### **ECE 558 - Digital Imaging System**

**Question 1A**

1. **The code is written in such a manner that the User is given options to select between wolves.png and lena.png**
2. **Then the user chooses the type of filter operation**
3. **Then chooses the type of padding and the code also allows the user to choose the Width of the padding on the image.**
4. **The code asks the user to select to select the operations whether on an RGB or a Gray Image.**
5. **Then finally the user is aksed to enter the Stride count.**
6. **Output is obtained and saved.**

**import cv2**

**import numpy as np**

**I = int(input("'1' for wolves.png \n '2' for lena.png"))**

**if I == 1:**

**testImage = cv2.imread(r'D:\NCSU\NCSU Courses\DIS - 558\DIS Project2\wolves.png')**

**grayImage = cv2.cvtColor(testImage, cv2.COLOR\_BGR2GRAY)**

**imageShape = grayImage.shape**

**cv2.imshow('Original',grayImage)**

**cv2.waitKey(0)**

**else:**

**testImage = cv2.imread(r'D:\NCSU\NCSU Courses\DIS - 558\DIS Project2\lena.png')**

**grayImage = cv2.cvtColor(testImage, cv2.COLOR\_BGR2GRAY)**

**imageShape = grayImage.shape**

**cv2.imshow('Original',grayImage)**

**cv2.waitKey(0)**

**#AVERAGE**

**def avgFilter():**

**krnl = np.ones((3,3))/(3\*\*2)**

**return krnl**

**#PREWITT**

**def prewittFilter\_mx():**

**krnl = np.array([[-1, 0, 1],**

**[-1, 0, 1],**

**[-1, 0, 1]])**

**return krnl**

**def prewittFilter\_my():**

**krnl = np.array([[1, 1, 1],**

**[0, 0, 0],**

**[-1, -1, -1]])**

**return krnl**

**#SOBEL**

**def sobelFilter\_mx():**

**krnl = np.array([[-1, 0, 1],**

**[-2, 0, 2],**

**[-1, 0, 1]])**

**return krnl**

**def sobelFilter\_my():**

**krnl = np.array([[1, 2, 1],**

**[0, 0, 0],**

**[-1, -2, -1]])**

**return krnl**

**#ROBERTS**

**def robertsFilter\_mx():**

**krnl = np.array([[0, 1],**

**[-1, 0]])**

**return krnl**

**def robertsFilter\_my():**

**krnl = np.array([[1, 0],**

**[0, -1]])**

**return krnl**

**#FIRST ORDER DERIVATIVE**

**def firstOrder():**

**fod = int(input("'1' for first order row matrix \n '2' for first order column matrix"))**

**if fod == 1:**

**krnl = np.array([[-1, 1]])**

**elif fod == 2:**

**fod\_c = int(input("'1' for column type 1 -- [-1;1] \n '2' for column type 2 -- [1;-1]"))**

**if fod\_c == 1:**

**krnl = np.array([[-1],[1]])**

**elif fod\_c == 2:**

**krnl = np.array([[1], [-1]])**

**return krnl**

**#ZERO PADDING**

**def zeroPadding(f, w):**

**size = f.shape**

**horizontalPadding = size[0] + w - 1**

**verticalPadding = size[1] + w - 1**

**paddedImage = np.zeros ((horizontalPadding,verticalPadding))**

**for r in range(size[0]):**

**for c in range(size[1]):**

**paddedImage[r+int((w-1)/2),c+int((w-1)/2)] = grayImage[r,c]**

**return paddedImage**

**#WRAP AROUND**

**def wrapAround(f,w):**

**image = f**

**paddingMeausre = w**

**leftTop=f[: paddingMeausre,: paddingMeausre]**

**leftBottom=f[-paddingMeausre :,: paddingMeausre]**

**rightTop=f[: paddingMeausre,-paddingMeausre :]**

**rightBottom=f[-paddingMeausre :,-paddingMeausre :]**

**left = np.array([f[:,-1]]).T**

**right = np.array([f[:, 0]]).T**

**for i in range(paddingMeausre):**

**image = np.vstack(((f[-1-i,:]) , image, (f[(0+i),:])))**

**if i < (paddingMeausre-1):**

**left = np.hstack(((np.array([f[:,-(2+i)]]).T), left))**

**right = np.hstack((right,(np.array([f[:,(1+i)]]).T)))**

**else:**

**left = np.vstack((rightBottom, left, rightTop))**

**right = np.vstack((leftBottom, right, leftTop))**

**image= np.hstack(((left, image, right)))**

**return image**

**#COPY EDGE**

**def copyEdge(f,w):**

**image = f**

**paddingMeausre = w**

**leftTop = f[: paddingMeausre, : paddingMeausre]**

**leftBottom = f[-paddingMeausre:, : paddingMeausre]**

**rightTop = f[: paddingMeausre, -paddingMeausre:]**

**rightBottom = f[-paddingMeausre:, -paddingMeausre:]**

**left = np.array([f[:,-1]]).T**

**right = np.array([f[:, 