BIRZEIT UNIVERSITY
Faculty of Engineering and Technology
Electrical and Computer Engineering Department

ENCS5343 Computer Vision
Assignment #1

Student name: Al-Ayham Maree

Student ID: 1191408

Notes:

1- **Use this page as a cover for your homework.**
2- **Late home works will not be accepted (the system will not allow it).**
3- **Due date is December 18th, 2023 at 11:59 pm on ritaj.**
4- **Report including Input and Output images (Soft Copy). Include all the code you write to implement/solve this assignment with your submission.**
5- **Organize your output files (images) as well as used codes to be in a folder for each question (Q1, Q2, etc.). Then add the solution of all questions along with this report in one folder named** Assign1**. Compress the** Assign1 **folder and name it as (**Assign1_LastName_FirstName_StudetnsID.Zip**).**
6- **Please write some lines about how to run your code.**
7- **You might need to use OpenCV Python in your implementation.**

Question#1: Look for an image from the internet with the following properties: 8-bit gray-level, 256x256 pixels in size.

1- Show this image. Don't use your friends' ones.

Figure 1: Tiger Photo with the specifications (Original Photo)

The question was solved using the tiger photo above, which has the following specifications: 256 × 256 pixels, 8-bit grayscale.

2- Apply a power law transformation with gamma=0.4 to the image and show the image after the transformation.

The original image has been transformed by using a power law transformation, where s is the density of the transformed pixels, and r is the density of the original pixels and gamma and c are constants in the following equation has been used:

$$s = c.\ r^{\gamma}$$

And in our task the gamma constant equals to 0.4 has been used, and the result was:

Figure 2: Tiger image transformed with power law

In the figure shown above as a result of transformation using power law, the result obvious that there is an increasing in the brightness and when gamma was greater than one, all pixel that has high brightness will be improved and the other pixels stay same as before transformation, and vice versa, when the gamma less than one, all pixels that has low brightness will be improved and the other pixels stay same.

```python
# Part2
import cv2

import numpy as np

#read the image using cv2
tigerImage = cv2.imread('tiger.jpg', cv2.IMREAD_GRAYSCALE)


#use power law tranformation with gamma value= 0.4, by created an array with changing in the values
tigerTransImage = np.array(255.0 * (tigerImage / 255.0) ** 0.4, dtype='uint8')


#save the trnasformed image
cv2.imwrite("tigerTransImage.jpg", tigerTransImage)


#showing the results and the original for the comparison
cv2.imshow("Original tiger Image", tigerImage)


cv2.imshow("Transformed tiger Image", tigerTransImage)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 3: Part 2 code

3- Add a zero-mean Gaussian noise (with variance =40 gray-levels) to the original image and show the resulting image.

It should be noticed that when the Gaussian noise with variance equals to 40, or the standard deviation was the square root of the variance and it is equal to 6.3246, and mean-value equals to zero, there are an differences shown on some regions in the whole pixels of the original image but the distribution of the noise was randomly based on the Gaussian equation has been used in the code.

Figure 4: Tiger image with Gaussian noise with standard deviation value equal to 6.3246

As mentioned above, the differences that added on the original image was a reason to low the resolution of the image, and it is more obvious that the image has a noise, and there is a distortion on the resulting image from the code.

```python
# Part3
import cv2
import numpy as np
#read the image using cv2
tigerImage = cv2.imread('tiger.jpg', cv2.IMREAD_GRAYSCALE)
# calculate the standard deviation using the square root of the variance value which equals 40
Standard_dev = np.sqrt(40)
#get the size of the image using shape parameter
height, width = tigerImage.shape
#generate and gaussian noise randomly based on the standrd deviation value and with zero mean-value
# and using the height and width of the image
Gaussian_Noise = np.random.normal(0,Standard_dev, (height, width))
#adding the nose on the array of original image by using + sign
tigerImageWithNoise = tigerImage + Gaussian_Noise
#clip the image to be within range between 0 to 255 and then convert the data type of the array to unsigned 8-bit.
tigerImageWithNoise = np.clip(tigerImageWithNoise, 0, 255).astype(np.uint8)
#save the image
cv2.imwrite('tigerImageWithNoise.jpg', tigerImageWithNoise)
#show results and original image
cv2.imshow('Original tiger Image', tigerImage)
cv2.imshow('Noisy tiger Image', tigerImageWithNoise)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 5: Part 3 code

4- Apply a 5 by 5 mean filter to the noisy-image you obtained in point 3 above and show the result. Discuss the results in your report.

Figure 6: Tiger image has been filter by mean filter after Gaussian noise has been added

Notice that in the image above, the mean filter blur () has been used to filter the noisy image from the previous part, with kernel size or matrix size 5x5, and the result. However, the mean filter has a primary purpose that the filter reduces the noise in the image, so this operation name is averaging, and the averaging make the image more smooth and the evidence of it, that the image has been applied to blurring by smoothing the pixel values, while the noise reduction is achieved, the mean filter led to blurring, so it leading to loss some small details in the image and sharping. Moreover, the size of filter is the basic in the filtering operation, so the choice of the filter size affects the extent of smoothing, and that's mean lager filter sizes may cause in smoothing and loss of details in the image.

```python
# Part4
import cv2
import numpy as np
#read noisy image from the previous images with adding gaussian noise
tigerImageWithNoise = cv2.imread('tigerImageWithNoise.jpg', cv2.IMREAD_GRAYSCALE)
#use mean filter or avargeing filter, to noise reduction,
# with kernel size 5x5 using function blur, with conversion to 8-bit datatype
tigerMeanFilteredImage = (cv2.blur(tigerImageWithNoise, (5, 5))).astype(np.uint8)
#save the image resulted
cv2.imwrite('tigerMeanFilteredImage.jpg', tigerMeanFilteredImage)
#show the results and original image
cv2.imshow('Noisy tiger Image', tigerImageWithNoise)
cv2.imshow('Mean filtered tiger Image', tigerMeanFilteredImage)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Figure 7: Part 4 code

5- Add salt and pepper noise (noise-density=0.1) to the original image and then Apply a 7 by 7 median filter to the noisy-image and show both images.

Figure 8: Tiger Image after a salt and pepper noise has been added to the original image

In the figure above, it is obvious that there is a noise has been added on the image, and this noise named by salt and pepper noise, so it was distributed with noise density 10%, and it's mean that 10% of the pixels in image were affected by this noise, so this type of noise degrades the quality of the image, and it's make abrupt and random intensity changes in the image, and this type of noise may be problematic when the case where the image quality is important.

So afterwards, the median filter has been applied as shown on the figure below:

Figure 9: Tiger image after adding SP nose and median filter applied on it

The median filter has been used to avoid the effects of salt and pepper noise, so the median filter is effective to remove the salt and pepper noise by replacing each pixel in the image with the median value of its local neighborhood pixel. Therefore, the filter is effective in preserving edges and details while removing the extreme values introduced by noise, and In contrast with mean filter, the median filter is effective to save edges and details, but the mean filter make a loss in details of the image.

