Faculty of Engineering and Technology
Electrical and Computer Engineering Department

ENCS5343 Computer Vision
Assignment # 1

Student name: **Basheer Arouri**

Student ID: **120114**

Table Of Figures

Q1:

Part 1: Showing The Image

Code:

```python
import random
import cv2
import numpy as np



# ----------------------------------First Part----------------------------------
# Load the image
image_path = "image.jpg"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
cv2.imshow("256*256 bits size and 8-bits grey level Image", image)
cv2.waitKey(0)
copy_image = np.copy(image)  # Copy the image for the last part

```

Figure 1 Q1 Part 1 Code

Input and Output Images:

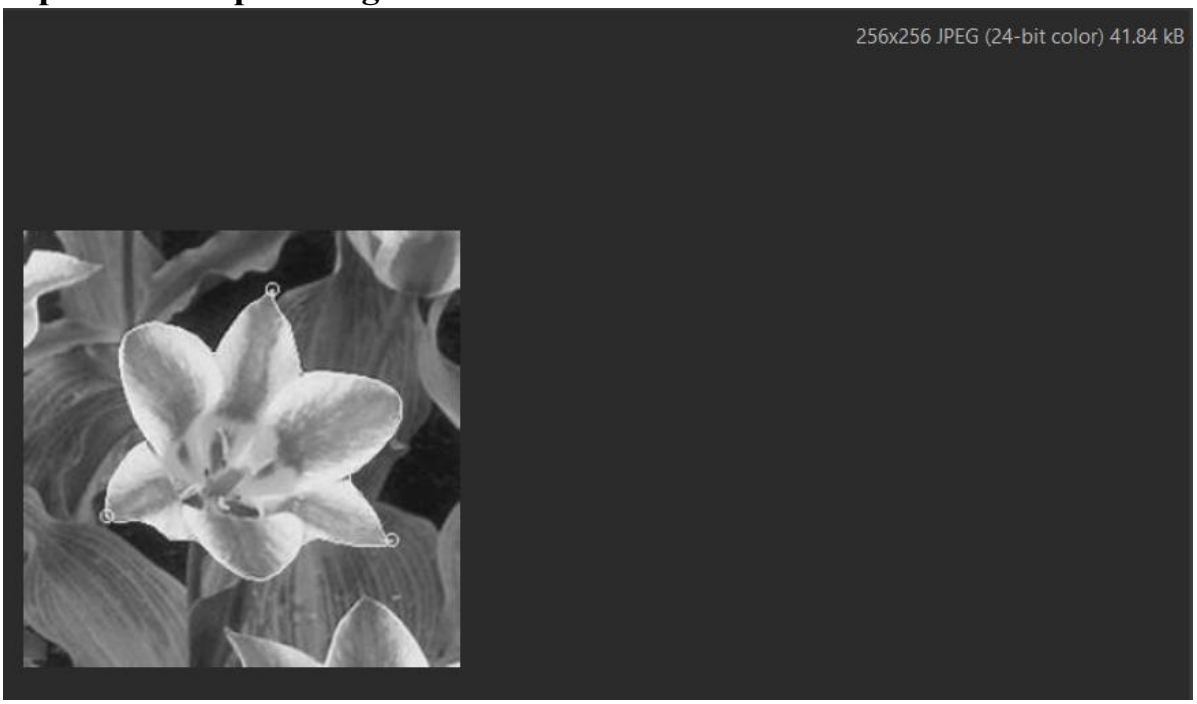

Figure 2 Q1 Part 1 Input Image

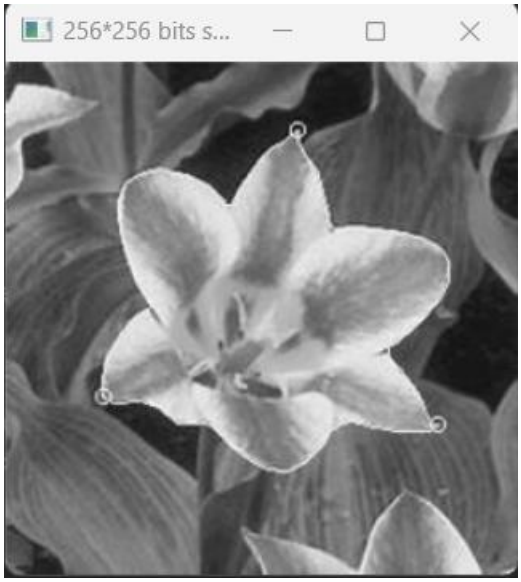

Figure 3 Q1 Part 1 Output Image

Explanation:
In this part, I just loaded the 256*256 pixels image, converted it to gray scale, and showed it by openCV package in python.