0]]).T**

**for i in range(paddingMeausre):**

**image = np.vstack(((f[0,:]) , image, (f[-1,:])))**

**if i < (paddingMeausre-1):**

**left = np.hstack(((np.array([f[:,0]]).T), left))**

**right = np.hstack((right,(np.array([f[:,-1]]).T)))**

**else:**

**left = np.vstack((leftTop , left , leftBottom))**

**right = np.vstack((rightTop , right , rightBottom))**

**image= np.hstack(((left, image, right)))**

**return image**

**#REFLECT ACROSS**

**def reflectAcross(f,w):**

**image = f**

**paddingMeausre = w**

**for i in range (paddingMeausre):**

**image = np.vstack(((f[(1+i),:]) , image, (f[-(2+i),:])))**

**outputImage = image**

**for i in range (paddingMeausre):**

**outputImage = np.hstack(((np.array([image[:,(1+i)]]).T), outputImage, (np.array([image[:,-(2+i)]]).T)))**

**return outputImage**

**#PADDING**

**def padding\_Type1(img,pT,w):**

**if pT == 1:**

**paddedImage = zeroPadding(img, w)**

**d = conv2Gray(paddedImage, kernal)**

**elif pT == 2:**

**paddedImage = wrapAround(img, w)**

**d = conv2Gray(paddedImage, kernal)**

**elif pT == 3:**

**paddedImage = copyEdge(img, w)**

**d = conv2Gray(paddedImage, kernal)**

**elif pT == 4:**

**paddedImage = reflectAcross(img, w)**

**d = conv2Gray(paddedImage, kernal)**

**return d**

**def padding\_Type2(img,pT,w):**

**if pT == 1:**

**paddedImage = zeroPadding(img, w)**

**elif pT == 2:**

**paddedImage = wrapAround(img, w)**

**elif pT == 3:**

**paddedImage = copyEdge(img, w)**

**elif pT == 4:**

**paddedImage = reflectAcross(img, w)**

**return paddedImage**

**#2D CONVOLUTION**

**def conv2Gray(f, w):**

**paddedWidth, paddedHeight = f.shape**

**kernalWidth, kernalHeight = w.shape**

**stride = int(input("stride:"))**

**newWidth = (paddedWidth - kernalWidth) // stride +1**

**newHeight = (paddedHeight - kernalHeight) // stride +1**

**newImage = np.zeros((newWidth, newHeight)).astype(np.float32)**

**for x in range(newWidth):**

**if x > paddedWidth - kernalWidth:**

**break**

**if x % stride == 0:**

**for y in range(newHeight):**

**if y > paddedHeight - kernalHeight:**

**break**

**if y % stride == 0:**

**newImage[x][y] = np.sum(f[x \* stride:x \* stride + kernalWidth, y \* stride:y \* stride +kernalHeight]\*kernal).astype(np.float32)**

**return newImage**

**def conv2RGB(f, paddingType,kernal, w):**

**b,g,r = cv2.split(f)**

**b = np.array(b)**

**g = np.array(g)**

**r = np.array(r)**

**b = padding\_Type2(b,paddingType,w)**

**g = padding\_Type2(g, paddingType, w)**

**r = padding\_Type2(r, paddingType, w)**

**paddedWidth, paddedHeight = b.shape**

**kernalWidth ,kernalHeight = kernal.shape**

**stride = int(input("stride:"))**

**newWidth = (paddedWidth - kernalWidth) // stride +1**

**newHeight = (paddedHeight - kernalHeight) // stride +1**

**newImage\_b= np.zeros((newWidth, newHeight)).astype(np.float32)**

**newImage\_g = np.zeros((newWidth, newHeight)).astype(np.float32)**

**newImage\_r = np.zeros((newWidth, newHeight)).astype(np.float32)**

**for x in range(newWidth):**

**if x > paddedWidth - kernalWidth:**

**break**

**if x % stride == 0:**

**for y in range(newHeight):**

**if y > paddedHeight - kernalHeight:**

**break**

**if y % stride == 0:**

**newImage\_b[x][y] = np.sum(b[x \* stride:x \* stride + kernalWidth, y \* stride:y \* stride +kernalHeight]\*kernal).astype(np.float32)**

**newImage\_g[x][y] = np.sum(g[x \* stride:x \* stride + kernalWidth, y \* stride:y \* stride + kernalHeight] \* kernal).astype(np.float32)**

**newImage\_r[x][y] = np.sum(r[x \* stride:x \* stride + kernalWidth, y \* stride:y \* stride + kernalHeight] \* kernal).astype(np.float32)**

**newImage = cv2.merge([newImage\_r,newImage\_g,newImage\_b])**

**return newImage**

**kernalType = int(input("'1' for averaging filter \n '2' for Prewitt Filter \n '3' for Sobel Filter \n '4' for Roberts Filter \n '5' for first order derivative Filters"))**