```python
#Part 5
import numpy as np
import cv2
#read the image
tigerImage = cv2.imread('tiger.jpg', cv2.IMREAD_GRAYSCALE)
#get 10% of over all pixel in the image by multiplying the size of the original image with 10%
TenPercentOfPixels = int(0.1 * tigerImage.size)
#get a copy from the original image, to apply salt and pepper noise on 10% from it
tigerImageWithSaltAndPepperNoise = tigerImage.copy()
# generate random indices based on 10% of all pixels in the original image
TenPercentIndices = np.random.choice(np.arange(tigerImage.size), size=TenPercentOfPixels)
# compute number of pixels to set as salt (white) and pepper (black) noise
NumOfSaltPixels = TenPercentOfPixels // 2
# get the first half of the random indices for set salt noise
NumOfSaltIndices = TenPercentIndices[:NumOfSaltPixels]
# get the second half of the random indices for set pepper noise
NumOfPepperIndices = TenPercentIndices[NumOfSaltPixels:]
# put pixels at salt indices with 255 (white)
tigerImageWithSaltAndPepperNoise.flat[NumOfSaltIndices] = 255
# put pixels at pepper indices with 0 (black)
tigerImageWithSaltAndPepperNoise.flat[NumOfPepperIndices] = 0
#filter the image using median filter with kernel size 7x7,
# using MedianBlur function for the image with added noise as shown above
tigerMedianFilteredImage = cv2.medianBlur(tigerImageWithSaltAndPepperNoise, 7)
#save the images , noisy image , and filtered noisy image
#save the images , noisy image , and filtered noisy image
cv2.imwrite('tigerImageWithSalt&PepperNoise.jpg', tigerImageWithSaltAndPepperNoise)
cv2.imwrite('tigerMedianFilteredImage.jpg', tigerMedianFilteredImage)
#show result to show the images with noise and filtered image
cv2.imshow('tiger Noisy Image with Salt and Pepper Noise', tigerImageWithSaltAndPepperNoise)
cv2.imshow('tiger Median Filtered Image', tigerMedianFilteredImage)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 10: Part 5 code

6- Apply a 7 by 7 mean filter to the salt and pepper noisy-image and show the result. Discuss the results in your report.

Figure 11: Tiger image with SP noise has been filter with mean filter

It should be noticed that when image with salt and pepper noise has been applied to the mean filter, the filter make a smoothing that averages the pixel values in a local neighborhood, so it's meanly used to Nosie reduction, so the mean filter make the image with salt and pepper noise has been reduced the effect of the noise by smoothing out the abrupt change in the pixel values. Moreover, when compare the median filter with the mean filter, the mean filter tends to be less effective in saving details and edges, it may result in blurring the image, especially in the pixels that have high frequency from the noise.

Note that the median filter is generally more adept at retaining details and edges while effectively reducing salt and pepper noise.

```python
# Part6
import numpy as np
import cv2
#read the image with salt and pepper noise
tigerImageWithSP = cv2.imread('tigerImageWithSalt&PepperNoise.jpg', cv2.IMREAD_GRAYSCALE)
#filter the image read above using mean or averaging fiter with kernel size 7x7, using blur functiom
tigerImagewithMeanFilterForSP = cv2.blur(tigerImageWithSP, (7, 7))
#save the filtered image
cv2.imwrite('tigerImageSPwithMeanFilter.jpg',tigerImagewithMeanFilterForSP)
#show the result of the code
cv2.imshow('tiger image with salt and pepper with mean filter', tigerImagewithMeanFilterForSP)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 12: Part 6 code

7- Apply a Sobel filter to the original image and show the response (don't use ready functions to do this part).

Figure 13: Tiger image after applying Sobel filter

Figure 14: Sobel X for tiger image

Figure 15: Sobel Y for tiger image

Sobel X and Sobel Y as shown in the figures above represents the Gradients, Gx and Gy, after summation and multiplying.

In this part, Sobel filter has been applied on the tiger image, by using the convolution with this equations as follows:

Assume O is the original image, so the gradients of the horizontal X and vertical Y:

$G_x =$

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * O$$

Figure 16: Gx in Sobel

$G_Y =$

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * O$$

Figure 17: Gy in Sobel

And the above equation, is the convolution operations with the horizontal and vertical changes with the Sobel Kernels
And if we need to get the response from the Sobel filter, we can do it using the magnitude gradient, as follows:

$$G = \sqrt{Gx^2 + Gy^2}$$

And we can find the orientation or direction of edge by using Arctangent, as follows:

$$\Theta = \tan^{-1}\left(\frac{Gy}{Gx}\right)$$

In the code, the Sobel filter has been applied to the tiger image, without using pre-built functions, and the function ConvolveToSobel () has been used for the convolution operation with the gradients. However, the result image from the filter was an image with highlighted edges and boundaries as shown in figure above.

The mechanism used by Sobel filter, it is meanly enhances edges in the image by ensuring regions with huge changes in intensity. Usually, the resulting image contains lower value in smoother area and high values in areas has an edges. While Sobel filter are very effective for edge detection, but also it is sensitive to the noise in the image, so the noisy regions in the pixels may result false positives in edge detection, and the normalization has been used to detect the small details in the face of the tiger, if the normalization hasn't been used, the small details will lost.