Part 2: Applying Power Low Transformation

Code:

```
# ---------------------------------Second Part---------------------------------
# Apply the power-law transformation
c = 1
gamma = 0.4  # Define the tuning parameters for this equation s = cr^gamma -----> where c is constant, r is the
# input pixel and gamma is the gamma correction
image_after_normalized = image.astype(np.float32)/255.0
image_after_power_low = c * np.power(image_after_normalized, gamma)  # s = c*r^gamma
image_after_power_low = (image_after_power_low * 255).astype(np.uint8)  # Convert the result to uint8 (
# standard image format) -----> The output image will be grayed, because we have gamma = 0.4 (less than 1 ---> Log)
cv2.imshow("Image With Power Low Transformation ----> gamma = 0.4", image_after_power_low)
cv2.waitKey(0)
```

Figure 4 Q1 Part 2 Code

Input and Output Images:

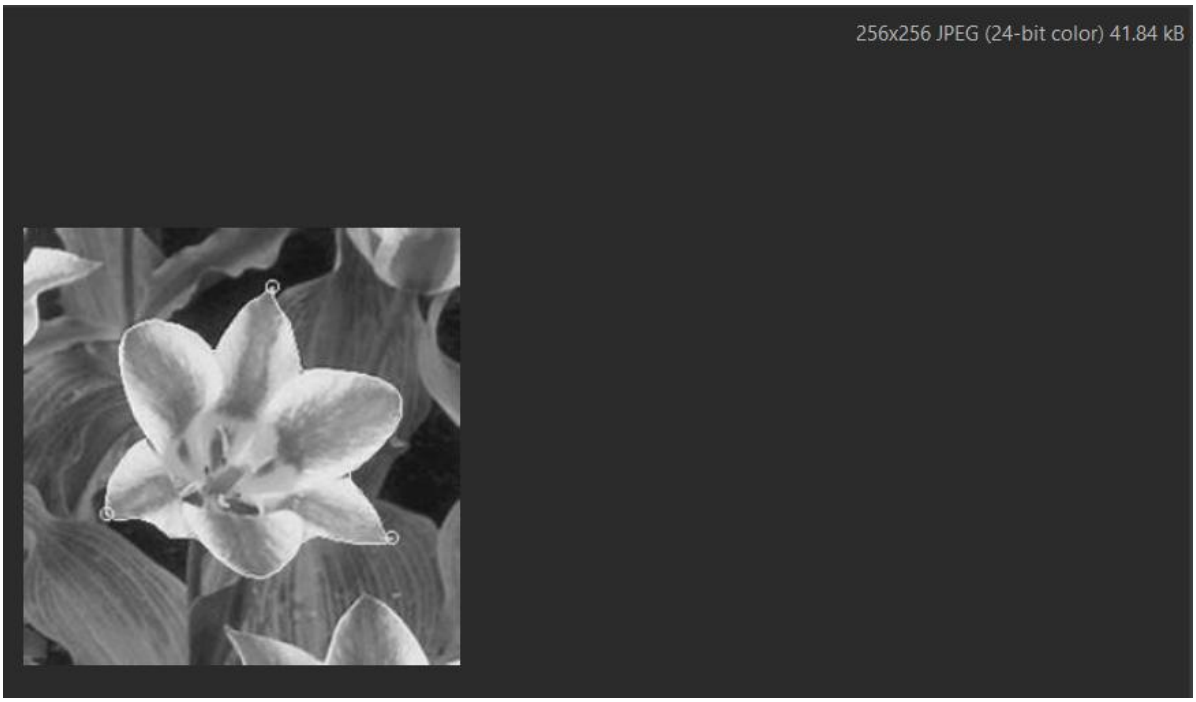

Figure 5 Q1 Part 2 Input Image

Figure 6 Q1 Part 2 Output Image

Explanation:

In this part, I applied a power law transformation with gamma = 0.4, which is less than 1, so the output has been more gray. I normalized the the pixels values with min-max normalization from range [0,255] to [0,1] in order to apply the power law transformation, after that I converted them again to range [0,255].

Part 3: Adding Gaussian Noise to the Image

Code:

```
# ---------------------------------Third Part-----------------------------------
mean = 0  # Mean = 0 for the Gaussian noise
variance = 40  # Variance for the gaussian filter, by default the mean = 0
gaussian_noise = np.random.normal(mean, np.sqrt(variance), image.shape)
image_after_gaussian_noise = image.astype(np.float32) + gaussian_noise
image_after_gaussian_noise = np.clip(image_after_gaussian_noise, 0, 255)
image_after_gaussian_noise = np.uint8(image_after_gaussian_noise)
cv2.imshow("Image With Gaussian Noise ----> Variance = 40 And Mean = 0", image_after_gaussian_noise)
cv2.waitKey(0)
```

Figure 7 Q1 Part 3 Code

Input and Output Images:

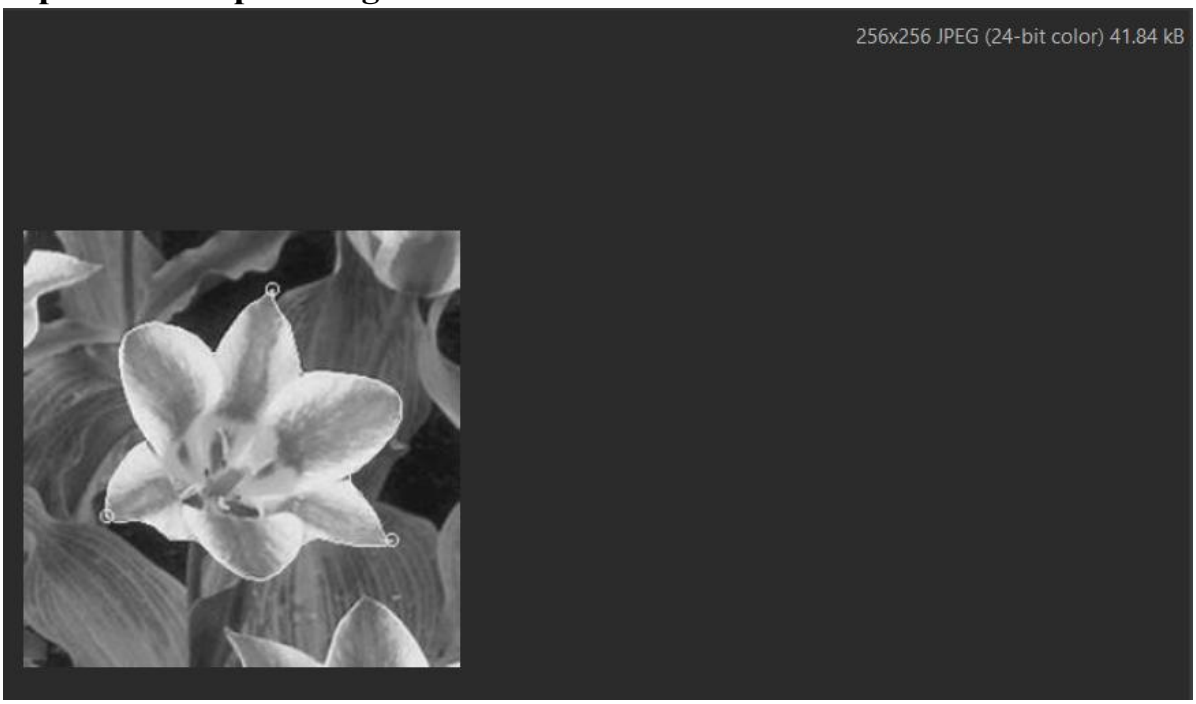

Figure 8 Q1 Part 3 Input Image

Figure 9 Q1 Part 3 Output Image

Explanation:

In this part, I added to the original image a normal (Gaussian) noise with mean = 0 and variance = 40. I used some ready functions in openCV package and in numpy package. **np.clip(0,255)** used to eliminate the values to exceeds the range (0,255) and **np.unit8()** used to convert the floating numbers gutted from np.clip() to unsigned integer values for the range [0,255].

Part 4: Applying Box/Averaging Filter

Code:

```
# ---------------------------------Forth Part-----------------------------------
kernel_size = (5, 5) # Kernel Size = 5 * 5
image_after_box_filter = cv2.boxFilter(image_after_gaussian_noise, -1, kernel_size, normalize=True)
cv2.imshow("Image With Box Filter ----> Kernel size = 5*5", image_after_box_filter)
cv2.waitKey(0)
```

Figure 10 Q1 Part 4 Code

Input and Output Images:

Figure 11 Q1 Part 4 Input Image

Figure 12 Q1 Part 4 Output Image

Explanation:

In this part, I applied a box/averaging filter with kernel size = 5. I did that by ready function in openCV called boxFilter, gave it the input image, which it is the noisy image from part 3, -1 for the depth, kernel size which = 5 and finally set the normalization to true to do normalization after getting the summation for each kernel pixel in the original image multiplied dot product with box kernel.

Part 5: Adding Solt-Pepper Noise and Applying Median Filter

Code subpart 1：

```python
# ---------------------------------Fifth Part-----------------------------------

#Adding salt and pepper noisy data
salt_probability_of_the_image = 0.1
pepper_probability_of_the_image = 0.1

number_of_salt_noises = int(salt_probability_of_the_image * image.size)
number_of_pepper_noises = int(pepper_probability_of_the_image * image.size)

salts_pixels = [[]]
pepper_pixels = [[]]

width = image.shape[0]
height = image.shape[1]

for i in range(0, number_of_salt_noises):
    salt_pixel_x = random.randint(0, width-1)
    salt_pixel_y = random.randint(0, height-1)

    pepper_pixel_x = random.randint(0, width-1)
    pepper_pixel_y = random.randint(0, height-1)

    image[salt_pixel_x][salt_pixel_y] = 255
    image[pepper_pixel_x][pepper_pixel_y] = 0
```

```python
cv2.imshow("Image With Salt and Pepper Noise", image)
cv2.waitKey(0)
```

Figure 13 Q1 Part 5 Subpart 1 Code

Input and Output Images subpart 1:

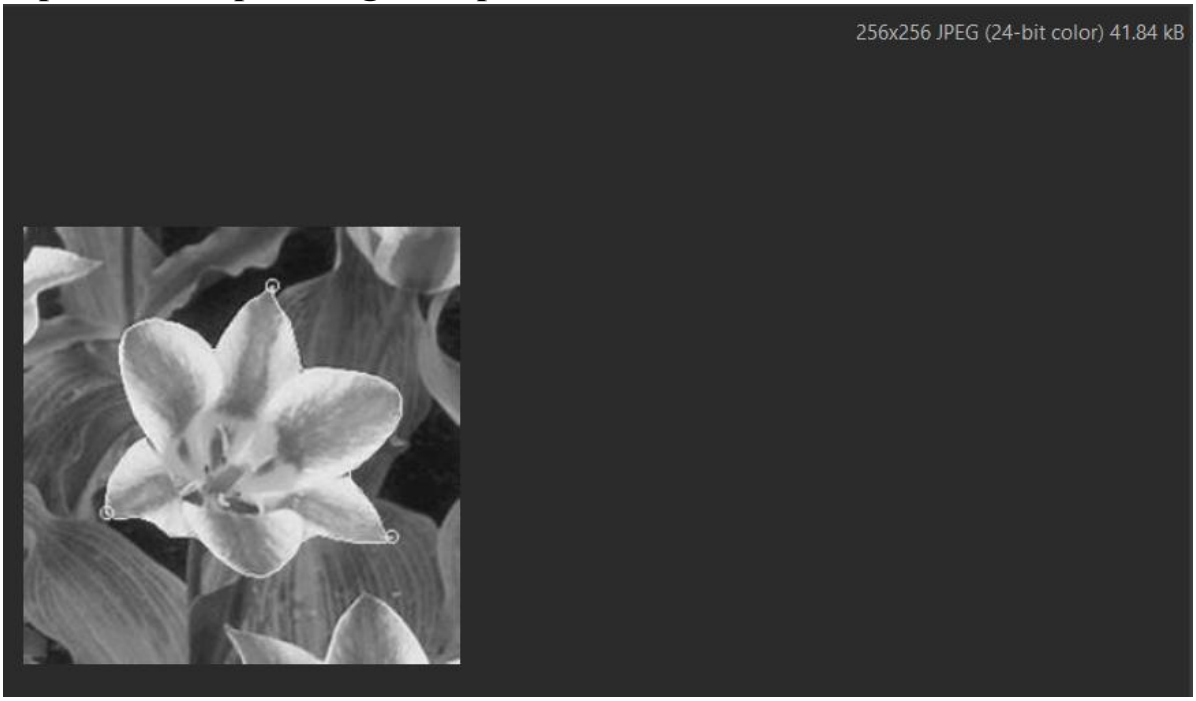

Figure 14 Q1 Part 5 Subpart 1 Input Image

Figure 15 Q1 Part 5 Subpart 1 Output Image

Explanation subpart 1:

In this part, I added to the original image solt-pepper noise, and I added them with density = 0.1 of the total pixels in the image. Salt = 255 and pepper = 0.

Code subpart 2:

```
# Remove (Not Reduce) the salt and pepper noisy data by the median filter
kernel_size = 7
image_after_median_filter = cv2.medianBlur(image, kernel_size)
cv2.imshow("Image With Median Filter ----> 7*7 Kernel Size", image_after_median_filter)
cv2.waitKey(0)
```

Figure 16 Q1 Part 5 Subpart 2 Code

Input and Output Images subpart 2:

Figure 17 Q1 Part 5 Subpart 2 Input Image

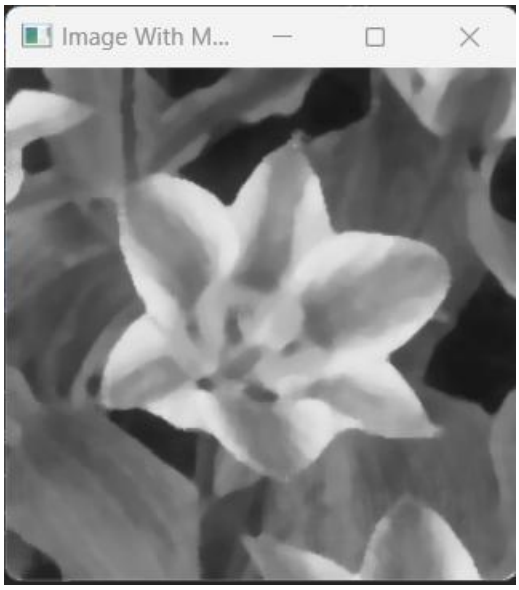

Figure 18 Q1 Part 5 Subpart 2 Output Image

Explanation subpart 2:

In this part, I applied a median filter to the noisy image with solt and pepper, we can see briefly how the salt and pepper noisy data are disappear, and this is the advantage of the median filter, better than averaging filter for salt and pepper noise, for two reasons: you take the median for the pixels and median is more trust than the mean with noisy data, you replace the salt-pepper noise with median, thus you remove it, not reduce it.