**if kernalType == 1:**

**kernal = avgFilter()**

**elif kernalType == 2:**

**type = input("mx or my:")**

**if type =="mx":**

**kernal = prewittFilter\_mx()**

**else:**

**kernal = prewittFilter\_my()**

**elif kernalType == 3:**

**type = input("mx or my:")**

**if type == "mx":**

**kernal = sobelFilter\_mx()**

**else:**

**kernal = sobelFilter\_my()**

**elif kernalType == 4:**

**type = input("mx or my:")**

**if type == "mx":**

**kernal = robertsFilter\_mx()**

**else:**

**kernal = robertsFilter\_my()**

**elif kernalType == 5:**

**kernal = firstOrder()**

**paddingType=int(input("'1' for zero padding \n '2' for wrap around \n '3' for copy edge \n '4' for reflect across edge"))**

**w = int(input("Padding Dimension:"))**

**imageType = int(input("'1' for Gray Scale \n '2' for RGB image"))**

**if imageType == 1:**

**d = padding\_Type1(grayImage,paddingType,w)**

**else:**

**d = conv2RGB(testImage,paddingType,kernal,w)**

**convolutedImage = d[w:d.shape[0]-w, w:d.shape[1]-w]**

**cv2.imwrite('paddedImage.png', d)**

**testImage1 = cv2.imread(r'D:\PyCharm Community Edition 2022.2.2\pythonProject2\paddedImage.png')**

**cv2.imshow('Padded Image',testImage1)**

**cv2.waitKey(0)**

**cv2.imwrite('convoluted.png', convolutedImage)**

**testImage2 = cv2.imread(r'D:\PyCharm Community Edition 2022.2.2\pythonProject2\convoluted.png')**

**cv2.imshow('Convoluted Image',testImage2)**

**cv2.waitKey(0)**

**cv2.destroyAllWindows()**

**Output:**

**Input image**

****

**Gray outputs for zero padding with padding width 50 and box filter 3x3, with stride 1.**

****

****

**Outputs for wrap around padding with padding width 50 and prewit flter\_mx, with stride 1.**

****

**Outputs for copy edge padding with padding width 50 and sobel flter\_my, with stride 1.**

****

**Outputs for reflect across padding with padding width 50 and roberts flter\_mx, with stride 1.**

****

**Similar outputs for wolves.png at copy edge padding with padding width 50 and sobel flter\_mx, with stride 1.**

****

**Logic works for all types of images of different sizes and for 3 or 2 channels.**

**Question 1B**

**# importing all the libraries required**

**import cv2**

**import numpy as np**

**def unitImpusle():**

**unitImage = np.zeros((1024, 1024), dtype="uint8")**

**width, height = unitImage.shape**

**impulseLocation = [511, 511]**

**for i in range(width):**

**for j in range(height):**

**if (i == impulseLocation[0] and j == impulseLocation[1]):**

**unitImage[511, 511] = 255**

**return unitImage**

**unitImage = unitImpusle()**

**cv2.imwrite('unitImage.png',unitImage)**

**# loading the image sample**

**testImage = cv2.imread(r'D:\PyCharm Community Edition 2022.2.2\pythonProject3\unitImage.png')**

**testImage = cv2.cvtColor(testImage, cv2.COLOR\_BGR2GRAY)**

**cv2.imshow('Original',testImage)**

**cv2.waitKey(0)**

**def firstOrder():**

**fod = int(input("'1' for first order row matrix \n '2' for first order column matrix"))**

**if fod == 1:**

**krnl = np.array([[-1, 1]])**

**elif fod == 2:**

**fod\_c = int(input("'1' for column type 1 -- [-1;1] \n '2' for column type 2 -- [1;-1]"))**

**if fod\_c == 1:**

**krnl = np.array([[-1],[1]])**

**elif fod\_c == 2:**

**krnl = np.array([[1], [-1]])**

**return krnl**

**#AVERAGE**

**def avgFilter():**

**krnl = np.ones((3,3))/(3\*\*2)**

**return krnl**

**#PREWITT**

**def prewittFilter\_mx():**

**krnl = np.array([[-1, 0, 1],**

**[-1, 0, 1],**

**[-1, 0, 1]])**

**return krnl**

**def prewittFilter\_my():**

**krnl = np.array([[1, 1, 1],**

**[0, 0, 0],**

**[-1, -1, -1]])**

**return krnl**

**#SOBEL**

**def sobelFilter\_mx():**

**krnl = np.array([[-1, 0, 1],**

**[-2, 0, 2],**

**[-1, 0, 1]])**

**return krnl**

**def sobelFilter\_my():**

**krnl = np.array([[1, 2, 1],**

**[0, 0, 0],**

**[-1, -2, -1]])**

**return krnl**

**#ROBERTS**

**def robertsFilter\_mx():**

**krnl = np.array([[0, 1],**

**[-1, 0]])**

**return krnl**

**def robertsFilter\_my():**

**krnl = np.array([[1, 0],**