```python
# Part 7
import cv2
import numpy as np
#convolution function with Sobel filter , using the graidents Gx and Gy,
# and convolve the image with  the gradients and get the magnitude values
# using the square root of the squared value, and normalzie the magnitude and convert it to 8-bit datatype
def ConvolveToSobelFilter(tigerImage):
    #get the image size Horizontal and Vertical using shape parameter
    Vertical, Horizontal = tigerImage.shape
    # create an array for magnitude result from sobel filter with zero values
    # and the data type 64-bit float datatype
    MagnitudeG = np.zeros_like(tigerImage, dtype=np.float64)
    #Matrix for Vertical Gy
    GradientY = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    #Matrix for Horizontal Gx
    GradientX = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    #First Loop for Vertical pixels
    for i in range(1, Vertical - 1):
        #Second Loop for Horizontal pixel
        for j in range(1, Horizontal - 1):
            # get a 3x3 neghiborhood aroud the pixels i and j in the original image
            #and apply the sobel filter in the horizontal direction to the neghiborhood
            GxFiltered = np.sum(tigerImage[i - 1:i + 2, j - 1:j + 2] * GradientX)
            # get a 3x3 neghiborhood aroud the pixels i and j in the original image
            # and apply the sobel filter in the vertical direction to the neghiborhood
            GyFiltered = np.sum(tigerImage[i - 1:i + 2, j - 1:j + 2] * GradientY)
            #calculate the magnitude of Gx and Gy at each pixels i and j
            MagnitudeG[i, j] = np.sqrt(GxFiltered ** 2 + GyFiltered ** 2)

    #Normalize the magnitude to be in range from 0 to 255
    MagnitudeG = (MagnitudeG / MagnitudeG.max()) * 255
    #converts the magnitude to 8-bit datatype
    MagnitudeG = np.uint8(MagnitudeG)

    return MagnitudeG

# read the original image
tigerImg = cv2.imread('tiger.jpg', cv2.IMREAD_GRAYSCALE)
#apply convolution function with sobel filter
FinalResultForSobel = ConvolveToSobelFilter(tigerImg)
#save the result of the sobel filter
cv2.imwrite('tigerImageWithSobelFilter.jpg', FinalResultForSobel)
#show the original image and resulted image from sobel fiter
cv2.imshow('tiger Image', tigerImg)
cv2.imshow('tiger Sobel Filtered Image', FinalResultForSobel)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 18: Part 7 code

Question#2: Write a function that convolves an image with a given convolution filter

function [output_Image] = myImageFilter(Input_image, filter)

Your function should output image of the same size as that of input Image (use padding). Test your function (on attached images "`House1.jpg`" and "`House2.jpg`") and show results on the following Kernels.

Figure 19: House 1 Original Image

Figure 20: House 2 Original Image

The above figures was applied by the following filters:

Firstly, the padding has been used in the convolution function for ensuring that the output image has the same size as the input image, and for saving the spatial dimension of the inputs, and prevent information loss at the image borders, so the padding was applied on both dimensions based on half of the kernel size, and this was a reason to ensure that the convolution operation can be applied to pixels at the edges of the image, and in my code the padding was done with zeros, as specified in "mode= 'constant'" parameter.

Secondly, the convolution was done using a new array was created with zeros and same directions as the input image, this array was stored the result of the convolution. Furthermore, two nested loops has been iterated over each pixel in the original image. For each pixel, a region of the padded image was selected based on the dimensions of the kernel, the element-wise multiplication of the kernel and the selected region of the image is calculated, then the summation of the element-wise multiplication is computed and the result has been stored in the corresponding pixel.

```
def FunctionForPaddingAndConvolve(Image, Kernels):
    # get the size of the convolution kernel using width and height or row and column using shape parameter
    RowK, ColK = Kernels.shape
    # get the size of the image using width and height or row and column using shape parameter
    RowI, ColI = Image.shape
    # paad the image with zeros to ease the  convolution on the edges that have not enough size as kernel size
    PaddingResult = np.pad(Image, ((RowK // 2, RowK // 2), (ColK // 2, ColK // 2)), mode='constant')
    # create an array for the convolved image with zero values and data type 64-bit float
    EditedImage = np.zeros_like(Image, dtype=np.float64)
    # first loop for horizontal pixels
    for i in range(RowI):
        #second loop for vertical pixels
        for j in range(ColI):
            # calculate the convolution at each pixel using the kernel specified
            EditedImage[i, j] = np.sum(PaddingResult[i:i + RowK, j:j + ColK] * Kernels)

    return EditedImage
```

Figure 21: Padding and Convolution function

Finally, the function results the convolved image, which shows the result of the convolution operation on the input image with any of kernels below:

1- Averaging Kernel (3×3 and 5×5)

Figure 22: House 1 with Mean Filter 3 x 3

Figure 23: House 1 with mean filter 5x5

Figure 24: House 2 with mean filter 3x3

Figure 25: House 2 with mean filter 5x5

So as shown in the figures above, the 3x3 mean kernel resulted in a slight effect on the images, and the reason is to the averaging of pixel values in a small neighborhood, and the details in the images may become less pronounced and the overall image might appear smoother.

