Khoa Do (200388335)
ECE 558-01

October 19, 2021
Dr. Tianfu Wu

Project 1 – Report

Problem 1

a)

This part is to implement a two-dimensional convolution from scratch knowing g = conv2(f,w,pad) with g is the filtered image after applying the convolution conv2, f is the original image, w is the 3x3 kernel (filter) of choice, and pad is the padding choice including zero padding, wrap around, copy edge, and reflect across edges. The padding choice is implemented from scratch instead of using built-in numpy functions.

The choices for the filter are as follow:

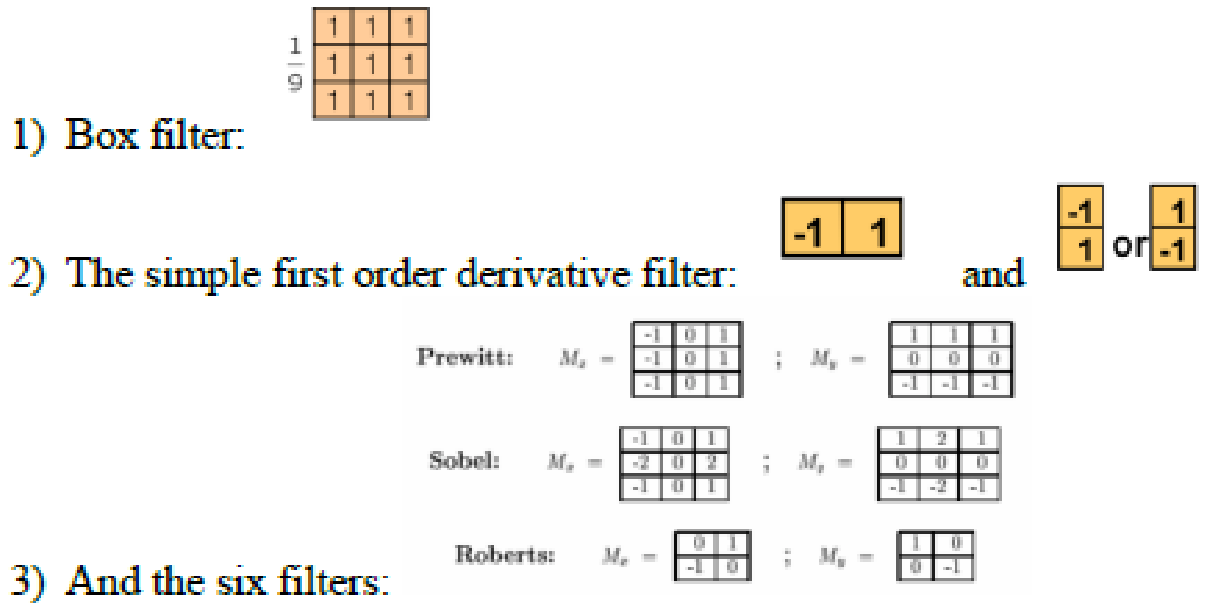
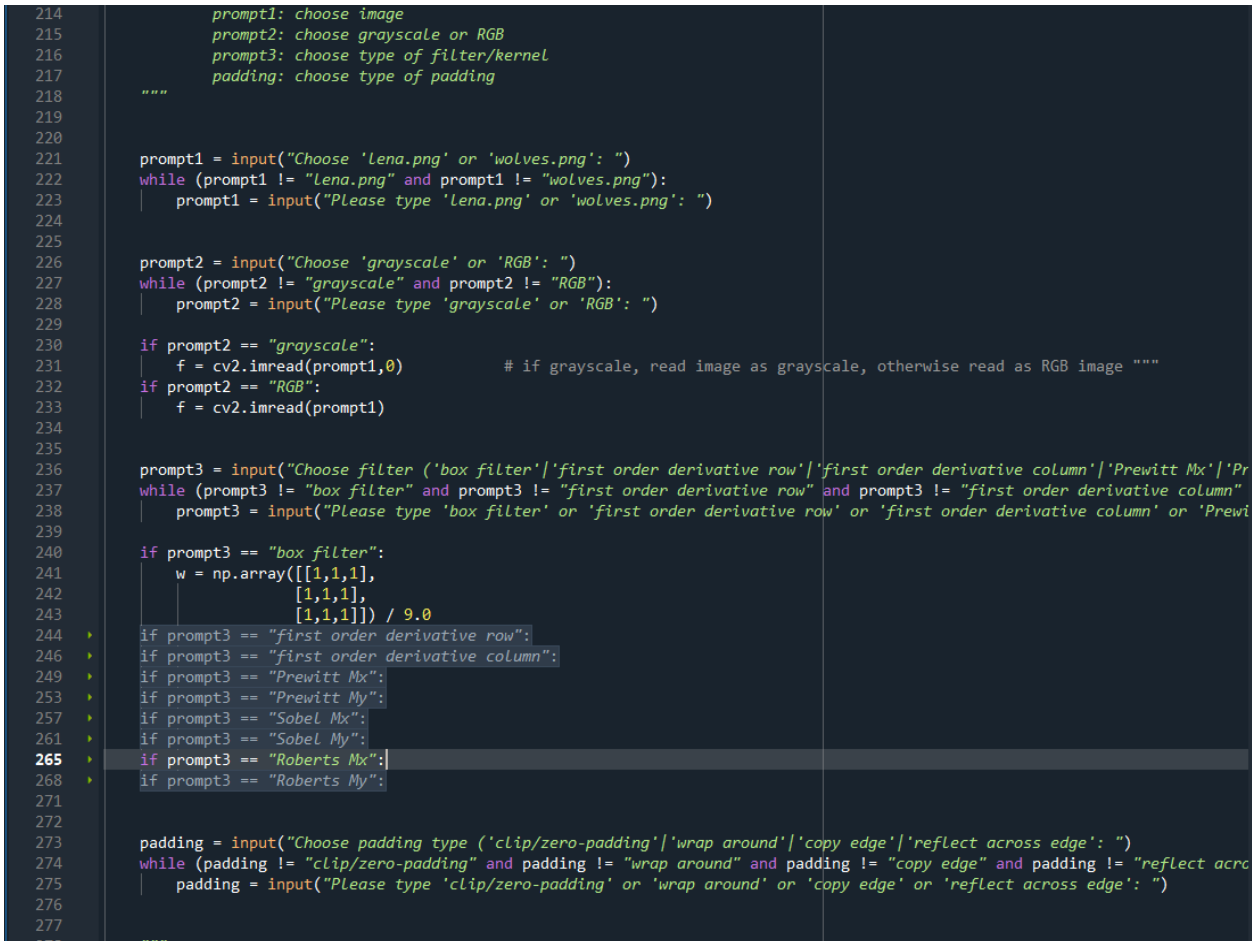

1) Box filter:
$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$
2) The simple first order derivative filter:
$\begin{bmatrix}-1&1\end{bmatrix}$
and
$\begin{bmatrix}-1\\1\end{bmatrix}$ or $\begin{bmatrix}1\\-1\end{bmatrix}$
3) And the six filters:
Prewitt: $M_x = \begin{bmatrix}-1&0&1\\-1&0&1\\-1&0&1\end{bmatrix}$; $M_y = \begin{bmatrix}1&1&1\\0&0&0\\-1&-1&-1\end{bmatrix}$
Sobel: $M_x = \begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}$; $M_y = \begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix}$
Roberts: $M_x = \begin{bmatrix}0&1\\-1&0\end{bmatrix}$; $M_y = \begin{bmatrix}1&0\\0&-1\end{bmatrix}$

Figure 1. Types of Filters.

