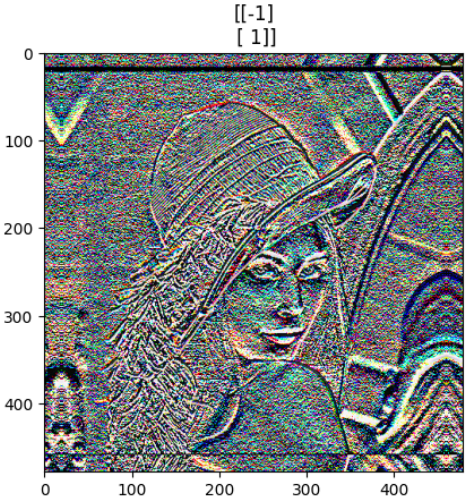
```
for kernel in kernel_list:
    new_img_rgb2 = conv(img_rgb2,kernel,"wrap around")

    new_img_gray2 = conv(img_gray2,kernel,"wrap around")
    plt.title(str(kernel))
    plt.imshow(new_img_rgb2.astype(np.uint8))
    plt.show()
    plt.imshow(new_img_gray2,cmap ="gray")
    plt.show()
```

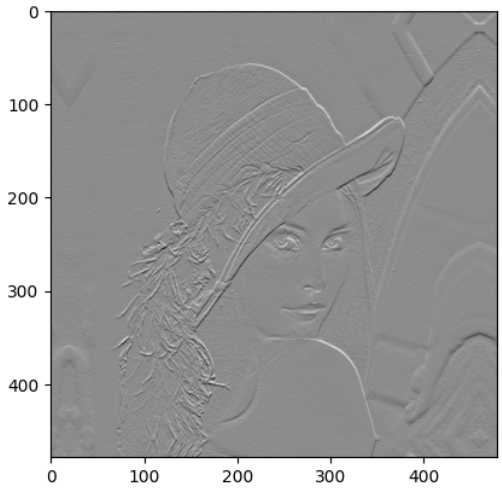

[[0.11111111 0.11111111 0.11111111]
[0.11111111 0.11111111 0.11111111]
[0.11111111 0.11111111 0.11111111]]
0
100
200
300
400
0
100
200
300
400

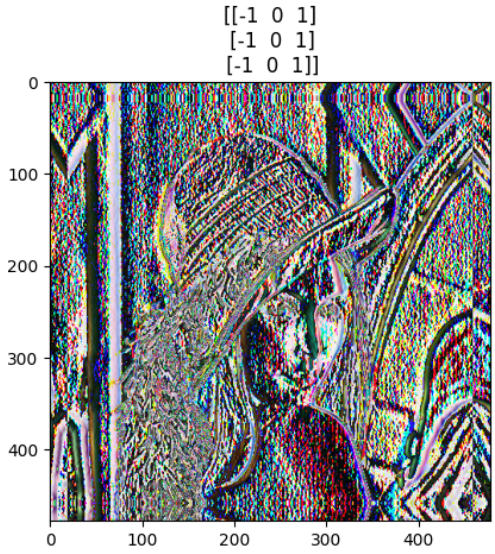

0
100
200
300
400
0
100
200
300
400

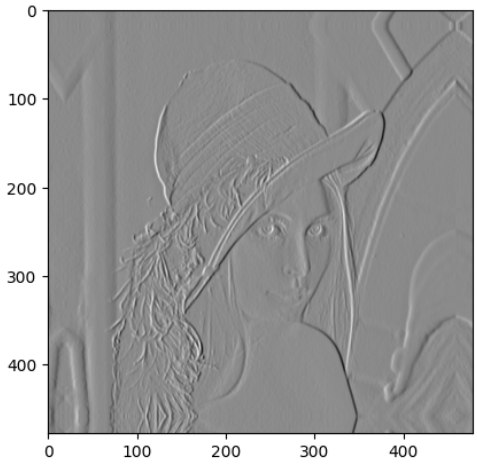

[[-1 1]]
0
100
200
300
400
0
100
200
300
400

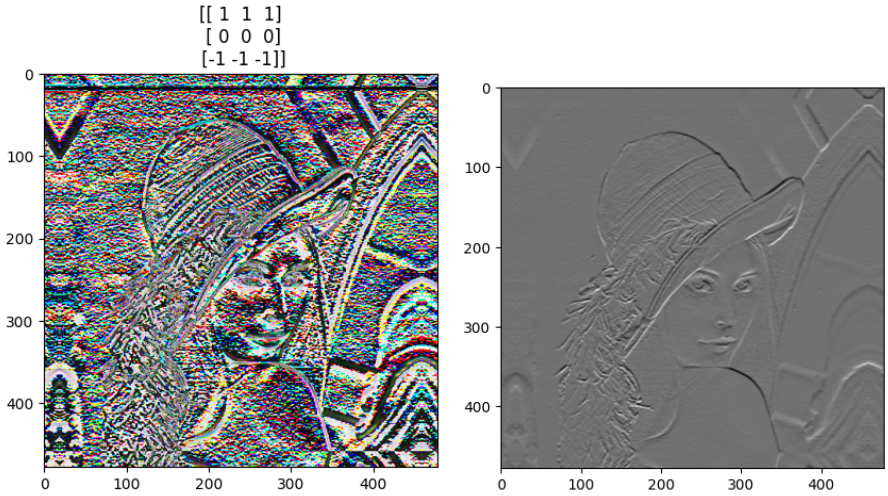

0
100
200
300
400
0
100
200
300
400


[[-1]
[1]]
0
100
200
300
400
0
100
200
300
400

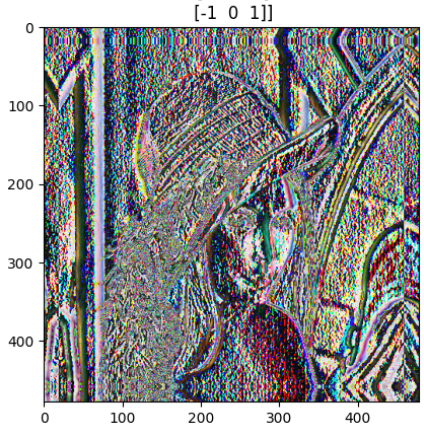

0
100
200
300
400
0
100
200
300
400

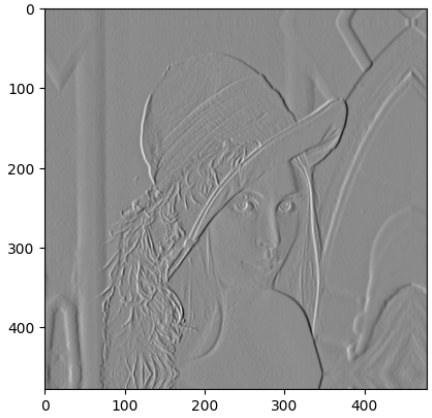

[[-1 0 1]
[-1 0 1]
[-1 0 1]]
0
100
200
300
400
0
100
200
300
400


0
100
200
300
400
0
100
200
300
400

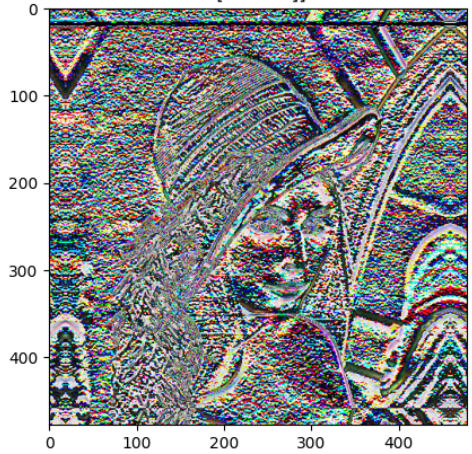

[[1 1 1]
 [0 0 0]
 [-1 -1 -1]]
0
100
200
300
400
0
100
200
300
400
0
100
200
300
400