In other hand, the 5x5 mean kernel resulted in produce a more pronounced blurring effect compared to the 3x3 kernel, and this is because a larger neighborhood with an averaging process, and there were fine details in the image may be further suppressed, resulting in a more globally smoothed appearance.

```python
# read the House_1 image
ImageOfHouse1 = cv2.imread('House1.jpg', cv2.IMREAD_GRAYSCALE)
# read the House_2 image
ImageOfHouse2 = cv2.imread('House2.jpg', cv2.IMREAD_GRAYSCALE)
#show the original image of House_1
cv2.imshow('House1 Original Image', ImageOfHouse1)
#show the original image of House_2
cv2.imshow('House2 Original Image', ImageOfHouse2)
#mean filter for images using mean filter or averaginf filter with kernel size 3x3
MeanFilter3BY3 = np.ones((3, 3)) / 9.0
#mean filter for images using mean filter or averaginf filter with kernel size 5x5
MeanFilter5BY5 = np.ones((5, 5)) / 25.0
#result of mean filtering on House_1 and House_2 images by 3x3 and 5x5 kernel size
ResultOfMeanFilter3BY3House_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, MeanFilter3BY3)
ResultOfMeanFilter5BY5House_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, MeanFilter5BY5)
ResultOfMeanFilter3BY3House_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, MeanFilter3BY3)
ResultOfMeanFilter5BY5House_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, MeanFilter5BY5)
```

Figure 26: mean filter code

2- Gaussian Kernel (σ = 1,2,3) Use $(2\sigma +1)\times(2\sigma +1)$ as size of Kernel (You may write a separate function to generate Gaussian Kernels for different values of σ). Discuss the results in your report.

```python
def GaussianLambada(Parameter, Volume):
    # creat an array showing a Gaussian kernel by  numpy  fromfunction()
    kernel = np.fromfunction(
        # create the function to generate each element of the kernel based on it x and y
        lambda x, y: (1 / (2 * np.pi * Parameter ** 2)) * np.exp(
            -((x - (Volume - 1) / 2) ** 2 + (y - (Volume - 1) / 2) ** 2) / (2 * Parameter ** 2)),
        # Specify the shape of the 2D array (Volume x Volume) to create the kernel
        (Volume, Volume)
    )

    # normailze the kernel by dividing each element on the sum of all elemnts on the kernel
    return kernel / np.sum(kernel)
```

Figure 27: Gaussian equation code

In this part, for generate Gaussian kernel function GaussianLambda () has been created, and the result of the kernel shown below:

Figure 28: House 1 with Gaussian kernel and sigma=1

Figure 29: House 1 with Gaussian kernel and sigma=2

Figure 30: House 1 with Gaussian kernel and sigma=3

Figure 31: House 2 with Gaussian kernel and sigma=1

Figure 32: House 2 with Gaussian kernel and sigma=2

Figure 33: House 2 with Gaussian kernel and sigma=3

As shown in the figures above, with sigma value: 1, 2, and 3, the increasing in sigma value in the Gaussian kernel was leaded into increased blurring, so a higher value of sigma corresponds to a wider distribution, so it resulted in a larger region of effect for each pixel. Therefore, when sigma increased, the images was became progressively smoother, with fine details being increasingly attenuated.

```
# create an array of parameter values (sigma)
ParameterValues = [1, 2, 3]
# loop on the parameter values and perform convolution with equations give in the question
for Pixel, Parameter in enumerate(ParameterValues):
    # compute Gaussian kernel with equation 2*sigma + 1 using Lambda function  with sigma 1 and 2 and 3
    GaussianK = GaussianLambada(Parameter, 2 * Parameter + 1)
    # apply convolution with padding to House_1 and House_@ using the Gaussian kernel with sigma 1 and 2 and 3
    ResultOfGaussianOfHouse1 = FunctionForPaddingAndConvolve(ImageOfHouse1, GaussianK)
    ResultOfGaussianOfHouse2 = FunctionForPaddingAndConvolve(ImageOfHouse2, GaussianK)
    # show images with applied Gaussian kernel
    cv2.imshow(f'House 1 with Gaussian Kernel with sigma{Parameter}', ResultOfGaussianOfHouse1)
    cv2.imshow(f'House 2 with Gaussian Kernel with sigma{Parameter}', ResultOfGaussianOfHouse2)
    # save images with applied Gaussian kernel
    cv2.imwrite(f'House1WithGaussianKernelWithSigma{Parameter}.jpg', ResultOfGaussianOfHouse1)
    cv2.imwrite(f'House2WithGaussianKernelWithSigma{Parameter}.jpg', ResultOfGaussianOfHouse2)


cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 34: Gaussian Kernel code

3- Sobel Edge Operators: $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ and $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

Figure 35: House 1 with Sobel kernel

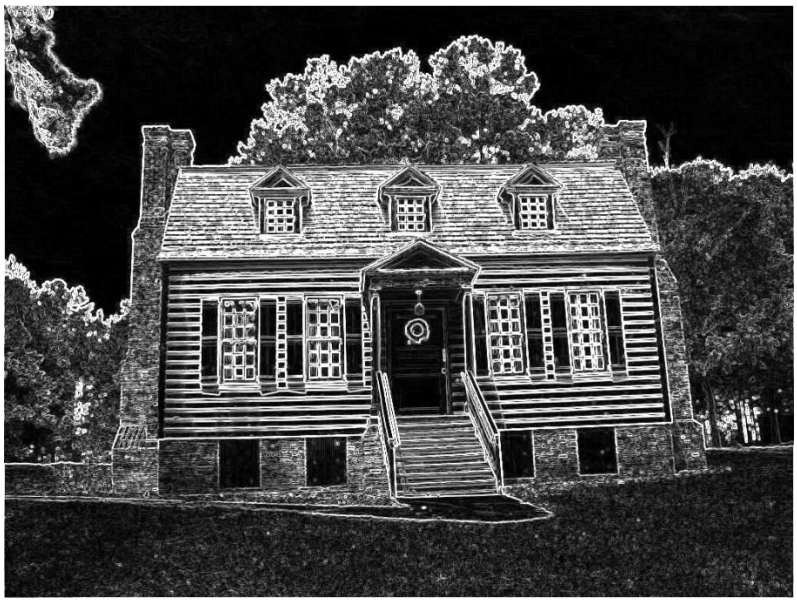

Figure 36: House 2 with Sobel filter

So here using vertical and horizontal sobel operator for edge detection, the sobel operators are design to detect edges as mentioned in the previous question, so it detect the edges by emphasizing the intensity changes in the vertical and horizontal directions, and the sobel compared with other operators, Sobel tends to give more weight to the center pixel in each kernel, and the weighting of the pixels can result in changes in the sensitivity of the filters to different types of edges and noise.

4- Prewitt Edge Operators:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & 1 \end{bmatrix}$$

Figure 37: House 1 with Prewitt kernel

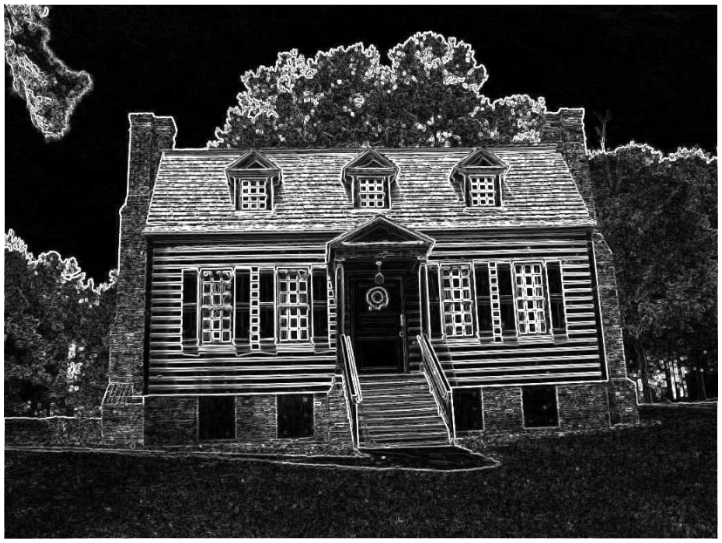

Figure 38: House 2 with Prewitt kernel

As shown in figure above, the results of the vertical and horizontal Prewitt operators, the resulted images was similar to the Sobel operators, so it is aim to detect edges in the vertical and horizontal directions, and the results for Sobel and Prewitt operators very similar, but the Prewitt gives equal weight to all pixels in the kernel, so as mentioned above, the weighting affects the sensitivity of the filter to different types of edges and noise.