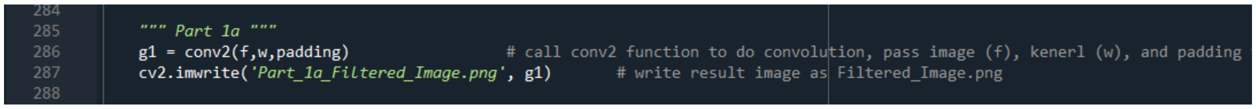
The code consists of three main functions: main function, convolution function, and padding function. The main function displays several prompts asking the user to choose the image to test, whether the test image to be processed as grayscale or RGB, choose a type of filter, and choose a type of padding. The prompts also make sure that the user type in the exact given options.



```
            prompt1: choose image
            prompt2: choose grayscale or RGB
            prompt3: choose type of filter/kernel
            padding: choose type of padding
    """


    prompt1 = input("Choose 'lena.png' or 'wolves.png': ")
    while (prompt1 != "lena.png" and prompt1 != "wolves.png"):
        prompt1 = input("Please type 'lena.png' or 'wolves.png': ")


    prompt2 = input("Choose 'grayscale' or 'RGB': ")
    while (prompt2 != "grayscale" and prompt2 != "RGB"):
        prompt2 = input("Please type 'grayscale' or 'RGB': ")

    if prompt2 == "grayscale":
        f = cv2.imread(prompt1,0)           # if grayscale, read image as grayscale, otherwise read as RGB image """
    if prompt2 == "RGB":
        f = cv2.imread(prompt1)


    prompt3 = input("Choose filter ('box filter'|'first order derivative row'|'first order derivative column'|'Prewitt Mx'|'Pr
    while (prompt3 != "box filter" and prompt3 != "first order derivative row" and prompt3 != "first order derivative column"
        prompt3 = input("Please type 'box filter' or 'first order derivative row' or 'first order derivative column' or 'Prewi

    if prompt3 == "box filter":
        w = np.array([[1,1,1],
                      [1,1,1],
                      [1,1,1]]) / 9.0
    if prompt3 == "first order derivative row":
    if prompt3 == "first order derivative column":
    if prompt3 == "Prewitt Mx":
    if prompt3 == "Prewitt My":
    if prompt3 == "Sobel Mx":
    if prompt3 == "Sobel My":
    if prompt3 == "Roberts Mx":
    if prompt3 == "Roberts My":


    padding = input("Choose padding type ('clip/zero-padding'|'wrap around'|'copy edge'|'reflect across edge': ")
    while (padding != "clip/zero-padding" and padding != "wrap around" and padding != "copy edge" and padding != "reflect acrc
        padding = input("Please type 'clip/zero-padding' or 'wrap around' or 'copy edge' or 'reflect across edge': ")
```

Figure 2. Code Snip 1.

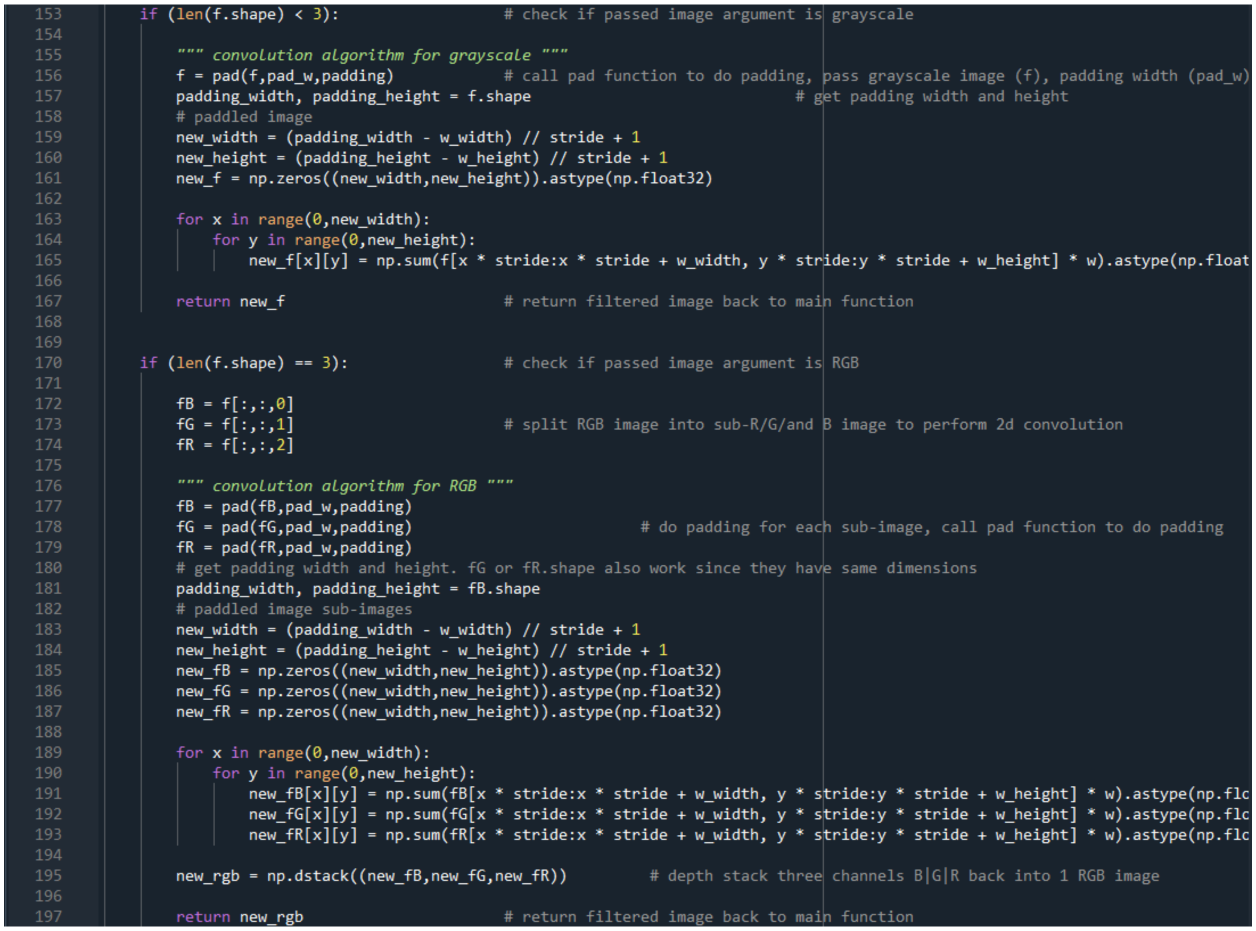
Once choices are made for f, w, and padding, they are passed to the conv2 function. The conv2 function then returns the filtered image as g to be displayed.



```
    """ Part 1a """
    g1 = conv2(f,w,padding)                 # call conv2 function to do convolution, pass image (f), kenerl (w), and padding
    cv2.imwrite('Part_1a_Filtered_Image.png', g1)       # write result image as Filtered_Image.png
```

Figure 3. Code Snip 2.