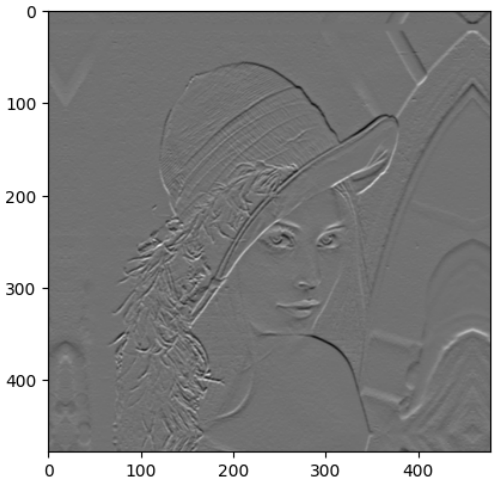
[[-1 0 1]
 [-2 0 2]
 [-1 0 1]]

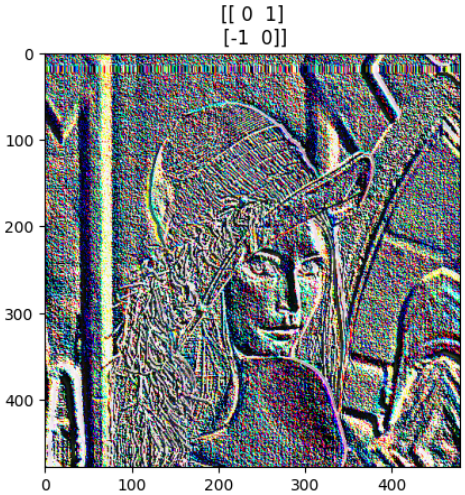


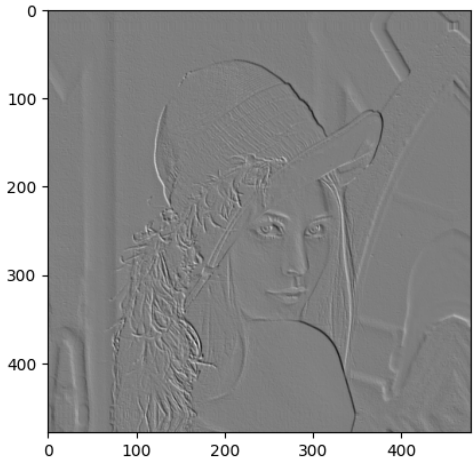


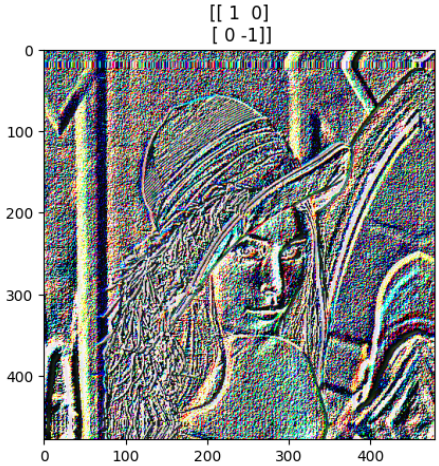
[[1 2 1]
 [0 0 0]
 [-1 -2 -1]]

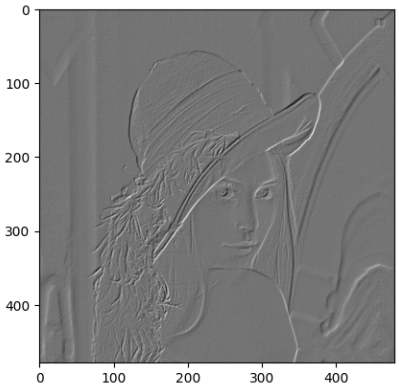



0
100
200
300
400
0
100
200
300
400


[[0 1]
[-1 0]]
0
100
200
300
400
0
100
200
300
400





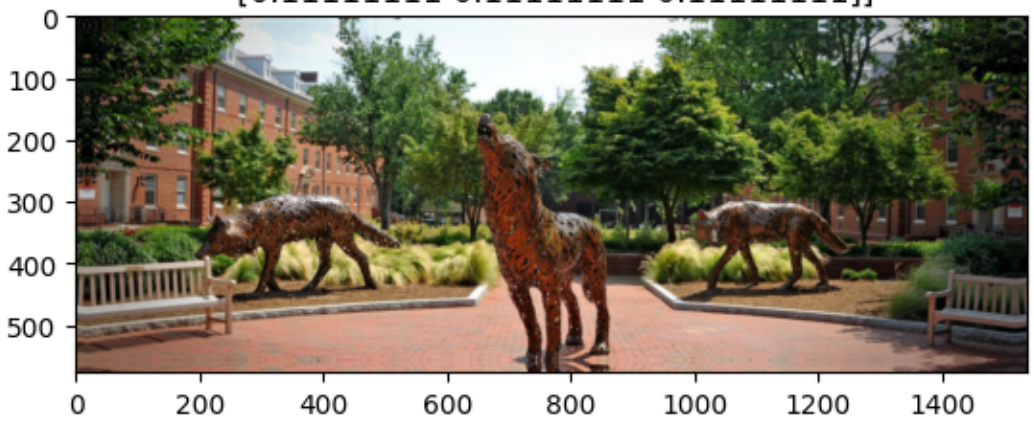


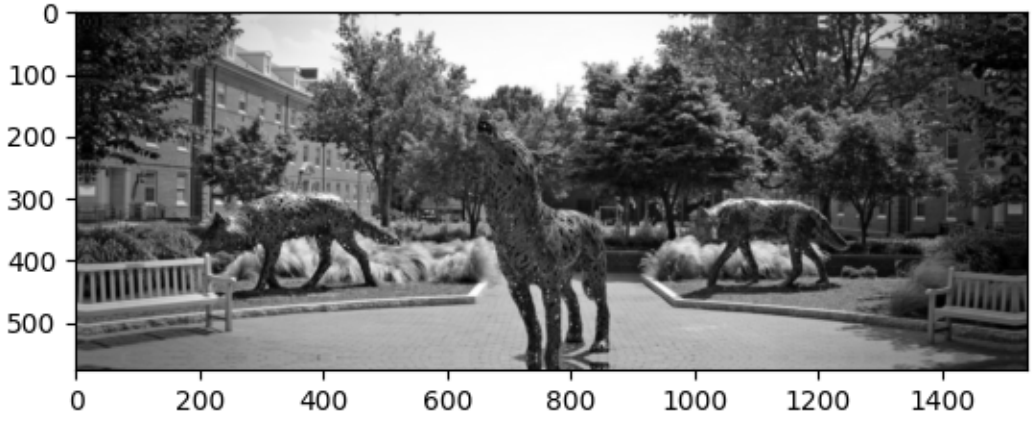
For reflection padding and wolves image:

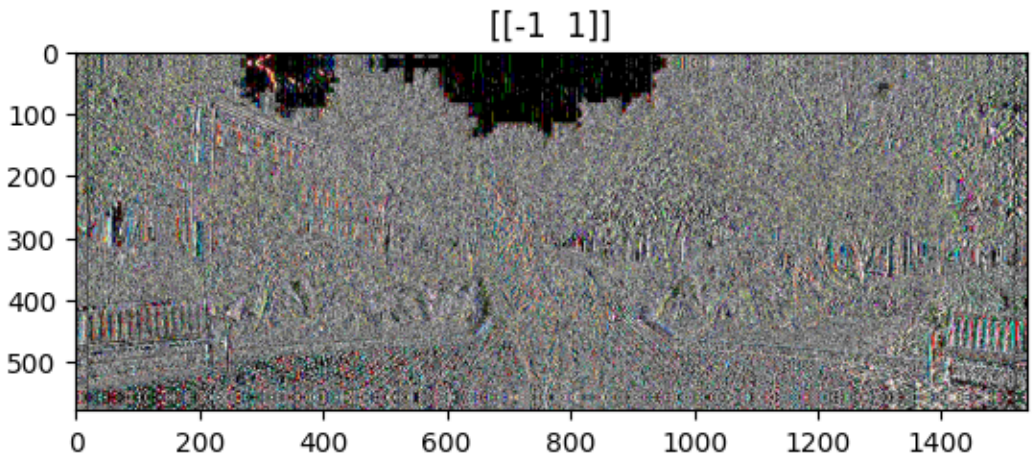
for kernel in kernel_list:

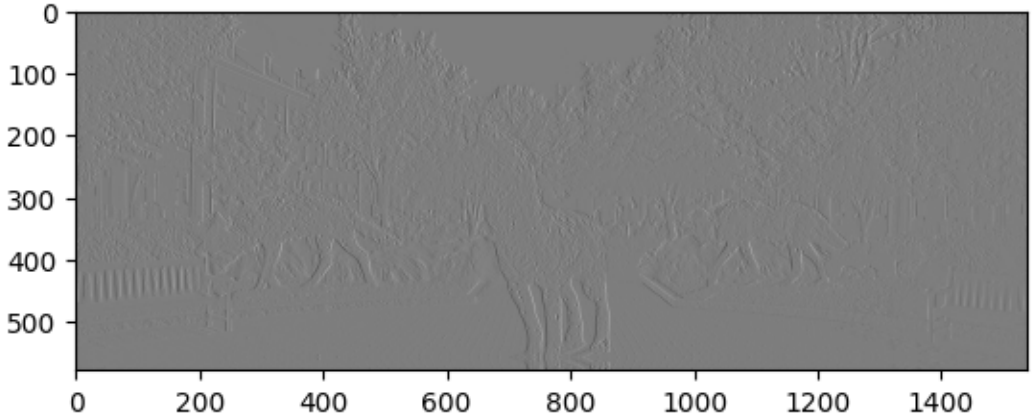
```
new_img_rgb2 = get_convolved_img_shape(img_rgb_padded2, kernel)
new_img_rgb2 = convolution(img_rgb_padded2,new_img_rgb2,kernel)


new_img_gray2 = get_convolved_img_shape(img_gray_padded2, kernel)
new_img_gray2 = convolution(img_gray_padded2,new_img_gray2,kernel)
plt.title(str(kernel))
plt.imshow(new_img_rgb2.astype(np.uint8))
plt.show()
plt.imshow(new_img_gray2,cmap ="gray")
plt.show()
```

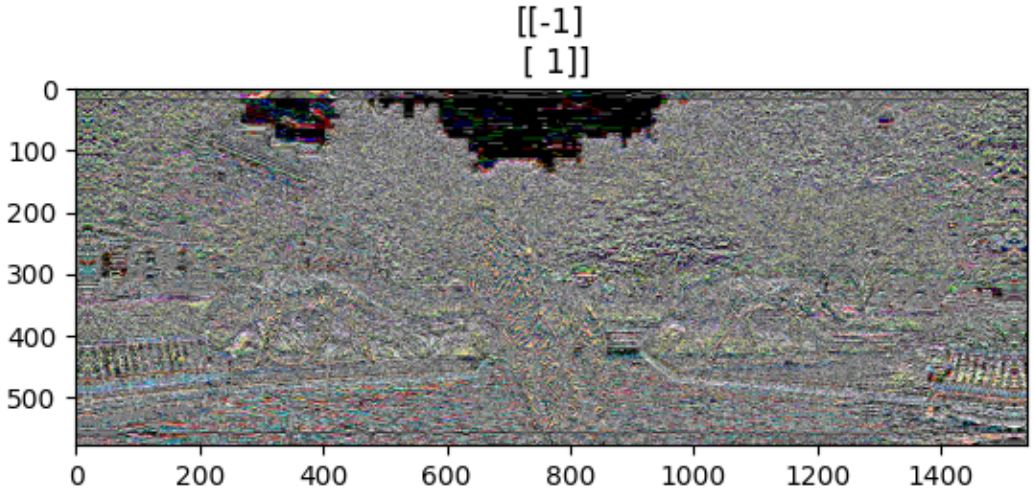
```
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]
```