Part 6: Applying 7x7 Mean filter To The Noisy Image

Code 1:

```python
# ---------------------------------Sixth Part-----------------------------------
kernel_size = (7, 7)
image_after_box_filter_with_solt_and_pepper = cv2.boxFilter(image, -1, kernel_size)
cv2.imshow("Image With Box Filter And Solt And Pepper Noise ----> Kernel size = 7*7",
           image_after_box_filter_with_solt_and_pepper)
cv2.waitKey(0)
```

Figure 19 Q1 Part 6 Code

Input and Output Images:

Figure 20 Q1 Part 6 Input Image

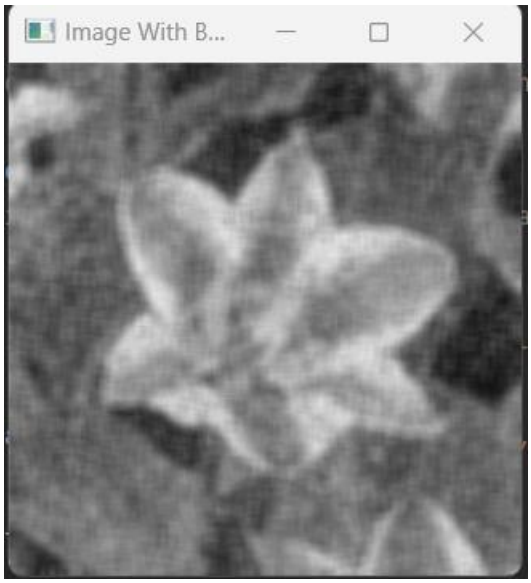

Figure 21 Q1 Part 6 Output Image

Explanation:
In this part, I applied a mean filter with size = 7x7 to the noisy image with salt-pepper noise, we can see briefly that the problem has enlarged, because we do not take care of the salt-pepper noise, just we get the average value of the pixels, thus, the salt-pepper noisy data remain, and the image has been blurred.

Part 7: Applying a Sobel filter to the original image

Code:

```
# ---------------------------------Seventh Part-----------------------------------

sobel_X_kernel = np.array([-1, -2, -1, 0, 0, 0, 1, 2, 1])
sobel_Y_kernel = np.array([-1, 0, 1, -2, 0, 2, -1, 0, 1])

# Add one row of zeros above row number 0
image_with_zeros_above = np.vstack([np.zeros((1, copy_image.shape[1]), dtype=np.uint8), copy_image])

# Add one row of zeros below row number 1
image_with_zeros_below = np.vstack([image_with_zeros_above, np.zeros((1, copy_image.shape[1]), dtype=np.uint8)])

# Add one column of zeros to the left of column number 0
image_with_zeros_left = np.hstack(
    [np.zeros((image_with_zeros_below.shape[0], 1), dtype=np.uint8), image_with_zeros_below])

# Add one column of zeros to the right of column number 1
image_with_zeros_both_sides = np.hstack(
    [image_with_zeros_left, np.zeros((image_with_zeros_left.shape[0], 1), dtype=np.uint8)])
```

```
rows = image_with_zeros_both_sides.shape[0]
columns = image_with_zeros_both_sides.shape[1]

height = copy_image.shape[0]
width = copy_image.shape[1]
```

```
for row in range(1, rows - 1):
    for column in range(1, columns - 1):
        # Do a filter for the current pixel
        first_row = image_with_zeros_both_sides[row - 1][column - 1:column + 2]
        second_row = image_with_zeros_both_sides[row][column - 1:column + 2]
        third_row = image_with_zeros_both_sides[row + 1][column - 1:column + 2]

        kernel_image_for_this_pixel = np.concatenate([first_row, second_row, third_row])

        # Do a dot product between the current filter for this pixel and the sobel x
        gx = np.dot(sobel_X_kernel, kernel_image_for_this_pixel)
        gy = np.dot(sobel_Y_kernel, kernel_image_for_this_pixel)

        G[row - 1, column - 1] = (gx ** 2 + gy ** 2) ** 0.5

#Normalize the Gradient
sobel_mag_image = cv2.normalize(G, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)
```

```
cv2.imshow('Normalized Gradient', cv2.convertScaleAbs(sobel_mag_image))
cv2.waitKey(0)
```

Figure 22 Q1 Part 7 Code

Input and Output Images:

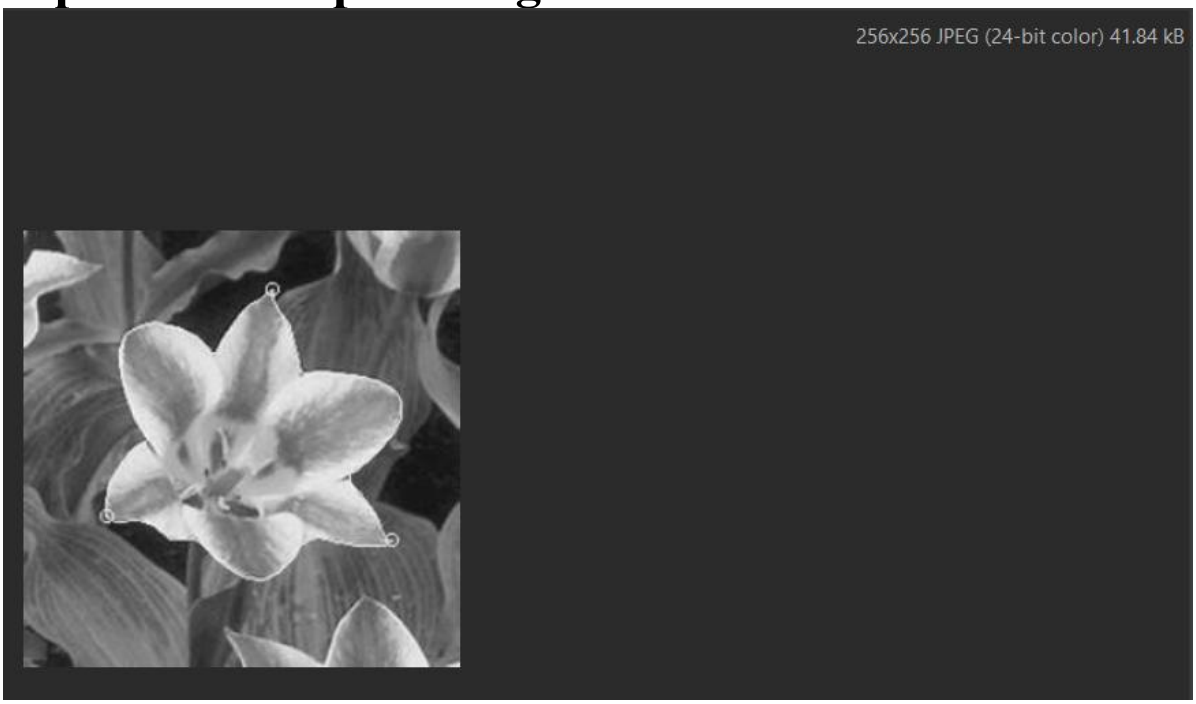

Figure 23 Q1 Part 7 Input Image

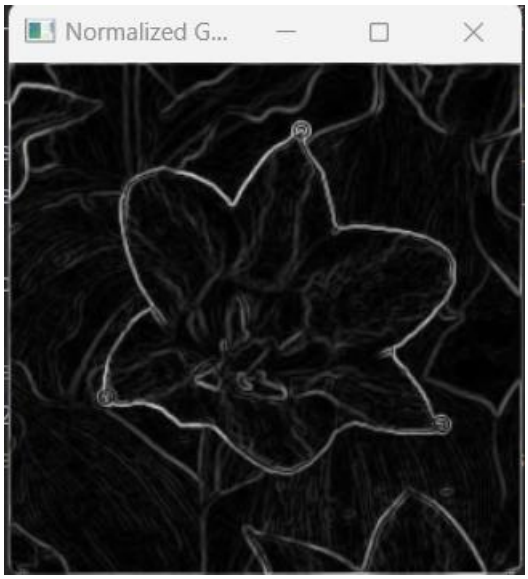

Figure 24 Q1 Part 7 Output Image

Explanation:
In this part, I implemented the sobel operator from scratch. In the first two pictures in the code part above, I just inserted one row and one column of zeros in all directions. In the third picture, I built the sobel operator. And the idea for that is to iterate for each pixels in the 256*256 image,extract the kernel for it, and make a dot product. The last picture, I showed the gradient (magnitude) matrix as an image.

Q2:

Pre-part : Implement The Functions

Code:

```
import cv2
import numpy as np


# ---------------------------------First Part----------------------------------
def insert_zeros(input_image, padded_by):
    image_with_zeros_above = np.vstack([np.zeros((padded_by, input_image.shape[1]), dtype=np.uint8), input_image])

    # Add rows of zeros (according to the filter size) below the last row
    image_with_zeros_below = np.vstack(
        [image_with_zeros_above, np.zeros((padded_by, input_image.shape[1]), dtype=np.uint8)])

    # Add columns of zeros (according to the filter size) to the left of column number 0
    image_with_zeros_left = np.hstack(
        [np.zeros((image_with_zeros_below.shape[0], padded_by), dtype=np.uint8), image_with_zeros_below])

    # Add columns of zeros (according to the filter size) to the right of column number 1
    image_with_zeros_both_sides = np.hstack(
        [image_with_zeros_left, np.zeros((image_with_zeros_left.shape[0], padded_by), dtype=np.uint8)])

    return image_with_zeros_both_sides
```

```
def generate_the_output_image(padded_by, rows, columns, filter, filter_size, image_with_zeros_both_sides, height,
                              width):
    output_image = np.zeros((height, width))
    for row in range(padded_by, rows - padded_by):
        for column in range(padded_by, columns - padded_by):

            kernel_image_for_this_pixel = np.zeros((filter_size, filter_size))
            # Do a filter for the current pixel
            for row_index in range(filter_size):
                current_row = image_with_zeros_both_sides[row - padded_by + row_index][
                              column - padded_by:column + padded_by + 1]
                kernel_image_for_this_pixel[row_index] = current_row

            kernel_image_for_this_pixel = kernel_image_for_this_pixel.flatten()
            # Do a dot product between the current filter for this pixel and the sobel x
            current_pixel = np.dot(kernel_image_for_this_pixel, filter)
            output_image[row - padded_by, column - padded_by] = current_pixel

    return output_image
```

```python
# Define the function
def myImageFilter(input_image, filter):
    filter_size = int(np.sqrt(filter.size))
    padded_by = int((filter_size - 1) / 2)

    image_with_zeros_both_sides = insert_zeros(input_image, padded_by)

    rows = image_with_zeros_both_sides.shape[0]
    columns = image_with_zeros_both_sides.shape[1]

    height = input_image.shape[0]
    width = input_image.shape[1]

    output_image = generate_the_output_image(padded_by, rows, columns, filter, filter_size, image_with_zeros_both_sides,
                                             height, width)
    return output_image
```

Explanation:
In this part, I implemented three functions. The first is to insert row and columns of zeros as previous. The second is to generate the the image by the dot product between image and the kernel. The last one that you requested from me, get the input image and the filter, then pass it to the other function, and retrieve it.