```
#Matrix for Horizontal Gx
Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
#Matrix for Vertical Gy
Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
#Apply Convolution with Gx and Gy for each image
ConvolveGxWithImageHouse_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, Gx)
ConvolveGyWithImageHouse_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, Gy)
ConvolveGxWithImageHouse_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, Gx)
ConvolveGyWithImageHouse_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, Gy)
#Matrix for Horizontal Px
Px = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
#Matrix for Vertical Py
Py = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
#Apply Convolution with Px and Py for each image
ConvolvePxWithImageHouse_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, Px)
ConvolvePyWithImageHouse_1 = FunctionForPaddingAndConvolve(ImageOfHouse1, Py)
ConvolvePxWithImageHouse_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, Px)
ConvolvePyWithImageHouse_2 = FunctionForPaddingAndConvolve(ImageOfHouse2, Py)
#calculate magnitude for Sobel filter and Prewitt filter using the square root of the gradients
MagnitudeG_House_1 = np.sqrt(np.power(ConvolveGxWithImageHouse_1,2) + np.power(ConvolveGyWithImageHouse_1,2))
MagnitudeG_House_2 = np.sqrt(np.power(ConvolveGxWithImageHouse_2,2) + np.power(ConvolveGyWithImageHouse_2,2))
MagnitudeP_House_1 = np.sqrt(np.power(ConvolvePxWithImageHouse_1,2) + np.power(ConvolvePyWithImageHouse_1,2))
MagnitudeP_House_2 = np.sqrt(np.power(ConvolvePxWithImageHouse_2,2) + np.power(ConvolvePyWithImageHouse_2,2))
```

Figure 39: Sobel and Prewitt filter code

Question#3: Attached "`Noisyimage1`" and "`Noisyimage2`" are corrupted by salt and paper noise. Apply 5 by 5 Averaging and Median filter and show your outputs. Why Median filter works better than averaging filter?

Figure 40: First Noisy Image

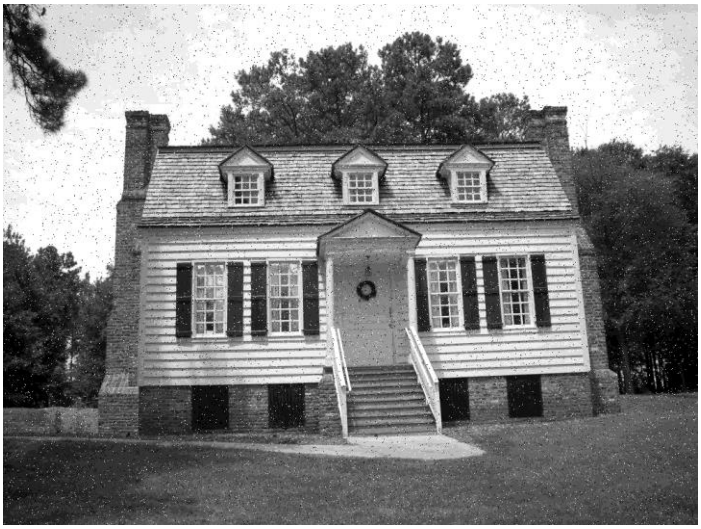

Figure 41: Second Noisy Image

Figure 42: First Noisy Image with Mean Filter

Figure 43: First Noisy Image with median filter

Figure 44: Noisy Image 2 with mean filter

Figure 45: Noisy Image 2 with median filter

As shown in the figures above and after the noisy images have been applied on the mean filter and the median filter, two noisy images has a noise of type salt and pepper, so this noise make an image corruption on random pixels, and it is appears as isolated bright and dark pixels in the image.

And the mean filter or averaging filter, works by taking the average of the pixel values in a 5x5 neighborhood around each pixel in the image, and this filter used for reducing noise by smoothing out the changes in the pixel intensity.

The median filter, works by replacing each pixel with the median value in 5x5 neighborhood pixels, and this filter is fine in removing salt and pepper noise, because it's less sensitive to the outliers.

The figures shown above ensures that the median filter is more robust or reliable to outliers and it saves edges better because it does not blurring the image, but the mean filter when works to reduce the noise, the smoothing out of the edges make some small losses in the details of the image, so the median filter works on remove the noise, but the mean filter work on reduce the noise by smoothing out edges, which make the details has a losses because of blurring.