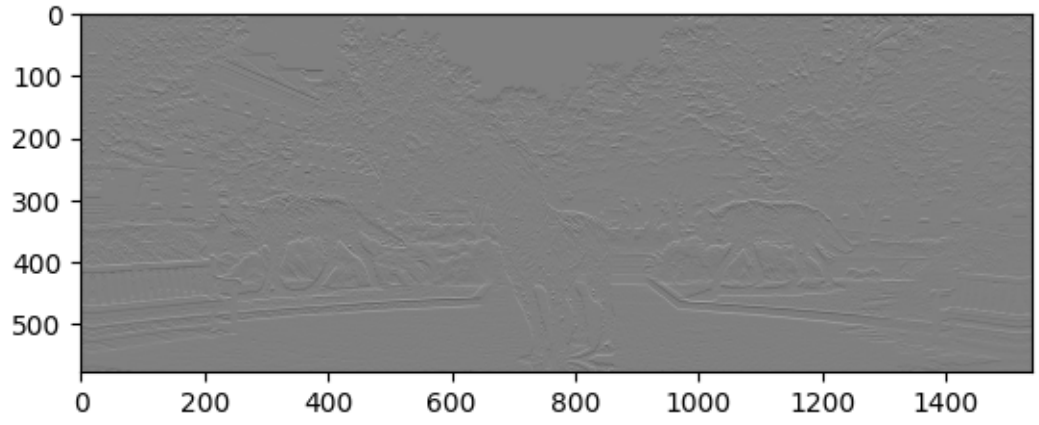




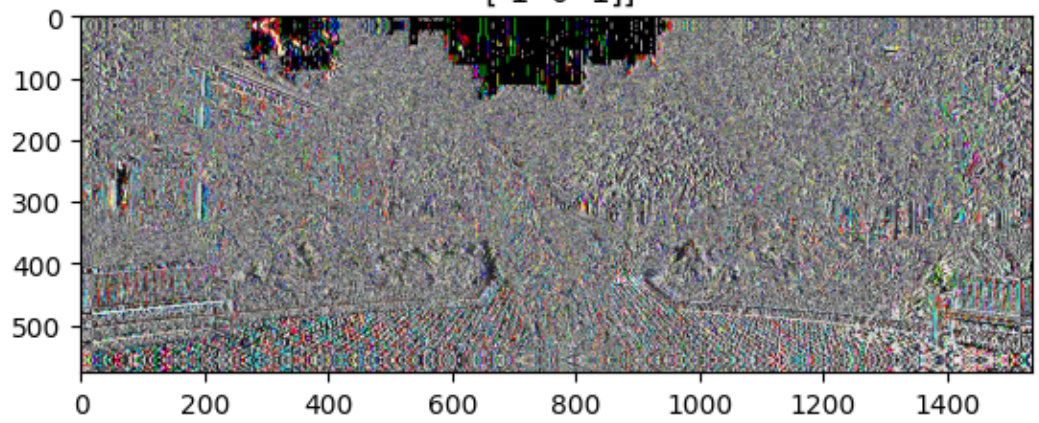

[[-1 1]]
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

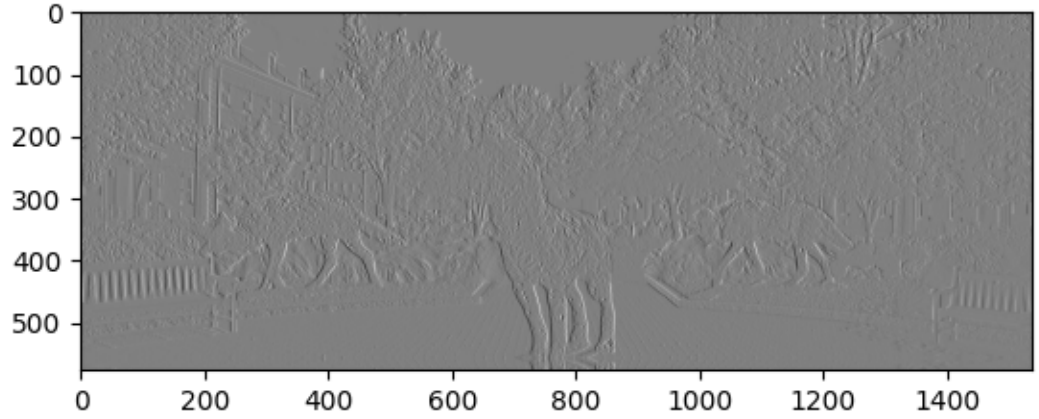

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


[[-1]
[1]]
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

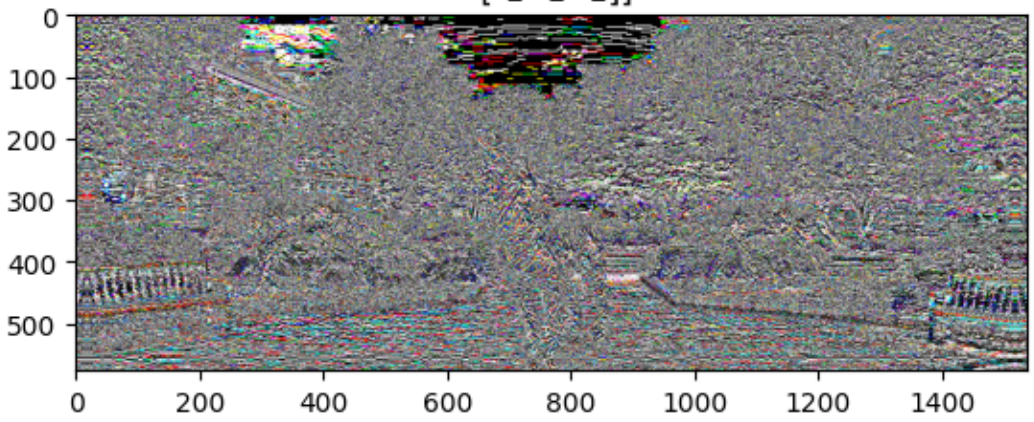

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

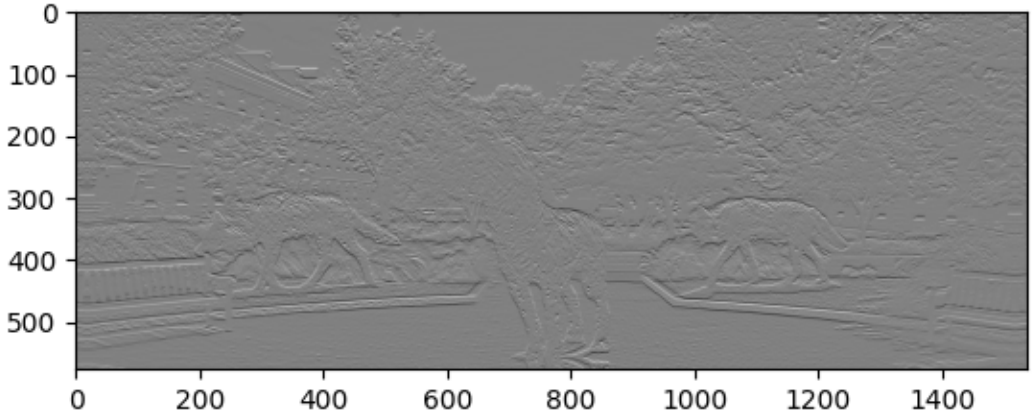
[[-1 0 1]
[-1 0 1]
[-1 0 1]]