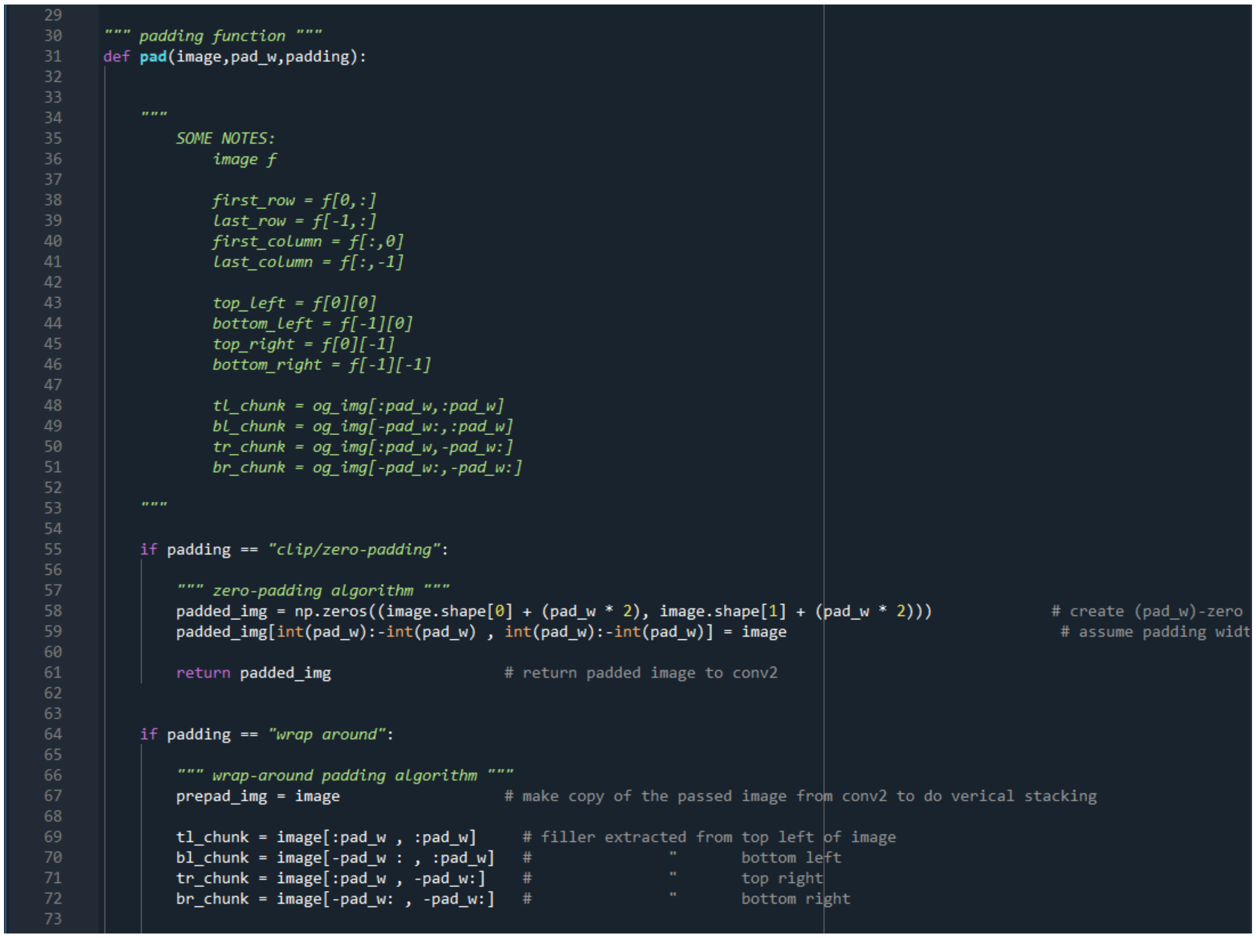
The conv2 function takes the passed parameters from the main function and begins processing. For this project, the padding width can be hard coded as any number and the stride is assumed to be one. However, I give the user an option to input their desired padding width and stride in conv2 since the called conv2 in main function only takes f, w, and padding. Conv2 first checks whether the input image is grayscale or RGB. The convolution algorithms for these two types of image are the same except RGB is split in to red, green, and blue channels. The three channels are then convoluted separately before being stacked into one RGB image and returned to main function.



```python
    if (len(f.shape) < 3):                  # check if passed image argument is grayscale

        """ convolution algorithm for grayscale """
        f = pad(f,pad_w,padding)            # call pad function to do padding, pass grayscale image (f), padding width (pad_w)
        padding_width, padding_height = f.shape                                     # get padding width and height
        # paddled image
        new_width = (padding_width - w_width) // stride + 1
        new_height = (padding_height - w_height) // stride + 1
        new_f = np.zeros((new_width,new_height)).astype(np.float32)

        for x in range(0,new_width):
            for y in range(0,new_height):
                new_f[x][y] = np.sum(f[x * stride:x * stride + w_width, y * stride:y * stride + w_height] * w).astype(np.float

        return new_f                        # return filtered image back to main function


    if (len(f.shape) == 3):                 # check if passed image argument is RGB

        fB = f[:,:,0]
        fG = f[:,:,1]                       # split RGB image into sub-R/G/and B image to perform 2d convolution
        fR = f[:,:,2]

        """ convolution algorithm for RGB """
        fB = pad(fB,pad_w,padding)
        fG = pad(fG,pad_w,padding)                                 # do padding for each sub-image, call pad function to do padding
        fR = pad(fR,pad_w,padding)
        # get padding width and height. fG or fR.shape also work since they have same dimensions
        padding_width, padding_height = fB.shape
        # paddled image sub-images
        new_width = (padding_width - w_width) // stride + 1
        new_height = (padding_height - w_height) // stride + 1
        new_fB = np.zeros((new_width,new_height)).astype(np.float32)
        new_fG = np.zeros((new_width,new_height)).astype(np.float32)
        new_fR = np.zeros((new_width,new_height)).astype(np.float32)

        for x in range(0,new_width):
            for y in range(0,new_height):
                new_fB[x][y] = np.sum(fB[x * stride:x * stride + w_width, y * stride:y * stride + w_height] * w).astype(np.flo
                new_fG[x][y] = np.sum(fG[x * stride:x * stride + w_width, y * stride:y * stride + w_height] * w).astype(np.flo
                new_fR[x][y] = np.sum(fR[x * stride:x * stride + w_width, y * stride:y * stride + w_height] * w).astype(np.flo

        new_rgb = np.dstack((new_fB,new_fG,new_fR))                 # depth stack three channels B|G|R back into 1 RGB image

        return new_rgb                      # return filtered image back to main function
```

Figure 4. Code Snip 3.

Within the conv2 function, the pad function is called to process the type of padding the user has chosen.



```

""" padding function """
def pad(image,pad_w,padding):


    """
        SOME NOTES:
            image f

            first_row = f[0,:]
            last_row = f[-1,:]
            first_column = f[:,0]
            last_column = f[:,-1]

            top_left = f[0][0]
            bottom_left = f[-1][0]
            top_right = f[0][-1]
            bottom_right = f[-1][-1]

            tl_chunk = og_img[:pad_w,:pad_w]
            bl_chunk = og_img[-pad_w:,:pad_w]
            tr_chunk = og_img[:pad_w,-pad_w:]
            br_chunk = og_img[-pad_w:,-pad_w:]

    """

    if padding == "clip/zero-padding":

        """ zero-padding algorithm """
        padded_img = np.zeros((image.shape[0] + (pad_w * 2), image.shape[1] + (pad_w * 2)))             # create (pad_w)-zero
        padded_img[int(pad_w):-int(pad_w) , int(pad_w):-int(pad_w)] = image                              # assume padding widt

        return padded_img                   # return padded image to conv2


    if padding == "wrap around":

        """ wrap-around padding algorithm """
        prepad_img = image                  # make copy of the passed image from conv2 to do verical stacking

        tl_chunk = image[:pad_w , :pad_w]     # filler extracted from top left of image
        bl_chunk = image[-pad_w : , :pad_w]   #                "       bottom left
        tr_chunk = image[:pad_w , -pad_w:]    #                "       top right
        br_chunk = image[-pad_w: , -pad_w:]   #                "       bottom right

```

Figure 5. Code Snip 4.