```
# Q3
import cv2
import numpy as np
#read the images by using cv
NoisyImage1 = cv2.imread('Noisyimage1.jpg', 0)
NoisyImage2 = cv2.imread('Noisyimage2.jpg', 0)
#filtering the images with mean filter and median filter with kernel size 5x5
# using blur() for mean filter and medianblur() for median filter
MeanFilterFirstImg = cv2.blur(NoisyImage1, (5, 5))
MedianFilterFirstImg = cv2.medianBlur(NoisyImage1, 5)
MeanFilterSecondImg = cv2.blur(NoisyImage2, (5, 5))
MedianFilterSecondImg = cv2.medianBlur(NoisyImage2, 5)
#save the images resulted from the filters with jpg extension
cv2.imwrite('NoisyImg1WithMeanFilter.jpg', MeanFilterFirstImg)
cv2.imwrite('NoisyImg1WithMedianFilter.jpg', MedianFilterFirstImg)
cv2.imwrite('NoisyImg2WithMeanFilter.jpg', MeanFilterSecondImg)
cv2.imwrite('NoisyImg2WithMedianFilter.jpg', MedianFilterSecondImg)
#show the resulted images from the filters and the original images
cv2.imshow('Noisy Image 1', NoisyImage1)
cv2.imshow('Noisy Image 1 With Mean Filter', MeanFilterFirstImg)
cv2.imshow('Noisy Image 1 With Median Filter', MedianFilterFirstImg)
cv2.imshow('Noisy Image 2', NoisyImage2)
cv2.imshow('Noisy Image 2 With Mean Filter', MeanFilterSecondImg)
cv2.imshow('Noisy Image 2 With Median Filter', MedianFilterSecondImg)
cv2.waitKey(0)
```

Figure 46: Question 3 code

Question#4: Compute gradient magnitude for attached image "`Q4_Image`" (using built-in sobel gradients function).

Figure 47: Original Image

In this part, the Sobel filter in CV2 library has been used, and the Sobel Gradients successfully computed the gradient magnitude for each pixel in the image, so this step was captured the rate of intensity change, and changing edges and features within the image.

1. Stretch the resulting magnitude (between 0 to 255) for better visualization

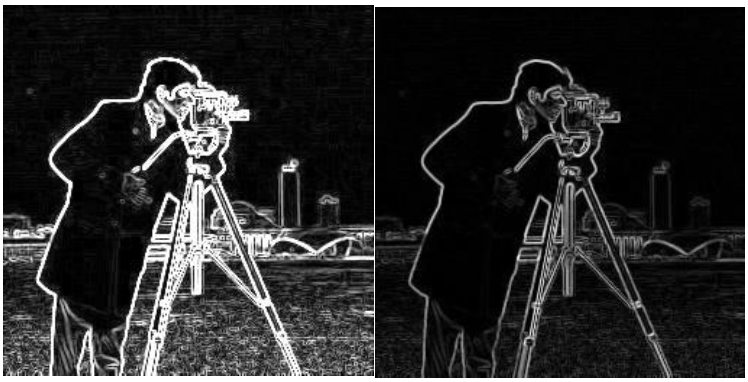

Figure 48: figure (1) Normal figure (2) Stretched Gradient Magnitude

As shown in figures above, the stretching operation of the gradient magnitude has been used to enhance visualization, and the resulting gradient magnitude was stretched to the range from 0 to 255. Therefore, this normalization allows for a clearer representation of intensity changes and facilitates better comparison between different areas or regions of the image.

2. Compute the histogram of gradient magnitude

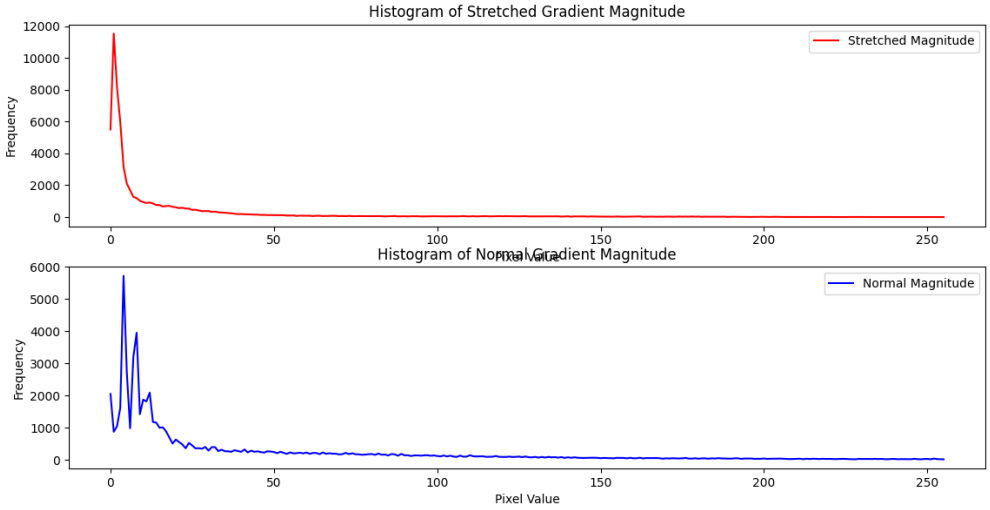

Figure 49: Histogram of the Normal and Stretched gradient magnitude

As shown in the plot above, the histogram of the stretched gradient magnitude provides valuable insights into the distribution of edge strengths across the image, and the peaks in the histogram may give an indication about notable edges or features, while the overall shape can reveal information about the image structure and the content, so the histogram of the stretched magnitude compared with the normal magnitude one enhanced the visualization by concentrating the values within a narrower range, with make changes more obvious, but the normal one may have peak and valleys that are not clearly discernible due to a broader range of values. Moreover the stretched histogram can make peaks more notable, which can help in the identification of important intensity changes, but the normal one may represent strong edges or features but might less pronounced or harder to identify, and the stretched histogram maintains the shape but emphasizes changes, and offers a clearer representation of edge strengths, the normal one has a shape of the original histogram provides insights into overall structure and content of the image.

3. Compute gradient orientation (the angle of gradient vector)

The orientation or direction of gradient by using Arctangent, as follows:

$$\Theta = \tan^{-1}\left(\frac{Gy}{Gx}\right)$$

Note that the orientation was calculated using arctan2 in the np library, so the computation of the gradient orientation was involved the determining the angle of the gradient vector for each pixel, and this information is important for understanding the directionality of edges and features within the image.

4. Compute histogram of gradient orientation (angle between 0 and 2*pi)

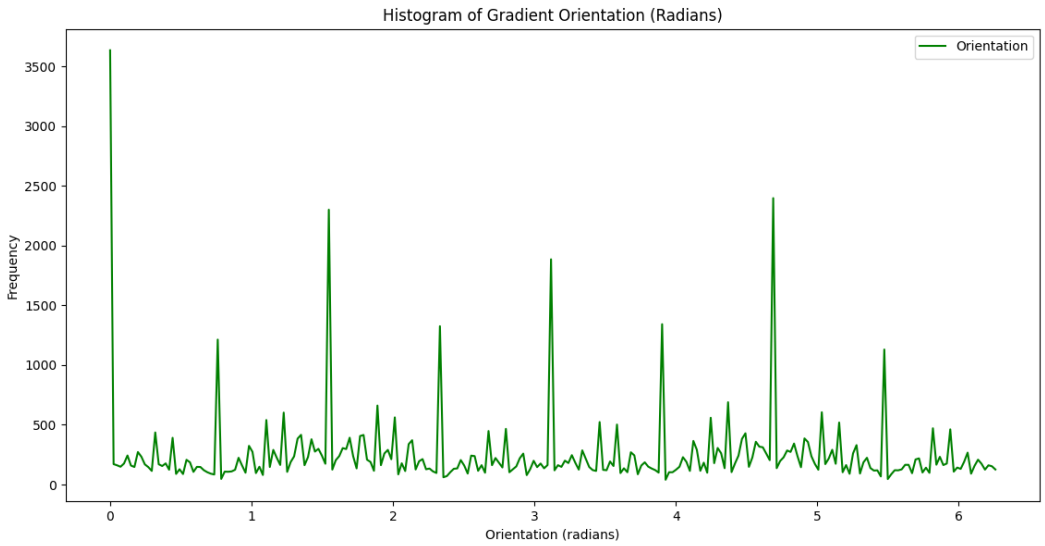

Figure 50: Histogram of the Gradient Orientation

As shown in the plot above, the histogram of gradient angles with range from 0 to 360 degree offers a comprehensive view of the dominating direction of edges in the image, and the peaks in the histogram correspond to dominant orientations, providing insights into the overall structure and patterns present.