**[0, -1]])**

**return krnl**

**def zeroPadding(f, w):**

**paddedWidth, paddedHeight = f.shape**

**horizontalPadding = paddedWidth + w - 1**

**verticalPadding = paddedHeight + w - 1**

**paddedImage = np.zeros ((horizontalPadding,verticalPadding))**

**for r in range(paddedWidth):**

**for c in range(paddedHeight):**

**paddedImage[r+int((w-1)/2),c+int((w-1)/2)] = testImage[r,c]**

**return paddedImage**

**def wrapAround(f,w):**

**image = f**

**paddedWidth, paddedHeight = f.shape**

**paddingMeausre = w**

**leftTop=f[: paddingMeausre,: paddingMeausre]**

**leftBottom=f[-paddingMeausre :,: paddingMeausre]**

**rightTop=f[: paddingMeausre,-paddingMeausre :]**

**rightBottom=f[-paddingMeausre :,-paddingMeausre :]**

**left = np.array([f[:,-1]]).T**

**right = np.array([f[:, 0]]).T**

**for i in range(paddingMeausre):**

**image = np.vstack(((f[-1-i,:]) , image, (f[(0+i),:])))**

**if i < (paddingMeausre-1):**

**left = np.hstack(((np.array([f[:,-(2+i)]]).T), left))**

**right = np.hstack((right,(np.array([f[:,(1+i)]]).T)))**

**else:**

**left = np.vstack((rightBottom, left, rightTop))**

**right = np.vstack((leftBottom, right, leftTop))**

**image= np.hstack(((left, image, right)))**

**return image**

**def copyEdge(f,w):**

**image = f**

**paddedWidth, paddedHeight = f.shape**

**paddingMeausre = w**

**leftTop = f[: paddingMeausre, : paddingMeausre]**

**leftBottom = f[-paddingMeausre:, : paddingMeausre]**

**rightTop = f[: paddingMeausre, -paddingMeausre:]**

**rightBottom = f[-paddingMeausre:, -paddingMeausre:]**

**left = np.array([f[:,-1]]).T**

**right = np.array([f[:, 0]]).T**

**for i in range(paddingMeausre):**

**image = np.vstack(((f[0,:]) , image, (f[-1,:])))**

**if i < (paddingMeausre-1):**

**left = np.hstack(((np.array([f[:,0]]).T), left))**

**right = np.hstack((right,(np.array([f[:,-1]]).T)))**

**else:**

**left = np.vstack((leftTop , left , leftBottom))**

**right = np.vstack((rightTop , right , rightBottom))**

**image= np.hstack(((left, image, right)))**

**return image**

**def reflectAcross(f,w):**

**image = f**

**paddedWidth, paddedHeight = f.shape**

**paddingMeausre = w**

**for i in range (paddingMeausre):**

**image = np.vstack(((f[(1+i),:]) , image, (f[-(2+i),:])))**

**outputImage = image**

**for i in range (paddingMeausre):**

**outputImage = np.hstack(((np.array([image[:,(1+i)]]).T), outputImage, (np.array([image[:,-(2+i)]]).T)))**

**return outputImage**

**def conv2(f, w):**

**paddedWidth, paddedHeight = f.shape**

**kernalWidth, kernalHeight = w.shape**

**stride = int(input("stride:"))**

**newWidth = (paddedWidth - kernalWidth) // stride +1**

**newHeight = (paddedHeight - kernalHeight) // stride +1**

**newImage = np.zeros((newWidth, newHeight)).astype(np.float32)**

**for x in range(newWidth):**

**if x > paddedWidth - kernalWidth:**

**break**

**if x % stride == 0:**

**for y in range(newHeight):**

**if y > paddedHeight - kernalHeight:**

**break**

**if y % stride == 0:**

**newImage[x][y] = np.sum(f[x \* stride:x \* stride + kernalWidth, y \* stride:y \* stride +kernalHeight]\*kernal).astype(np.float32)**

**return newImage**

**kernalType = int(input("'1' for averaging filter \n '2' for Prewitt Filter \n '3' for Sobel Filter \n '4' for Roberts Filter \n '5' for first order derivative Filters"))**

**if kernalType == 1:**

**kernal = avgFilter()**

**elif kernalType == 2:**

**type = input("mx or my:")**

**if type =="mx":**

**kernal = prewittFilter\_mx()**

**else:**

**kernal = prewittFilter\_my()**

**elif kernalType == 3:**

**type = input("mx or my:")**

**if type == "mx":**

**kernal = sobelFilter\_mx()**

**else:**

**kernal = sobelFilter\_my()**

**elif kernalType == 4:**

**type = input("mx or my:")**

**if type == "mx":**

**kernal = robertsFilter\_mx()**

**else:**

**kernal = robertsFilter\_my()**

**elif kernalType == 5:**

**kernal = firstOrder()**

**paddingType=int(input("'1' for zero padding \n '2' for wrap around \n '3' for copy edge \n '4' for reflect across edge"))**