For zero padding, arrays of zeros are created to wrap around the original image depending on the padding width. For wrap-around and copy-edge padding, fillers are extracted from top left, bottom left, top right, and bottom right of the original image. The fillers are then stacked according to wrap-around and copy-edge logics using built-in numpy functions vstack and hstack. For reflect-across-edge padding, fillers are stacked vertically to the original image according to the reflect-across-edge padding logic, which becomes a new image. Then using the same logics, fillers are stack horizontally to the new image to produce the final padded image and return it to conv2.

Upon completion, some filtered images are shown below.

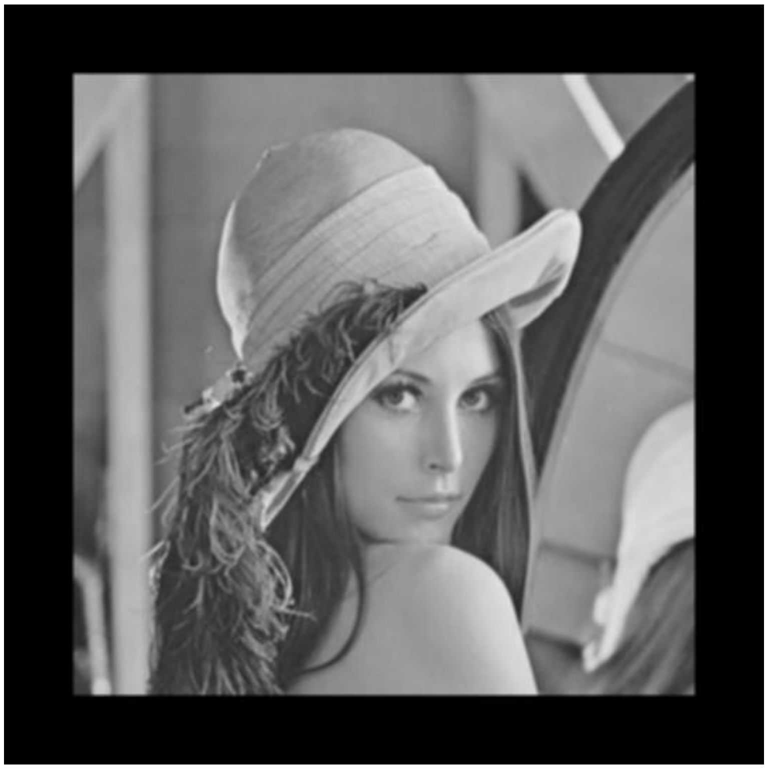


Figure 6. "Lena.png," Grayscale, Box Filter, Zero-Padding, Padding Width = 50, and Stride Count = 1.

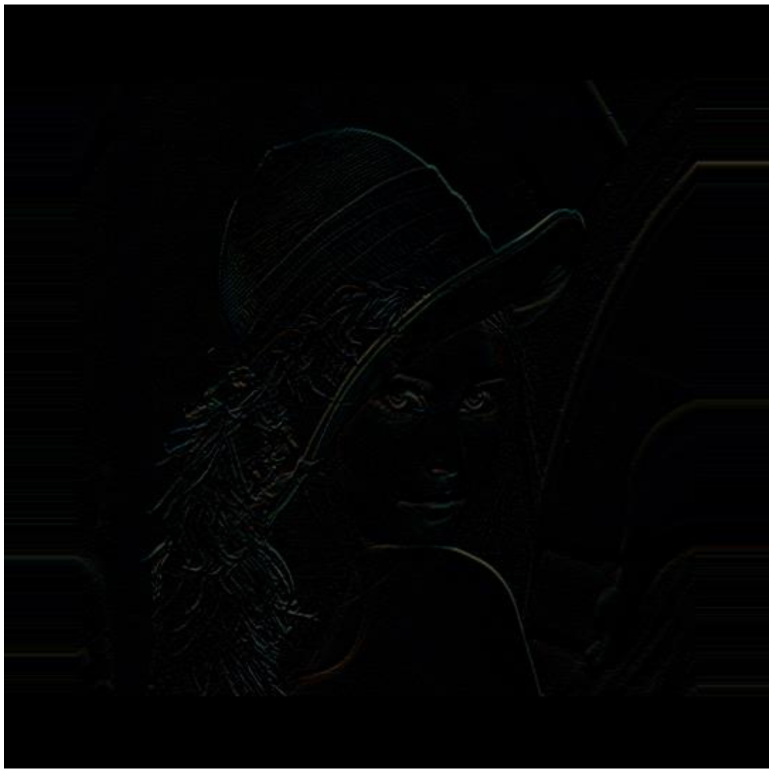


Figure 7. "Lena.png," RGB, First Order Derivative Column, Copy-Edge, Padding Width = 50, and Stride Count = 1.

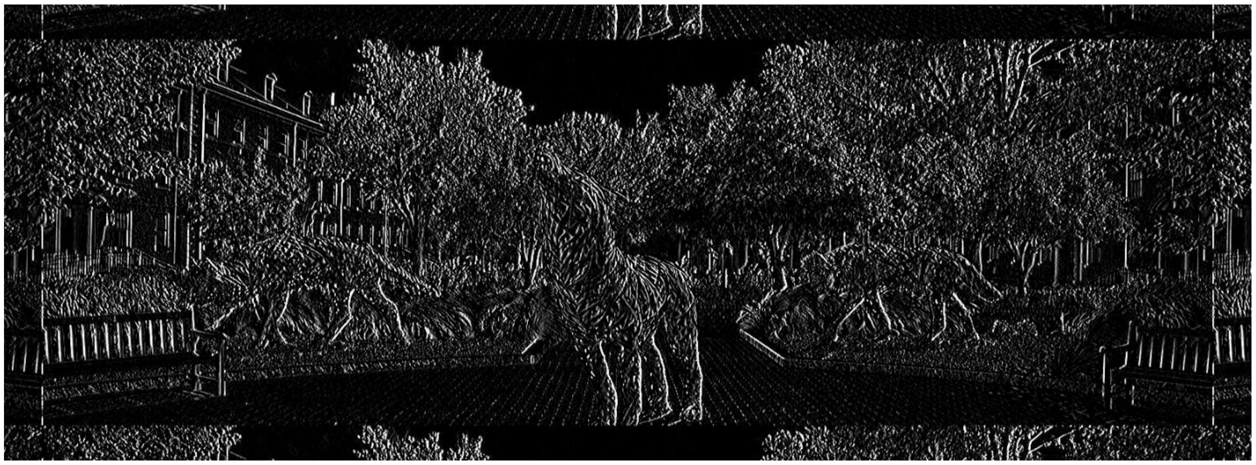


Figure 8. "Wolves.png," Grayscale, Sobel Mx, Wrap-Around, Padding Width = 50, and Stride Count = 1.