```
# read the image
OriginalImage = cv2.imread('Q_4.jpg', cv2.IMREAD_GRAYSCALE)
# apply Sobel gradients for horizontal Gx and vertical Gy with kernel size 3x3
Gx = cv2.Sobel(OriginalImage, cv2.CV_64F, 1, 0, ksize=3)
Gy = cv2.Sobel(OriginalImage, cv2.CV_64F, 0, 1, ksize=3)
# compute the gradient magnitude using square root
NormalMagnitude = np.sqrt(Gx ** 2 + Gy ** 2)
# stretch the magnitude for better visualization
StretchedMagnitude = cv2.normalize(NormalMagnitude, None, 0, 255, cv2.NORM_MINMAX)
# convert magnitude to 8-bit data type
StretchedMagnitude = np.uint8(StretchedMagnitude)
# compute the histogram of the stretched gradient magnitude
HistogramOfStretchedMagnitude = cv2.calcHist([StretchedMagnitude], [0], None, [256], [0, 256])
# compute the histogram of the normal gradient magnitude
HistogramOfNormalMagnitude = cv2.calcHist([np.uint8(NormalMagnitude)], [0], None, [256], [0, 256])
# compute the gradient orientation
GradientOrientation = np.arctan2(Gy, Gx)
# normalize the orientation to the range [0, 360]
OrientationNormalization = (GradientOrientation + 2 * np.pi) % (2 * np.pi)
# flat the array before computing the histogram
FlatToHistogramForOrientation = OrientationNormalization.flatten()
# compute the histogram of the gradient orientation
HistogramOfGradientOrientation = cv2.calcHist([np.float32(FlatToHistogramForOrientation)], [0], None, [256],
                                              [0, 2 * np.pi])
```

Figure 51: Question 4 code

Question5: Load *walk_1.jpg* and *walk_2.jpg* images in Python. Convert them to gray scale and subtract *walk_2.jpg* from *walk_1.jpg*. What is the result? Why?

Figure 52: Image1 for walking people to use it in subtraction

Figure 53: Image2 for walking people to use it in subtraction

So here after subtraction, in two ways, first way Walk_1 – Walk_2 and the second Walk_2 – Walk_1, to show the differences:

Figure 54: First Subtraction

Figure 55: Second Subtraction

It should be noticed that the result from the first subtraction show the pixel-wise intensity differences between Walk_1 and Walk_2, so the pixels has positive values give an indication about the areas where Walk_1 is brighter than Walk_2 and the pixels that has negative value give an indication about the areas where Walk_1 is darker than Walk_2. Furthermore, if the first subtraction, the observation that the features where Walk_1 is brighter than Walk_2, in the second subtraction the observation that the features in Walk_2 is brighter than Walk_1. Moreover, the absolute value for both subtraction operations, given the magnitude of the intensity differences.