Part 1 : Averaging Kernel (3x3, 5x5)

Code:

```python
def get_average_kernel(filter_size):
    return np.ones(filter_size, dtype=int)
```

---->Kernel With Size = 3x3<----

```python
# Kernel = 3*3
show_averaging_image(9, image_path_House1)
show_averaging_image(9, image_path_House2)
```

---->Kernel With Size = 5*5<----

```python
# Kernel = 5*5
show_averaging_image(25, image_path_House1)
show_averaging_image(25, image_path_House2)
```

Input and Output Images:

Figure 25 Q2 Part1 House1 input Image

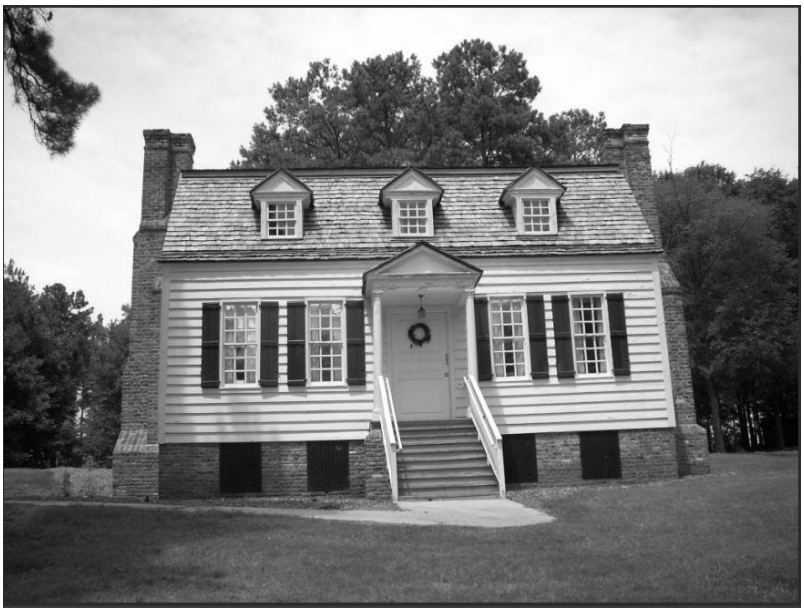

Figure 26 Q2 Part1 House2 input Image

Figure 27 Q2 Part1 House1 Output Image With Kernel = 3x3

Figure 28 Q2 Part1 House2 Output Image With Kernel = 3x3

Figure 29 Q2 Part1 House1 Output Image With Kernel = 5x5

Figure 30 Q2 Part1 House2 Output Image With Kernel = 5x5

Explanation:
In this part, we see briefly that the images have processed with two kernels, 3x3 and 5x5. We can see that the kernel with size 5x5 is more blurring for edges than the size = 3x3. The reason for that, you take more number of pixels to affect this current pixel. if we take the whole size of the image as a kernel for the current pixel, it will be the most blurred case.

Part 2 : Gaussian Kernel (σ = 1,2,3)

Code:

```
# --------------------------------Second Part----------------------------------

def get_gaussian_kernel(sigma):
    gaussian_kernel = cv2.getGaussianKernel(2 * sigma + 1, sigma)
    # Multiply the kernel by its transpose to get a 2D Gaussian filter
    return np.outer(gaussian_kernel, gaussian_kernel.transpose())


def show_gaussian_image(sigma, image_path):
    filter_size = (2 * sigma + 1) ** 2
    filter = get_gaussian_kernel(sigma)
    filter = filter.flatten()
    input_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    output_image = myImageFilter(input_image, filter)

    # The Gaussian Filter is already did the normalization
    output_image = cv2.convertScaleAbs(output_image)
    cv2.imshow("Apply Gaussian Kernel With Size = {}".format(filter_size), output_image)
    cv2.waitKey(0)
```

Explanation:
In this part, I implemented two functions. The first is to get the gaussian kernel depends on the value of the variance. The second is to pass the gaussian kernel and the image to the function ***myFilterImage() done in the prepart section***. After that, I just printed the results.