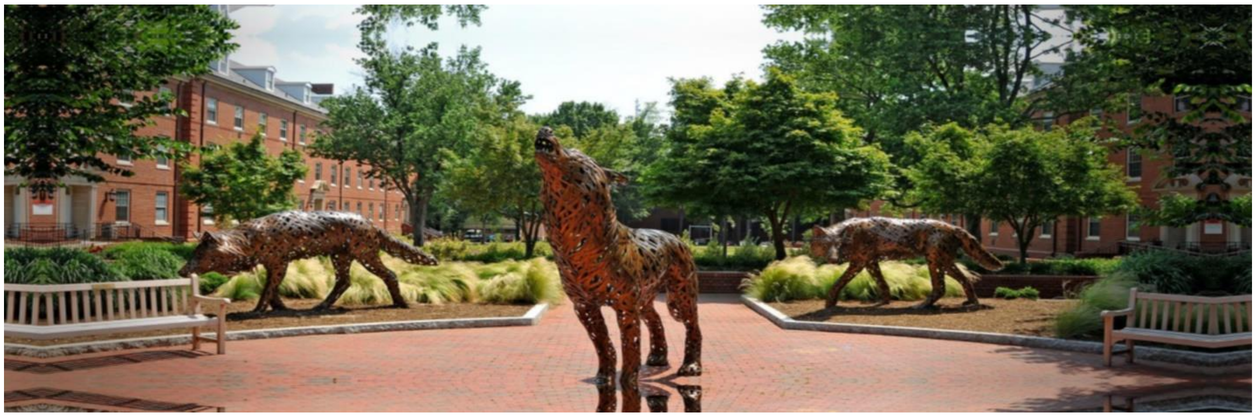
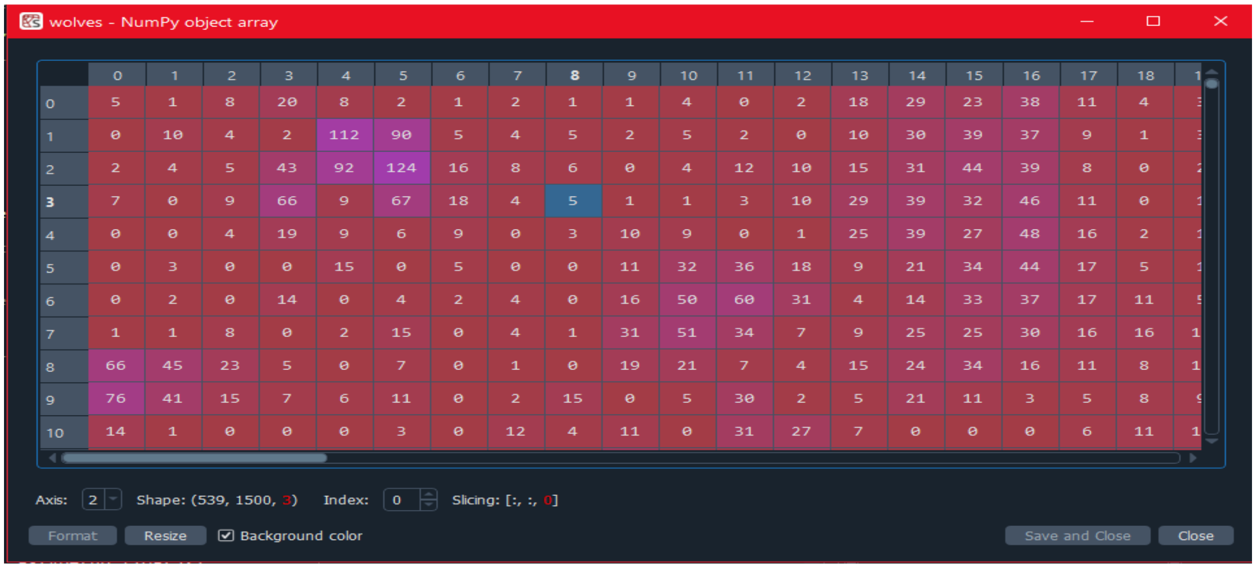


Figure 9. "Wolves.png," RGB, Box Filter, Reflect-Across-Edge, Padding Width = 50, and Stride Count = 1.



wolves - NumPy object array

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	5	1	8	20	8	2	1	2	1	1	4	0	2	18	29	23	38	11	4
1	0	10	4	2	112	90	5	4	5	2	5	2	0	10	30	39	37	9	1
2	2	4	5	43	92	124	16	8	6	0	4	12	10	15	31	44	39	8	0
3	7	0	9	66	9	67	18	4	5	1	1	3	10	29	39	32	46	11	0
4	0	0	4	19	9	6	9	0	3	10	9	0	1	25	39	27	48	16	2
5	0	3	0	0	15	0	5	0	0	11	32	36	18	9	21	34	44	17	5
6	0	2	0	14	0	4	2	4	0	16	50	60	31	4	14	33	37	17	11
7	1	1	8	0	2	15	0	4	1	31	51	34	7	9	25	25	30	16	16
8	66	45	23	5	0	7	0	1	0	19	21	7	4	15	24	34	16	11	8
9	76	41	15	7	6	11	0	2	15	0	5	30	2	5	21	11	3	5	8
10	14	1	0	0	0	3	0	12	4	11	0	31	27	7	0	0	0	6	11

Axis: 2 Shape: (539, 1500, 3) Index: 0 Slicing: [:, :, 0]

Figure 10. "Wolves.png," RGB, Box Filter, Reflect-Across-Edge, Padding Width = 50, and Stride Count = 1 as Matrix values.

Figure 10 show the image obtained in figure 9 imported into Spyder IDE for inspection. The matrix values shown in figure 10 and the matrix values obtained when using a built-in numpy function are the same.

b) This part is to verify that the implemented conv2 works by creating a 1024x1024 image which is completely black except for one center pixel located at (512,512) is white (value = 255) and convolute it with a kernel of choice. Upon completion, the following figures are obtained.

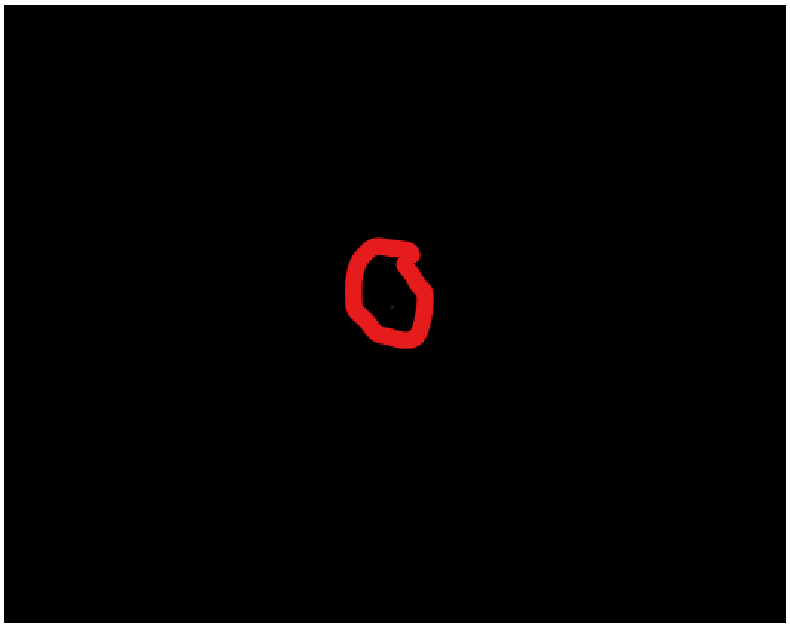
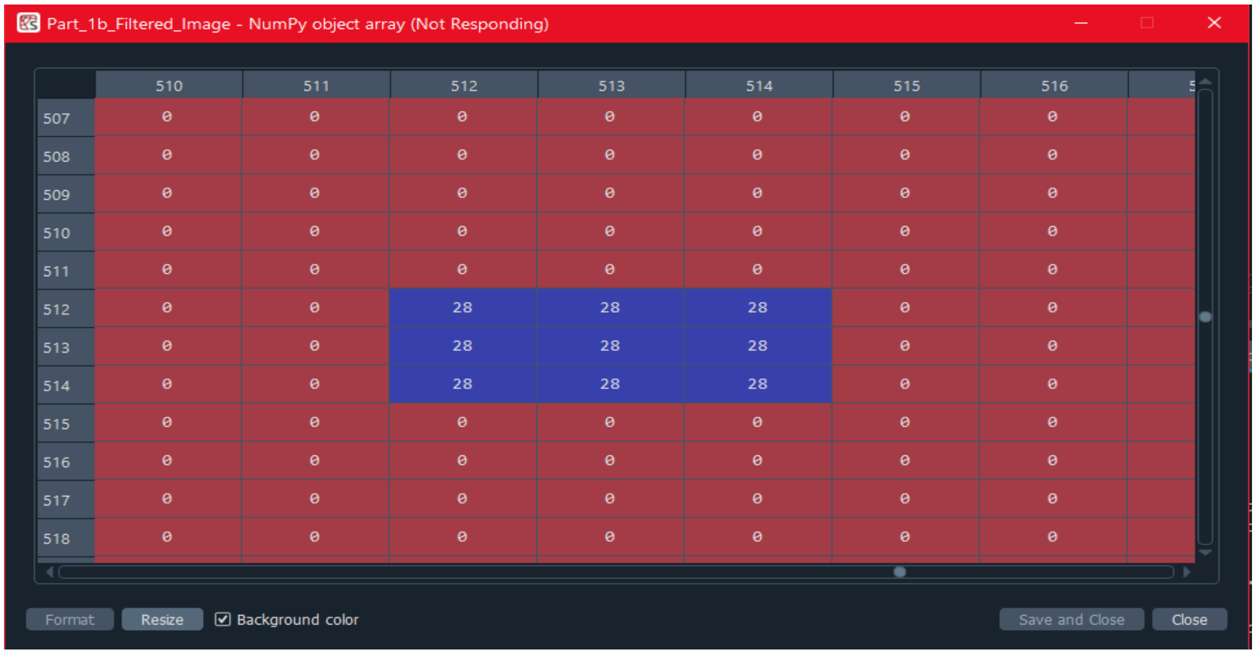