**if paddingType == 1:**

**w = int(input("Padding Dimension:"))**

**paddedImage = zeroPadding(testImage, w)**

**d = conv2(paddedImage, kernal)**

**elif paddingType == 2:**

**w = int(input("Padding Dimension:"))**

**paddedImage = wrapAround(testImage, w)**

**d = conv2(paddedImage, kernal)**

**elif paddingType == 3:**

**w = int(input("Padding Dimension:"))**

**paddedImage = copyEdge(testImage, w)**

**d = conv2(paddedImage, kernal)**

**elif paddingType == 4:**

**w = int(input("Padding Dimension:"))**

**paddedImage = reflectAcross(testImage, w)**

**d = conv2(paddedImage, kernal)**

**shaped1, shaped2 = d.shape**

**print(shaped1, shaped2)**

**convolutedImage = d[w:shaped1-w, w:shaped2-w]**

**convolutedImageSize = convolutedImage.shape**

**print(convolutedImageSize)**

**cv2.imwrite('unit.png', d)**

**testImage = cv2.imread(r'D:\PyCharm Community Edition 2022.2.2\pythonProject2\unit.png')**

**cv2.imshow('Padded Image',testImage)**

**cv2.waitKey(0)**

**cv2.imwrite('unit.png', convolutedImage)**

**testImage1 = cv2.imread(r'D:\PyCharm Community Edition 2022.2.2\pythonProject2\unit.png')**

**cv2.imshow('Padded Image',testImage1)**

**cv2.waitKey(0)**

**cv2.destroyAllWindows()**

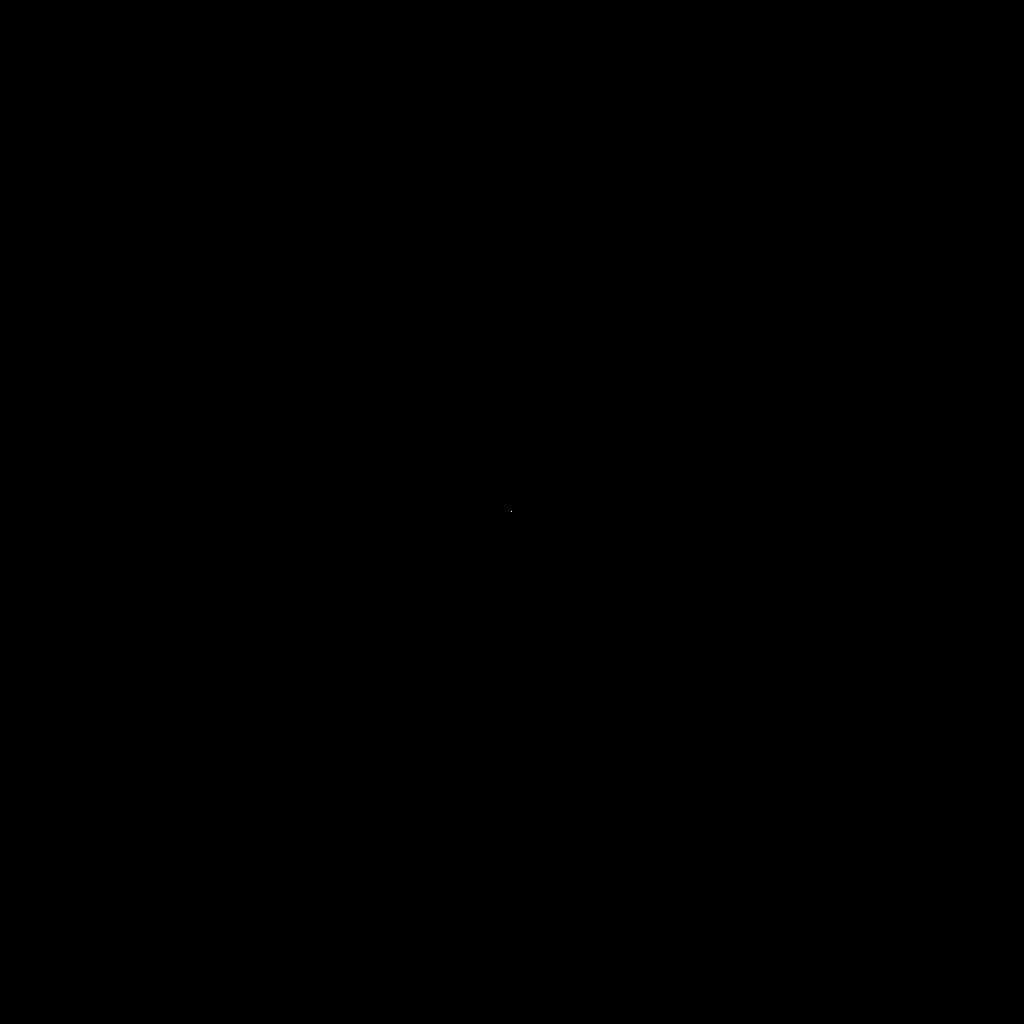
**Understanding:**

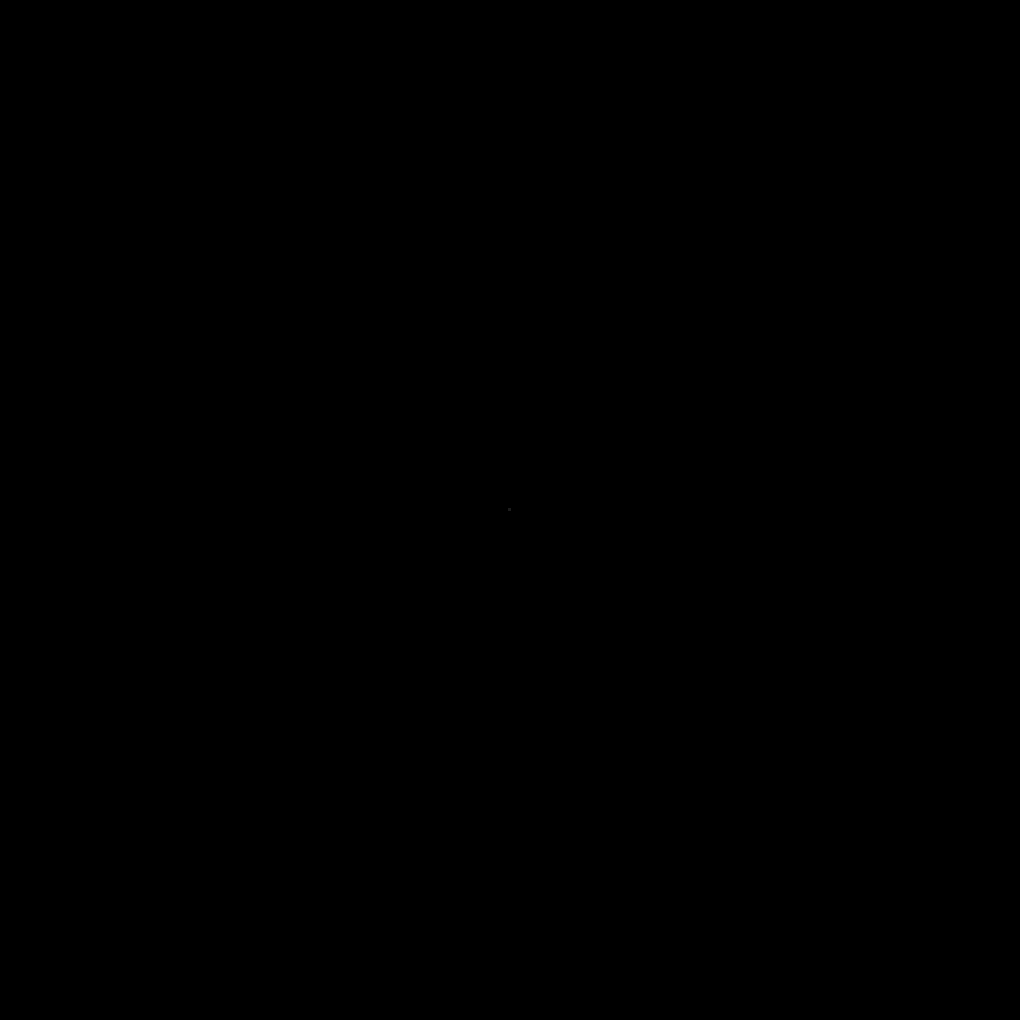
**The the center/ unit pixel takes the value of the target used kernel.**