```python
#read images using cv2 as
FirstImageWalk_1 = cv2.imread('walk_1.jpg', cv2.IMREAD_GRAYSCALE)
SecondImageWalk_2 = cv2.imread('walk_2.jpg', cv2.IMREAD_GRAYSCALE)
#check if the images exists or not, if not print Error
if FirstImageWalk_1 is None or SecondImageWalk_2 is None:
    print("Error loading images.")
else:
    #if images exists perform and subtraction and reverse subtraction
    # to show the diffrence between two results based on the images and the contnet of the images
    #chech if the images has the same size height and width using shape parameter,
    # if not print error diffrence in dimensions
    if FirstImageWalk_1.shape == SecondImageWalk_2.shape:
        #subtraction between two images using cv2
        result = cv2.subtract(FirstImageWalk_1, SecondImageWalk_2)
        result2 = cv2.subtract(SecondImageWalk_2, FirstImageWalk_1)
        #save and show result of subtraction and reverse subtraction
        cv2.imwrite('SubtractionResult.jpg', result)
        cv2.imwrite('ReversedSubtractionResult.jpg', result2)
        cv2.imshow('Subtraction Result', result)
        cv2.imshow('Inverse Subtraction Result', result2)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    else:
        #print for not equal sizes of images
        print("Images have different dimensions.")
```

Figure 56: Question 5 code

Question#6: Apply canny edge detector on the "`Q_4.jpg`" using OpenCV function "`Canny`". Test different values of '`Threshold`'.

Figure 57: Question 4 image before applying canny edge detector

Figure 58: Image when canny applied with Threshold in range from 50 to 100

Figure 59: Image when canny applied with Threshold from 100 to 150

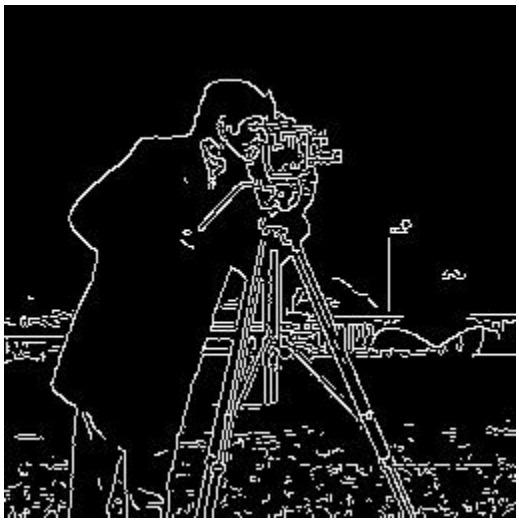

Figure 60: Image when canny applied with Threshold from 150 to 200

It should be noticed from the figures that Canny is an edge detector and use the threshold mechanism to detect bigger number from the edges in the image based on the range of the threshold. Firstly, when the threshold was from 50 to 100, the edges have detected more inclusively, and there were high number of edges visible in the resulting image, while this threshold values show a changes in the intensity, also it tends to introduce noise and may include not important details. Secondly, when the threshold was from 100 to 150, this range was made a balance between the sensitivity and the specificity, and it was provided a sensible number of well-defined edges, and this range of the threshold works to offer a good compromise and save important contours while reducing the effect of the noise on the image. Finally, when the threshold was from 150 to 200, it was made the result more moderate edge detection mechanism, and it ensures only the pronounced edges. However, this may led to a loss of some small details and might not show a changes or transition on the intensity.

So lower threshold values result in more comprehensive edge detection but at the expense of increased noise and false positives, and higher threshold values result in reduce the noise , but may miss certain edges and it is possible to affect the completeness of the detected features. Moreover, canny edge detector has a sensitivity to threshold values that highlights the need for careful parameter tuning based on properties of the input image.

```python
#read the image using cv
OriginalImage = cv2.imread('Q_4.jpg', cv2.IMREAD_GRAYSCALE)
#apply Canny edge detector using ranges of thresholds to show the strenghts of edges in each range:
#1- Low Threshold: 50-100, the result discussed in the report
EdgeDetctorLow = cv2.Canny(OriginalImage, threshold1=50, threshold2=100)
#2-Medium Threshold: 100-150, the result discussed in the report
EdgeDetctorMedium = cv2.Canny(OriginalImage, threshold1=100, threshold2=150)
#High Threshold: 150-200, the result discussed in the report
EdgeDetctorHigh = cv2.Canny(OriginalImage, threshold1=150, threshold2=200)
#show the original image to compare
cv2.imshow('Original Image', OriginalImage)
cv2.waitKey(0)
cv2.destroyAllWindows()
#show the result of Canny Edge detctor in each case of threshold ranges
cv2.imshow('Edges (Low Threshold)', EdgeDetctorLow)
cv2.imshow('Edges (Medium Threshold)', EdgeDetctorMedium)
cv2.imshow('Edges (High Threshold)', EdgeDetctorHigh)
#save the result of Canny Edge detctor in each case of threshold ranges
cv2.imwrite('EdgesLowThreshold.jpg', EdgeDetctorLow)
cv2.imwrite('EdgesMediumThreshold.jpg', EdgeDetctorMedium)
cv2.imwrite('EdgesHighThreshold.jpg', EdgeDetctorHigh)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 61: Question 6 code

(Good Luck)