Figure 11. "Part_1b_Filtered_Image.png," Box Filter, Zero-Padding, Padding Width = 2, and Stride Count = 1.



Part_1b_Filtered_Image - NumPy object array (Not Responding)

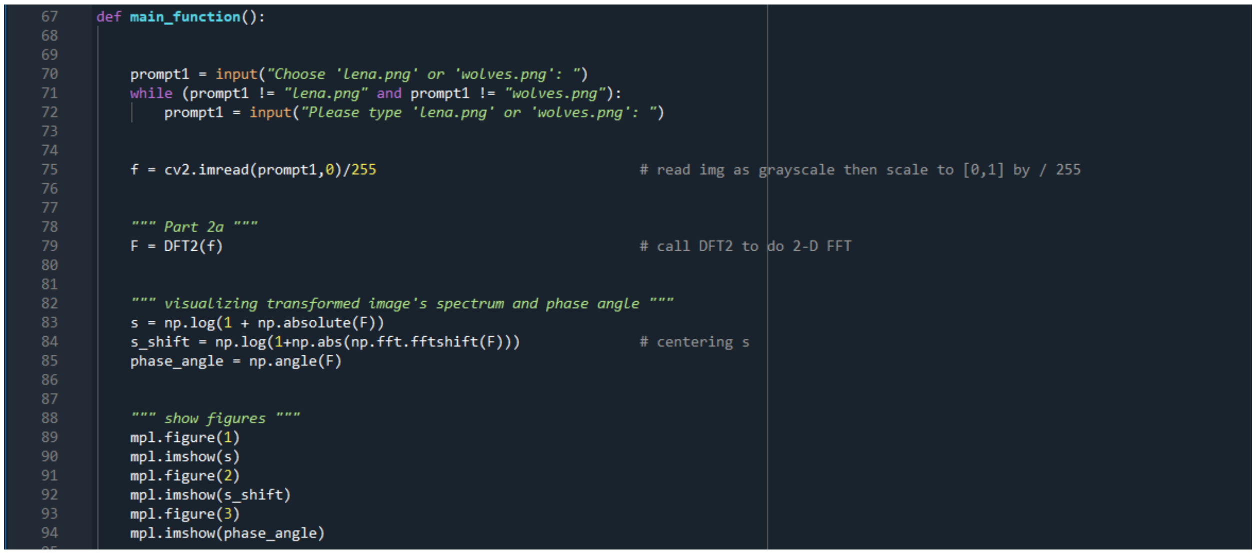
	510	511	512	513	514	515	516
507	0	0	0	0	0	0	0
508	0	0	0	0	0	0	0
509	0	0	0	0	0	0	0
510	0	0	0	0	0	0	0
511	0	0	0	0	0	0	0
512	0	0	28	28	28	0	0
513	0	0	28	28	28	0	0
514	0	0	28	28	28	0	0
515	0	0	0	0	0	0	0
516	0	0	0	0	0	0	0
517	0	0	0	0	0	0	0
518	0	0	0	0	0	0	0

Figure 12. "Part_1b_Filtered_Image.png" Matrix Values

Looking closely, there is a tiny white dot in figure 11 (circled). Doing the convolution with the box filter kernel by hand, the prediction is that there will be eight-adjacent pixels to the original pixel of value 255 at (512,512). This makes up a 3x3 cell filled with values 255/9 ≈ 28.33 spanning (511:513 , 511:513). Since padding width for zero-padding width is 2, the 3x3 cell is shifted by one unit diagonally. Thus, the 3x3 cell should be spanning (512:514 , 512:514). To verify the math, the image obtained in figure 11 is imported to Spyder IDE for inspection. Looking at matrix values of the image (figure 12), there is a 3x3 cell spanning (512:514 , 512:514) that is filled with values 255/8. This concludes that the conv2 is working.

Problem 2

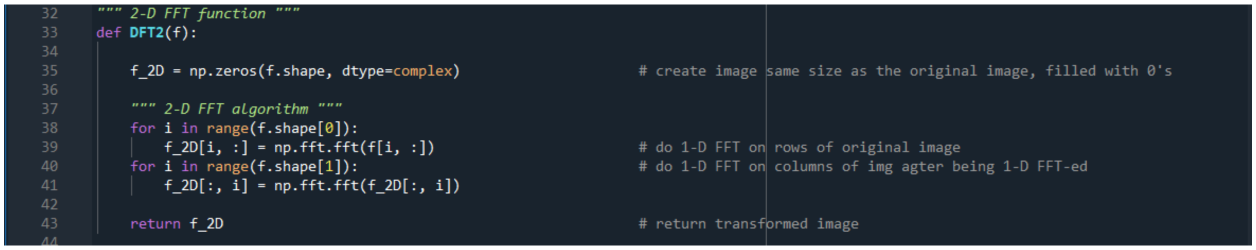
a) Using the built-in 1-D FFT from numpy, the 2-D FFT was implemented.



```
def main_function():


    prompt1 = input("Choose 'lena.png' or 'wolves.png': ")
    while (prompt1 != "lena.png" and prompt1 != "wolves.png"):
        prompt1 = input("Please type 'lena.png' or 'wolves.png': ")


    f = cv2.imread(prompt1,0)/255                           # read img as grayscale then scale to [0,1] by / 255


    """ Part 2a """
    F = DFT2(f)                                             # call DFT2 to do 2-D FFT


    """ visualizing transformed image's spectrum and phase angle """
    s = np.log(1 + np.absolute(F))
    s_shift = np.log(1+np.abs(np.fft.fftshift(F)))          # centering s
    phase_angle = np.angle(F)


    """ show figures """
    mpl.figure(1)
    mpl.imshow(s)
    mpl.figure(2)
    mpl.imshow(s_shift)
    mpl.figure(3)
    mpl.imshow(phase_angle)
```

Figure 13. Code Snip 5.