**For example, in case of box fiter 3x3, the pixel at 511x511 becomes 1/9.**

**Output:**

**(zoomed in pictures) 1. input image 2. output image for averaging filter**

****

****

**Question 2A and 2B:**

**import numpy as np**

**import cv2**

**from matplotlib import pyplot as plt**

**def DFT2(f):**

**imageShape = f.shape**

**outputImage = np.zeros(imageShape)**

**for r in range(imageShape[0]):**

**outputImage[r, :] = np.fft.fft(f[r, :])**

**for c in range(imageShape[1]):**

**outputImage[:, c] = np.fft.fft(outputImage[:, c])**

**return outputImage**

**def iDFT2(F):**

**imageShape = F.shape**

**outputImage = np.zeros(F.shape)**

**for R in range(imageShape[0]):**

**outputImage[R, :] = np.fft.ifft(F[R, :])**

**for C in range(imageShape[1]):**

**outputImage[:, C] = np.fft.ifft(outputImage[:, C])**

**return outputImage**

**I = int(input("'1' for wolves.png \n '2' for lena.png"))**

**if I == 1:**

**testImage = cv2.imread(r'D:\NCSU\NCSU Courses\DIS - 558\DIS Project2\wolves.png')**

**f = cv2.cvtColor(testImage, cv2.COLOR\_BGR2GRAY)/255**

**cv2.imshow('Original',f)**

**cv2.waitKey(0)**

**else:**

**testImage = cv2.imread(r'D:\NCSU\NCSU Courses\DIS - 558\DIS Project2\lena.png')**

**f = cv2.cvtColor(testImage, cv2.COLOR\_BGR2GRAY)/255**

**cv2.imshow('Original',f)**

**cv2.waitKey(0)**

**F = DFT2(f)**

**phase = np.log(1 + np.absolute(F))**

**phaseShift = np.log(1 + np.abs(np.fft.fftshift(F)))**

**phaseAngle = np.angle(F)**

**plt.imshow(phase, cmap = 'gray')**

**plt.show()**

**plt.imshow(phaseShift, cmap = 'gray')**

**plt.show()**

**plt.imshow(phaseAngle, cmap = 'gray')**

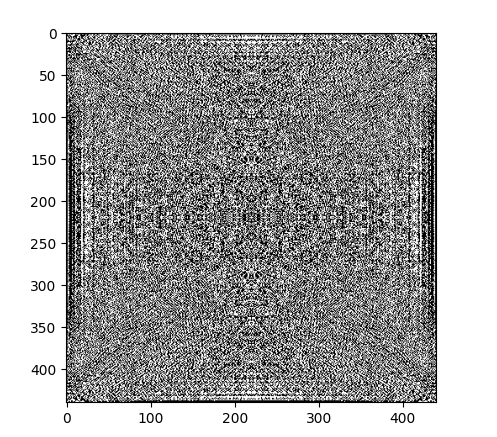
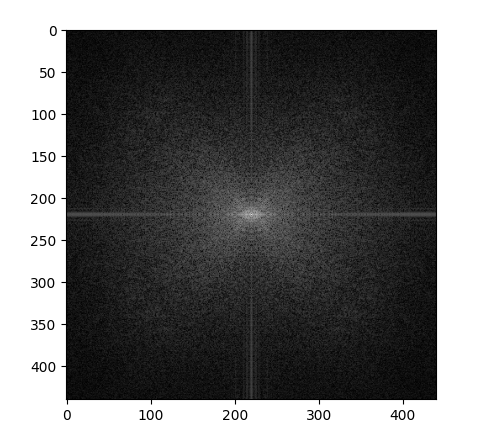
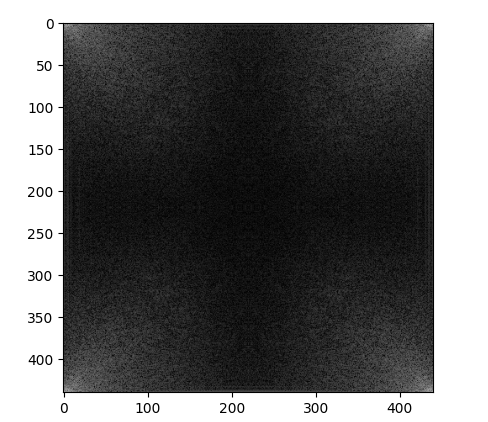
**plt.show()**