0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

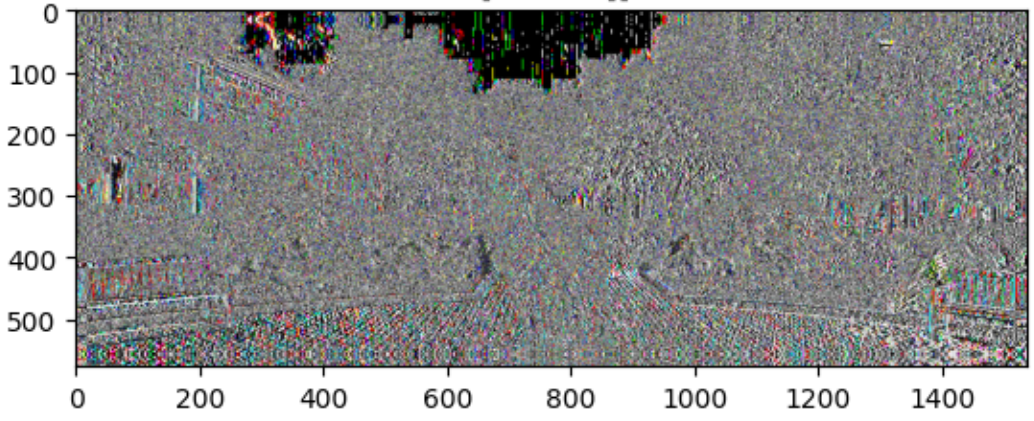

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

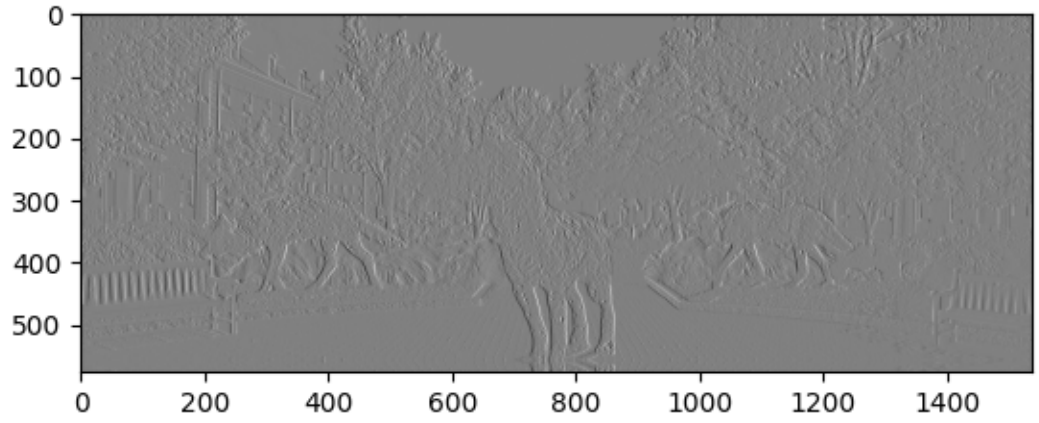
[[1 1 1]
 [0 0 0]
 [-1 -1 -1]]



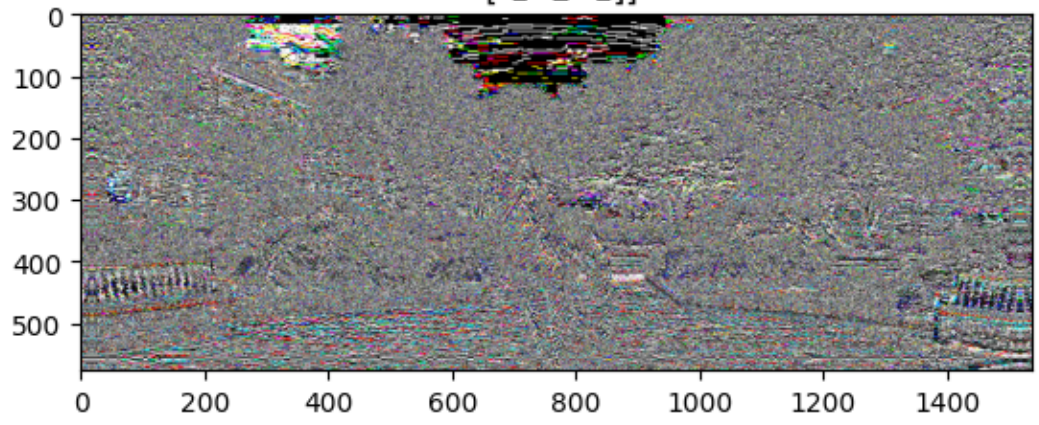


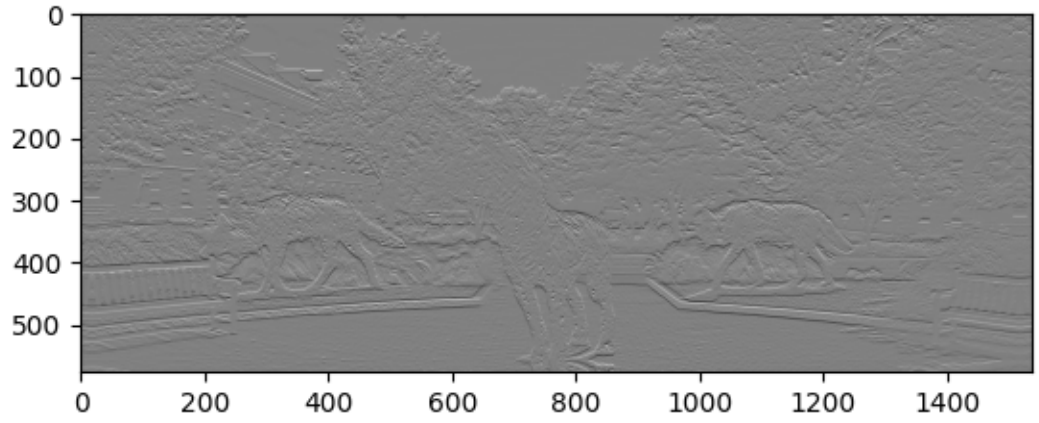
[[-1 0 1]
 [-2 0 2]
 [-1 0 1]]



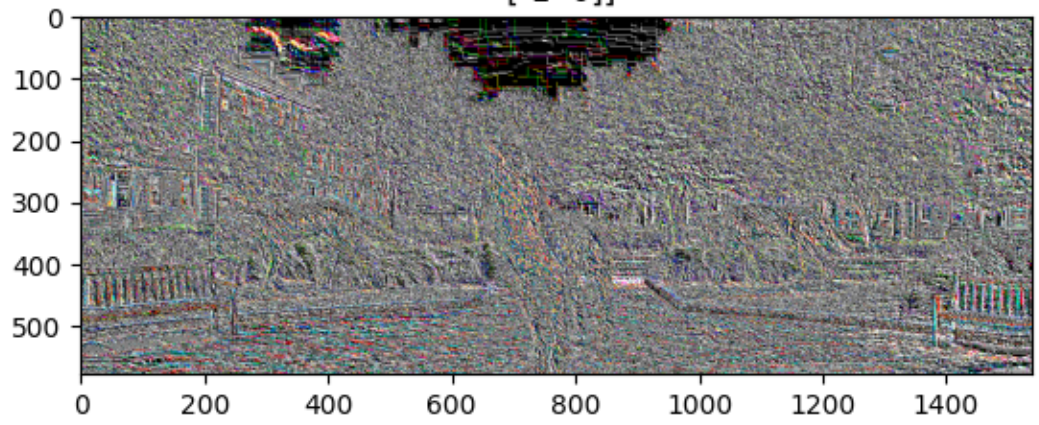

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

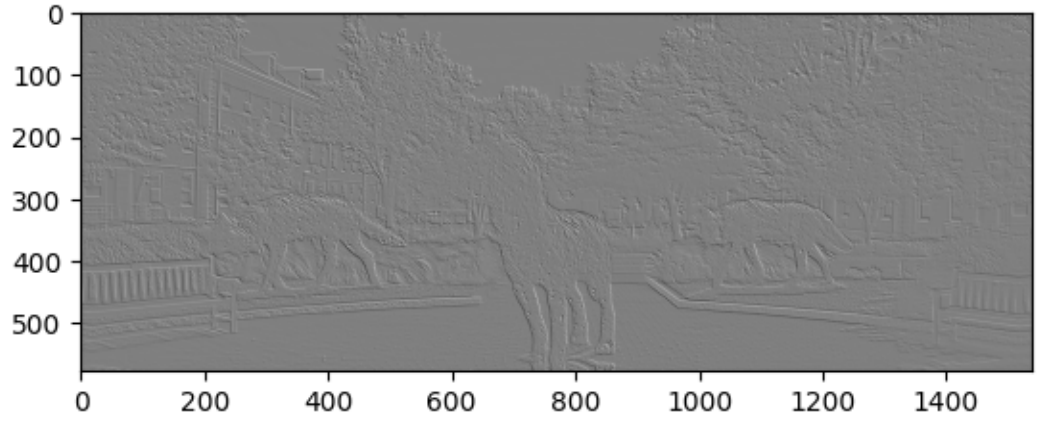
[[1 2 1]
 [0 0 0]
 [-1 -2 -1]]