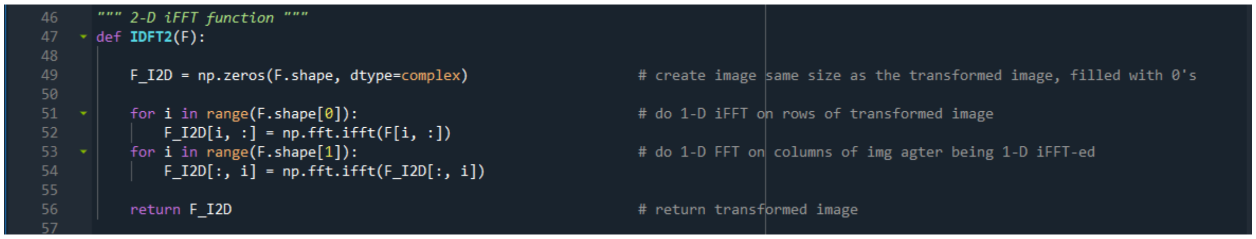
The 2-D FFT function DFT2 is called from the main function. The image of choice from the user (lena or wolves) is passed, transformed, and returned as transformed image F.



```
""" 2-D FFT function """
def DFT2(f):

    f_2D = np.zeros(f.shape, dtype=complex)                 # create image same size as the original image, filled with 0's

    """ 2-D FFT algorithm """
    for i in range(f.shape[0]):
        f_2D[i, :] = np.fft.fft(f[i, :])                    # do 1-D FFT on rows of original image
    for i in range(f.shape[1]):                             # do 1-D FFT on columns of img agter being 1-D FFT-ed
        f_2D[:, i] = np.fft.fft(f_2D[:, i])

    return f_2D                                             # return transformed image
```

Figure 14. Code Snip 6 – 2-D FFT.

The 2-D FFT algorithm is quite simple. Two “for” loops iterate through rows and columns of the image matrix respectively at the same time (parallelism). With the help of the built-in 1-D FFT, the first loop iterates through the original image’s row, performs 1-D FFT, and stores the row arrays to f_2D. Then, the second for loop follows the results obtained from the first loop to perform 1-D FFT column wide of the arrays stored in f_2D itself. The same logic is applied to the 2-D iFFT completed in part b of this problem using the built-in 1-D iFFT from numpy.



```
""" 2-D iFFT function """
def IDFT2(F):

    F_I2D = np.zeros(F.shape, dtype=complex)                  # create image same size as the transformed image, filled with 0's

    for i in range(F.shape[0]):                               # do 1-D iFFT on rows of transformed image
        F_I2D[i, :] = np.fft.ifft(F[i, :])
    for i in range(F.shape[1]):                               # do 1-D FFT on columns of img agter being 1-D iFFT-ed
        F_I2D[:, i] = np.fft.ifft(F_I2D[:, i])

    return F_I2D                                              # return transformed image
```

Figure 15. Code Snip 7 – 2-D iFFT.

Before being transformed, the original image is converted to grayscale. The gray image is then scaled to the range [0,1] by dividing 255.

After being transformed, the spectrum and phase angle of the transformed image are visualized by applying s = log(1+abs(F)). The following figures show the transformed "lena.png" spectrum s versus shifted s (s being centered) and its phase angle using matplot.figure

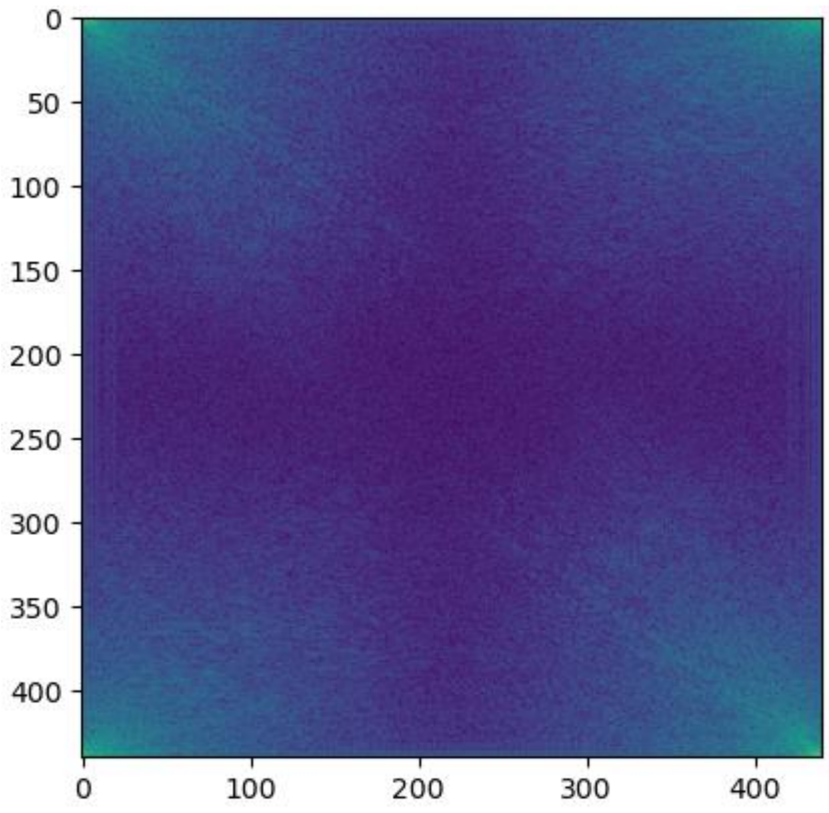


Figure 16. "Lena.png" Spectrum.

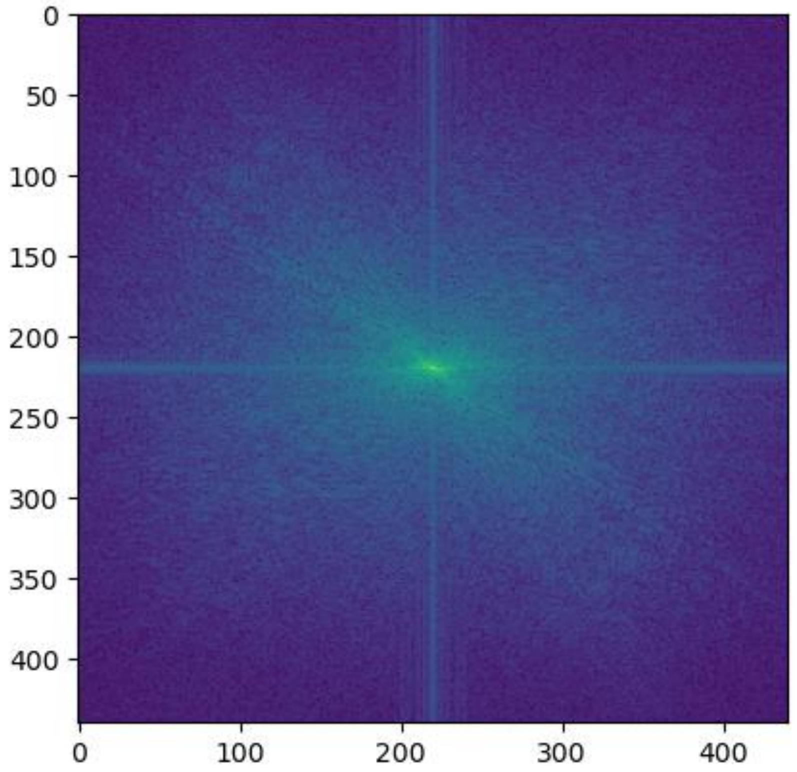


Figure 17. "Lena.png" "Shifted" Spectrum.