**g = iDFT2(F)**

**d = np.abs(f - np.abs(g))**

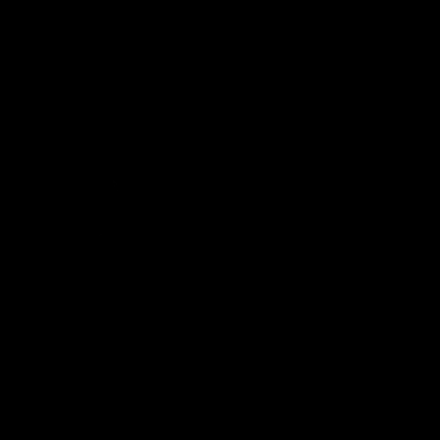
**cv2.imwrite('iDFFT.png', d)**

**Lena.png Output for 2A:**

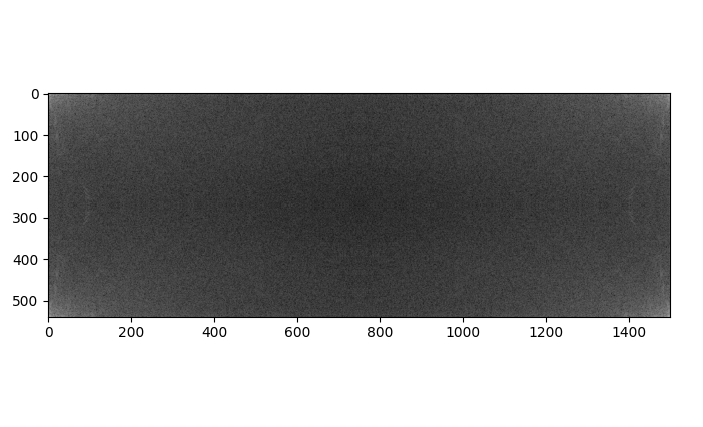
****

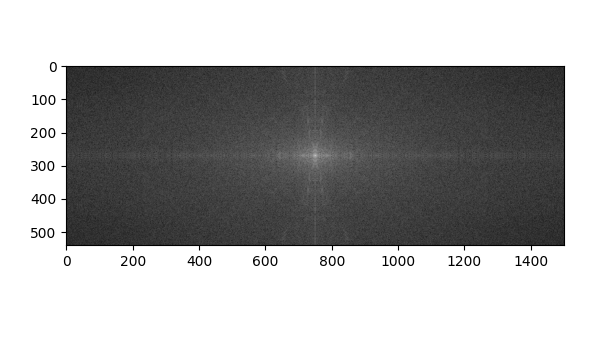
**Image 1: Phase Image 2: Phase Shift Image 3: Phase Angle**

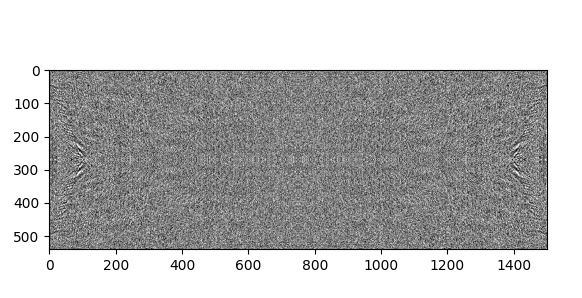
**Lena.png Output for 2B: pretty much a black image.**

****

**Wolves.png Output for 2A:**

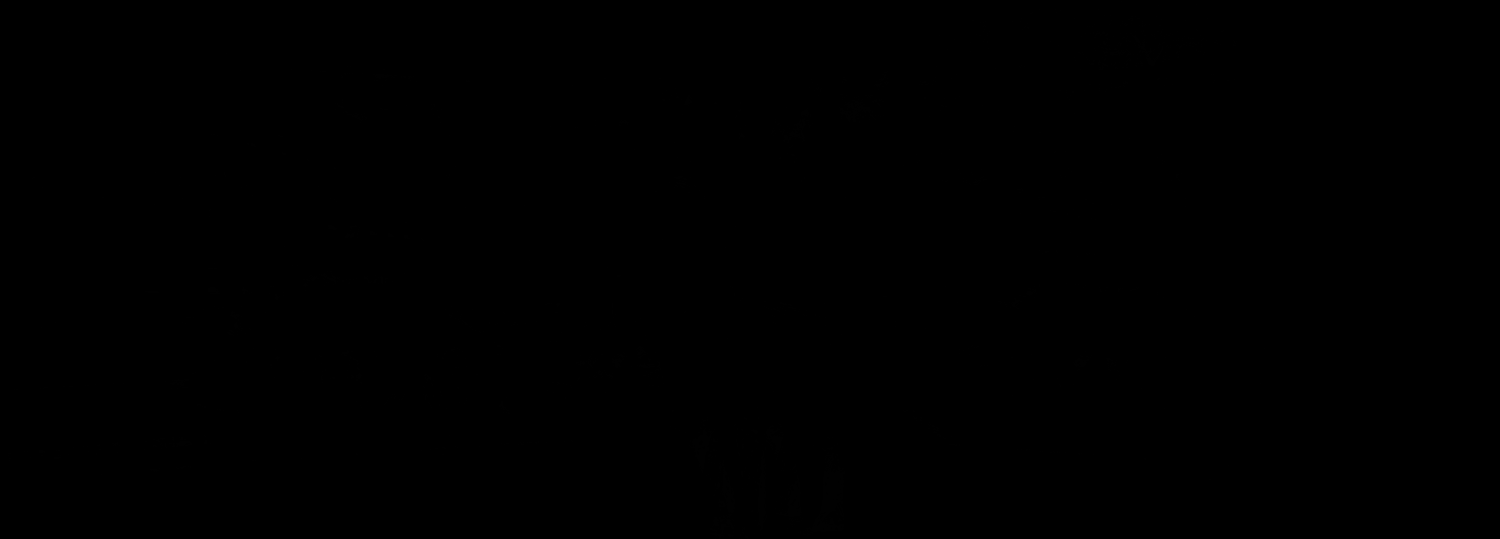
****

****

****

**Image 1: Phase Image 2: Phase Shift Image 3: Phase Angle**

**Wolves.png Output for 2B: pretty much a black image:**

****