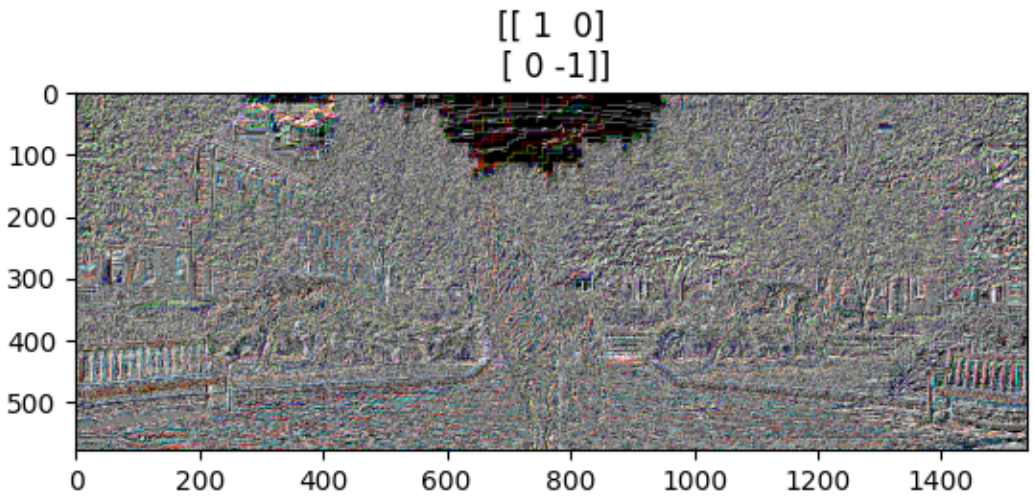

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

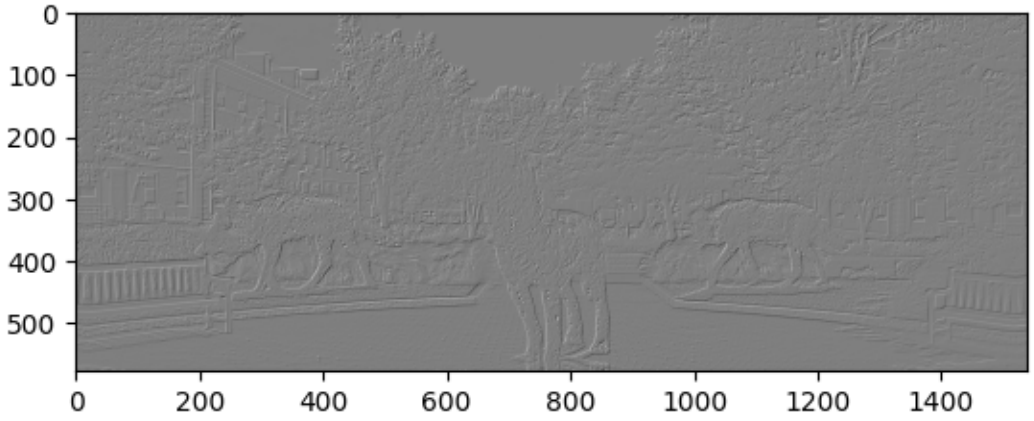

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

[[0 1]
[-1 0]]


0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

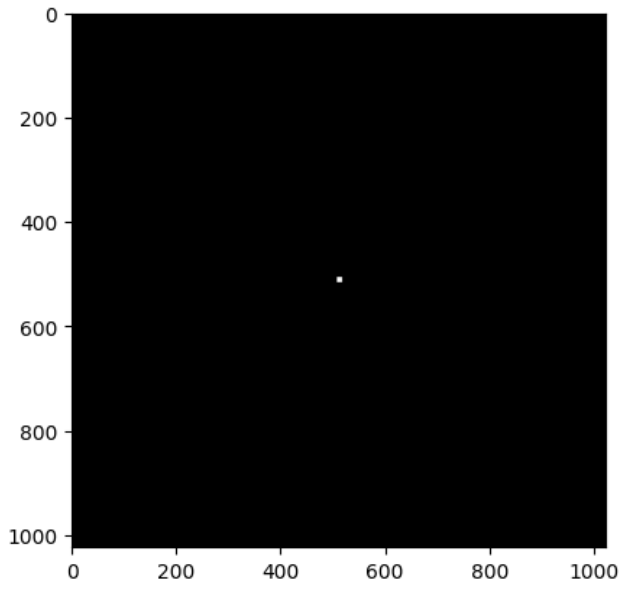

0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400





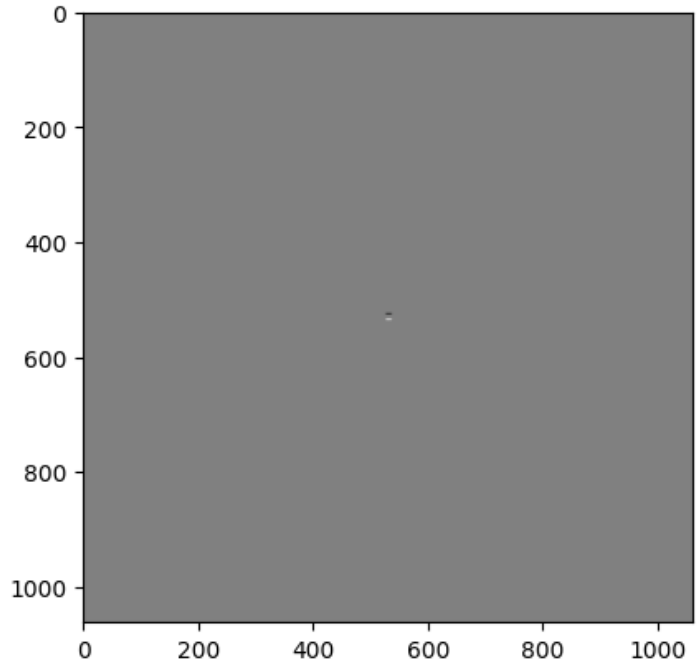
b. Create a grey image of size 1024x1024 pixels that consists of a unit impulse at the center of the image (512, 512) and zeros elsewhere. Use this image and a kernel of your choice (e.g., selected one from (a)) to confirm that your function is indeed performing convolution. Show your filter result and explain why your function is indeed performing convolution.

```
gray_img = np.zeros((1024,1023),np.uint8)
gray_img[512 -5:512 +5,512-5:512+5] = 1
plt.imshow(gray_img,cmap ="gray")
```