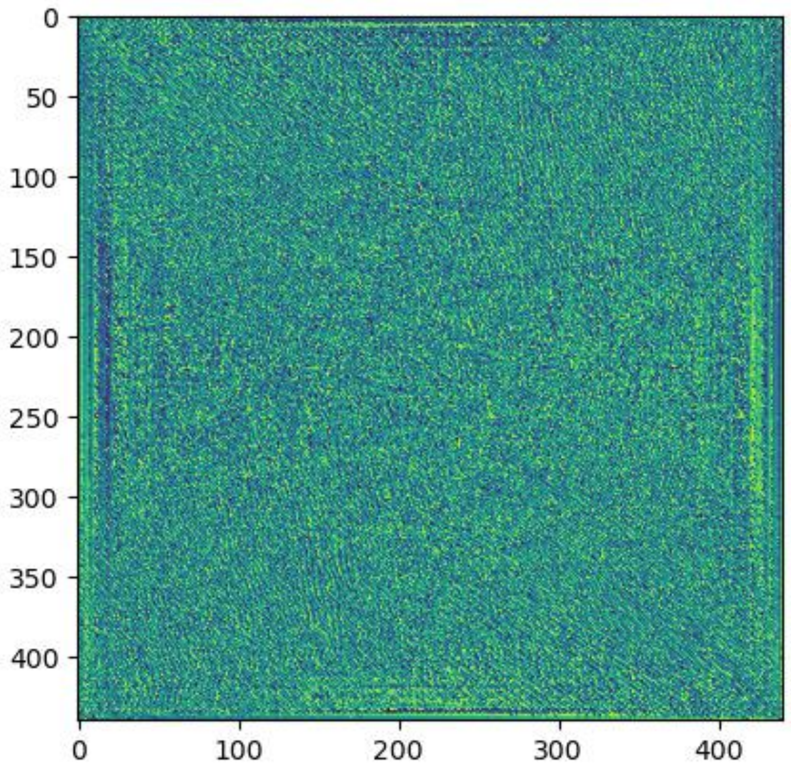
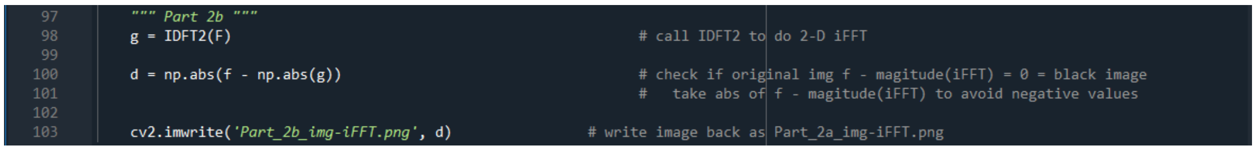


Figure 18. "Lena.png" Phase Angle.

b) Similar to part a, part b is to implement a 2-D iFFT using a built-in 1-D iFFT. The algorithm has been explained in part a. The side task of part b is two verify that after inverting the transformed image, the subtraction of iFFT(F), in which F is the FT of f, from f results a black image. It is another way to verify that the implemented DF2T and IDFT2 are working.

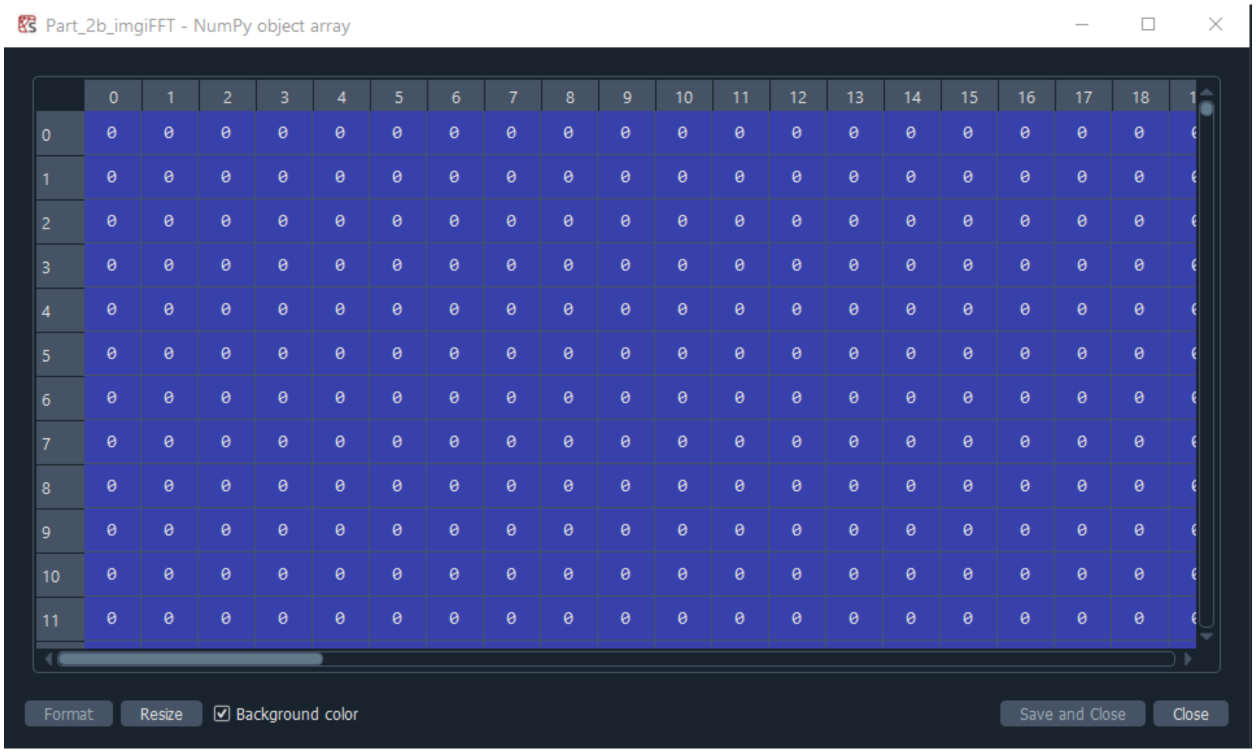


```
""" Part 2b """
g = IDFT2(F)                                        # call IDFT2 to do 2-D iFFT

d = np.abs(f - np.abs(g))                           # check if original img f - magitude(iFFT) = 0 = black image
                                                    #   take abs of f - magitude(iFFT) to avoid negative values

cv2.imwrite('Part_2b_img-iFFT.png', d)         # write image back as Part_2a_img-iFFT.png
```

Figure 19. Code Snip 8.

In figure 19, g is the result of F being iFFT-ed. Then, it is subtracted from f, which is the original image (“lena.png”). The absolute value of g calculates the magnitude of each individual value in the image matrix because the values in the matrix are imaginary numbers a + bi. Since, the observed values are extremely small ($10^{-15>}$), imaginary parts can also be neglected (e.g., insignificant to affect real parts) and use real parts for the subtraction. The absolute value is taken after the subtraction to make sure there are not negative values in the image matrix since there are some greater values of g being subtracted from smaller values of f. As a result, a black image is obtained, verifying that the implemented DFT2 and IDF2T are working properly. The verification is also done when compare the implementations to the numpy built-in fft2 and ifft2 and the results (matrix values) are identical.

Figure 20. “Part_2b_img-iFFT.png.”



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 21. "Part_2b_img-iFFT.png" Matrix Value.

<u>Attached to this report:</u>

1) kddo_code

- P1_q1.py
- P1_q2.py

2) kddo_images

- Original Images
  - lena,png
  - wolves.png
- Result Images
  - Part_1a_Filtered_Image.png
  - Part_1b_Filtered_Image.png
  - Part_2b_img-iFFT.png
  - Phase_Angle.png
  - s_Shifted_Spectrum.png
  - s_Spectrum.png