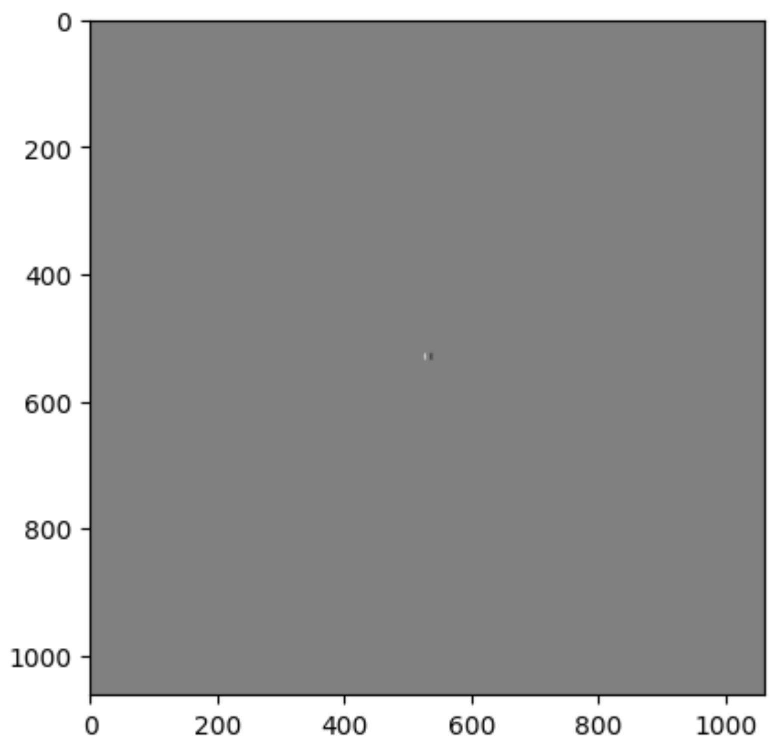
Taking edge detection filter prewitt:

```
gray_unit_impulse = conv(gray_img,prewitt_y,"zero")
plt.imshow(gray_unit_impulse,cmap ="gray")
```



```
gray_unit_impulse = conv(gray_img,prewitt_x,"zero")
plt.imshow(gray_unit_impulse,cmap ="gray")
```

Prewitt filter along the x axis



Thus, from the image we can see only the edges are visible, thus the fuction is performing convolution correctly.

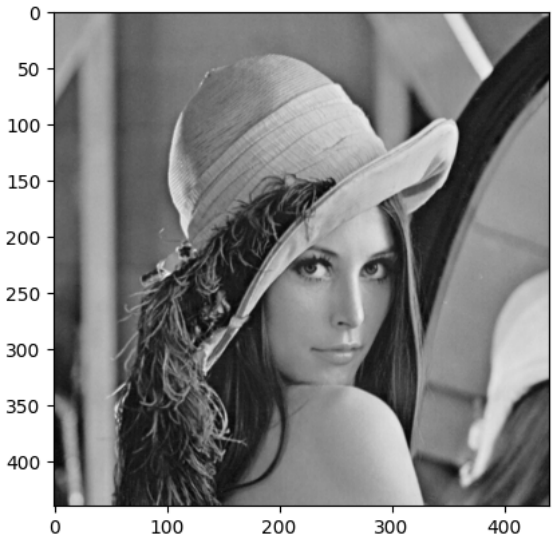
```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1.]
 [ 0.  2.  2.  0.  0.  0.  0.  0.  0.  0.  0. -2. -2.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  3.  3.  0.  0.  0.  0.  0.  0.  0.  0. -3. -3.]
 [ 0.  2.  2.  0.  0.  0.  0.  0.  0.  0.  0. -2. -2.]
 [ 0.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1.]]
```

Problem 2
Implementing and testing the 2-D FFT and its inverse using a built-in 1-D FFT algorithm.

```
import cv2
import numpy as np
from scipy.fft import fft,fft2,ifft,ifft2
import matplotlib.pyplot as plt
```

Normalize the image

```
def normalize_img(img):
    # Subtract minimum value from f
    img_normalized = img - np.min(img)
    img_normalized = img_normalized/ [np.max(img) - np.min(img)]
    return img_normalized
```




```
def fft_2d(img):
    fft_1d = fft(img)
    temp = fft(np.transpose(fft_1d))
    img_fft_2d = np.transpose(temp)
    return img_fft_2d
```

Taking 1d fourier transform across x-axis first and then the y axis by taking transpose of the array

1) Discrete Fourier transform (DFT) of $f(x, y)$

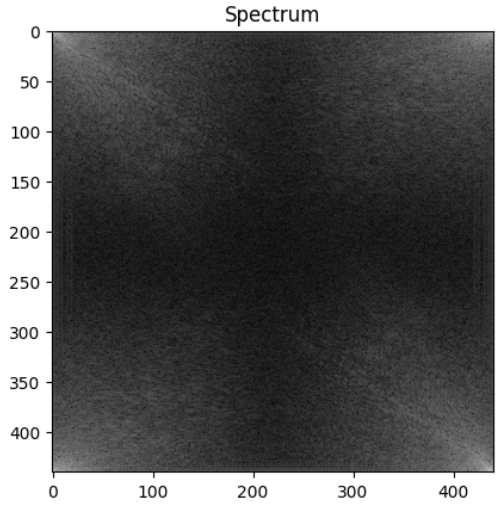
$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M+vy/N)}$$

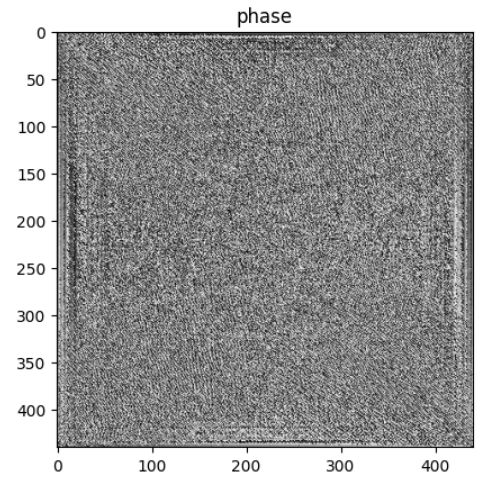
2) Inverse discrete Fourier transform (IDFT) of $F(u, v)$

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) e^{j2\pi(ux/M+vy/N)}$$

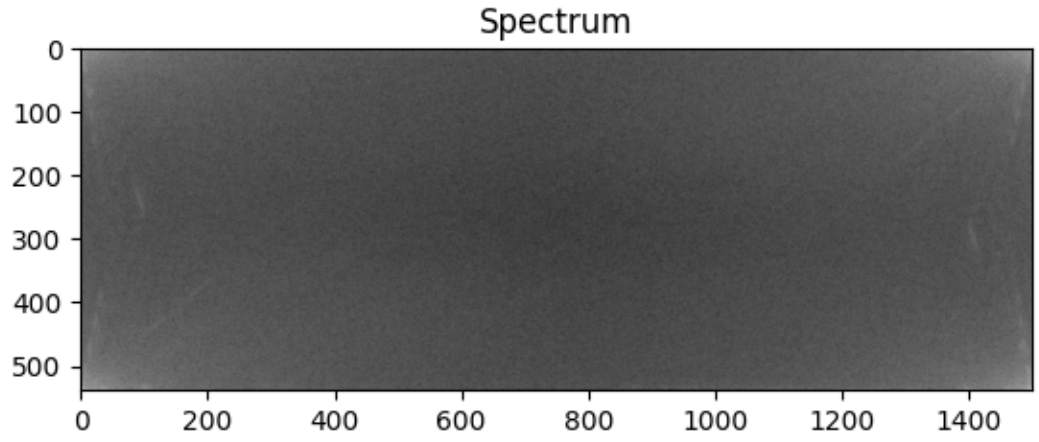
Visualizing phase and spectrum

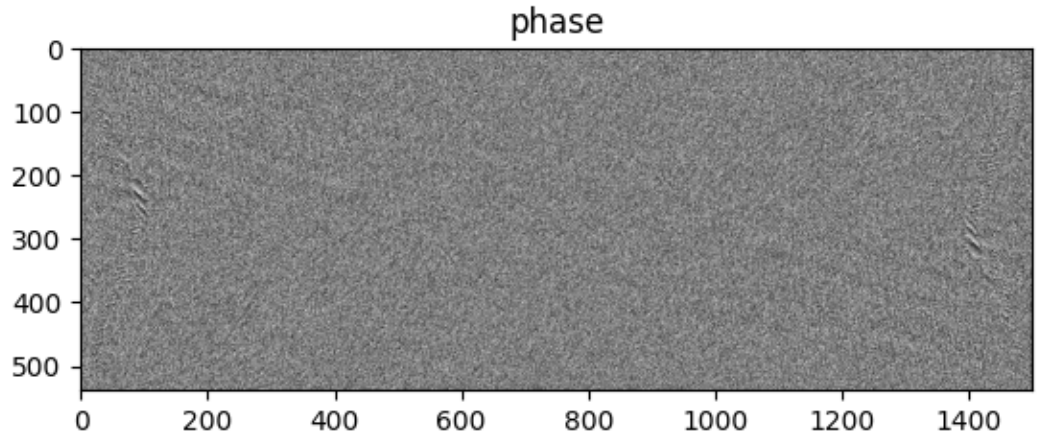
```
spectrum = np.log(1 + np.abs(img_fft_2d))
phase = np.angle(img_fft_2d)

plt.imshow(spectrum, cmap="gray")
plt.title("Spectrum")
plt.show()
plt.imshow(phase, cmap="gray")
plt.title("phase")
plt.show()
```





Taking fft in 2 dimension for wolves.png





3) Polar representation $F(u, v) = |F(u, v)|e^{j\phi(u,v)}$

4) Spectrum $|F(u, v)| = \left[R^2(u, v) + I^2(u, v)\right]^{1/2}$

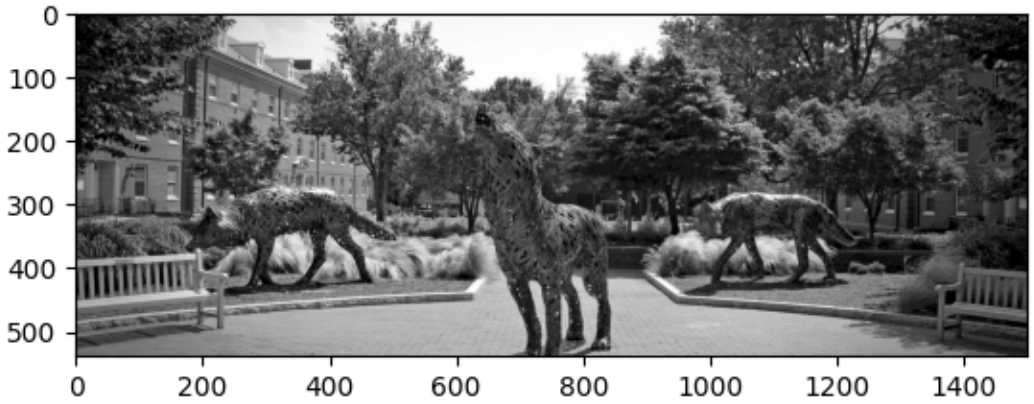
$R = \text{Real}(F); \quad I = \text{Imag}(F)$

5) Phase angle $\phi(u, v) = \tan^{-1}\left[\frac{I(u, v)}{R(u, v)}\right]$

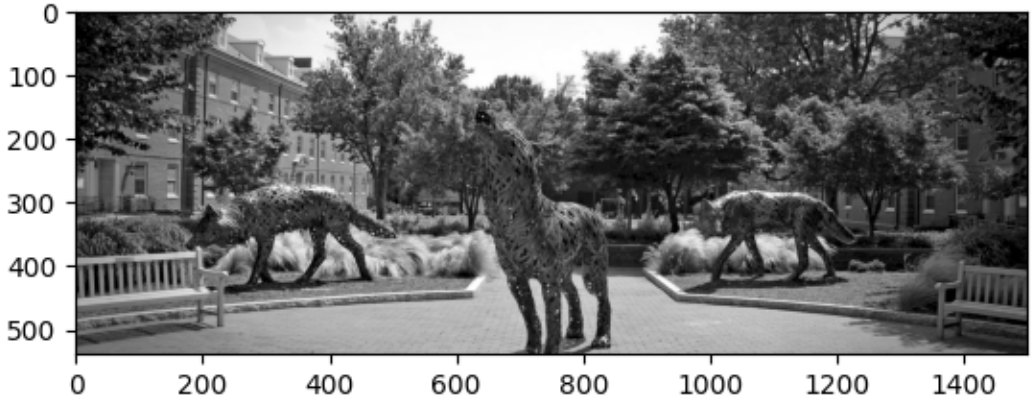
Using your DFT2 to implement the inverse FFT of an input transform F, g = IDFT2(F) from scratch

```
def ifft_2d(img):
    height,width = img.shape
    #taking conjugate
    img = np.conj(img)
    temp2 = fft_2d(img)
    img_ifft_2d = np.array(np.conj(temp2))/(height * width)

    return temp, img_ifft_2d

f= cv2.imread("/Users/tanishakhurana/Desktop/tkhuran3_project02/wolves.png",0)
f_normalized= normalize_img(img2)
plt.imshow(f, cmap ="gray")
plt.show()
plt.imshow(f_normalized, cmap ="gray")
plt.show()
f_normalized.shape
```

Normalized input image:



IFFT2d absolute image

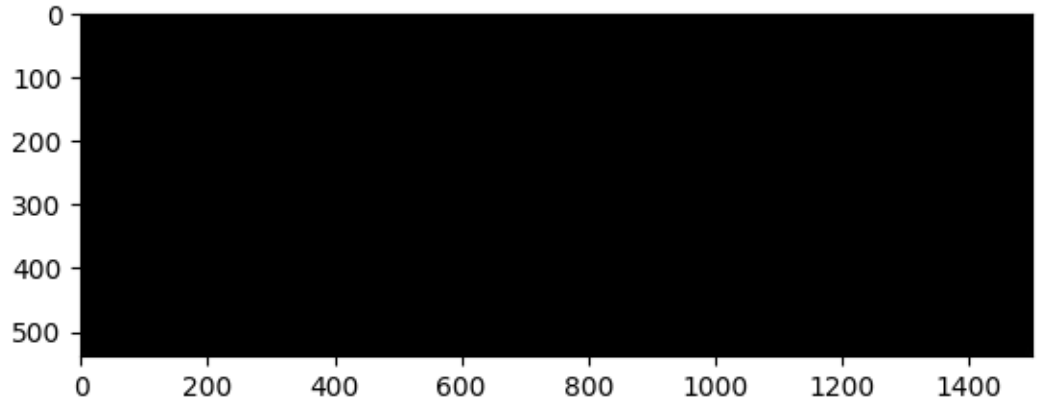


Test your function for the two results in (a). Given an input grey image f you use (a) DFT2 to compute its F and use your IDFT2 to compute g from F. Visualize g and F and they should look identical. To double-check it, visualize d = g − F, which should be a black image.

Subtracting g and F to get d:

F = fft_2d(f_normalized)
_,g = ifft_2d(F)

d = g -F



We get a black image