---->Kernel With σ = 1 , Gaussian Filter = 3x3<----

```
# Show For sigma = 1
show_gaussian_image(1, image_path_House1)
show_gaussian_image(1, image_path_House2)
```

---->Kernel With σ = 2 , Gaussian Filter = 5x5<----

```
#Show For sigma = 2
show_gaussian_image(2, image_path_House1)
show_gaussian_image(2, image_path_House2)
```

---->Kernel With σ = 3 , Gaussian Filter = 7x7<----

```
#Show For sigma = 3
show_gaussian_image(3, image_path_House1)
show_gaussian_image(3, image_path_House2)
```

Input and Output Images:

Figure 31 Q2 Part2 House1 input Image

Figure 32 Q2 Part2 House2 input Image

Figure 33 Q2 Part2 House1 Output Image with σ= 1

Figure 34 Q2 Part2 House2 Output Image with σ= 1

Figure 35 Q2 Part2 House1 Output Image with σ= 2

Figure 36 Q2 Part2 House2 Output Image with σ= 2

Figure 37 Q2 Part2 House1 Output Image with σ= 3

Figure 38 Q2 Part2 House2 Output Image with σ= 3

Explanation:

In this part, I just tested the function done in part 1 for different values of sigma. We notice that when we bigger the size of sigma, the blurring of the image increased,

because the kernel size increased.

Part 3 : Sobel Operator

Code:

```
def show_sobel_image(image_path, sobel_x, sobel_y):
    input_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    Gx = myImageFilter(input_image, sobel_x)
    Gy = myImageFilter(input_image, sobel_y)
    G = (Gx ** 2 + Gy ** 2) ** 0.5
    sobel_image = cv2.normalize(G, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)
    cv2.imshow('Normalized Gradient', cv2.convertScaleAbs(sobel_image))
    cv2.waitKey(0)

#  Test the sobel for the two images
sobel_x_kernel = np.array([-1, -2, -1, 0, 0, 0, 1, 2, 1])
sobel_y_kernel = np.array([-1, 0, 1, -2, 0, 2, -1, 0, 1])
show_sobel_image(image_path_House1, sobel_x_kernel, sobel_y_kernel)
show_sobel_image(image_path_House2, sobel_x_kernel, sobel_y_kernel)
```

Input and Output Images:

Figure 39 Q2 Part3 House1 input Image

Figure 40 Q2 Part3 House2 input Image

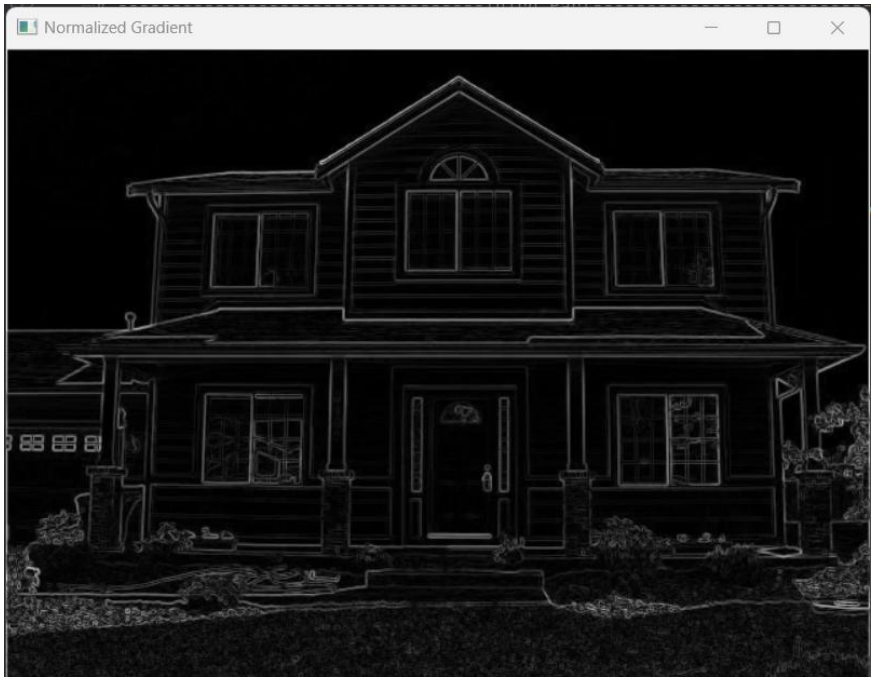

Figure 41 Q2 Part3 House1 Output Image

Figure 42 Q2 Part3 House2 Output Image

Explanation:

In this part, I tested the function myFilterImage() in a sobel operator. We can see that this part looks like the part 7 in Q#1. After applied the sobel operator, we see that just the strong edges were shown and the others have blurred. Sobel is better than prewitt because it has a kind of smoothing the image for two reasons: noise reduction and remain just the strong edges.

Part 4 : Prewitt Operator

Code:

```
# ---------------------------------Forth Part-----------------------------------

def show_prewitt_image(image_path, prewitt_x, prewitt_y):
    input_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    Gx = myImageFilter(input_image, prewitt_x)
    Gy = myImageFilter(input_image, prewitt_y)
    G = (Gx ** 2 + Gy ** 2) ** 0.5
    prewitt_image = cv2.normalize(G, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)
    output_image = cv2.convertScaleAbs(prewitt_image)
    cv2.imshow("Normalized Gradient: ", output_image)
    cv2.waitKey(0)
```

```
#  Test the prewitt for the two images
prewitt_x = np.array([-1, -1, -1, 0, 0, 0, 1, 1, 1])
prewitt_y = np.array([-1, 0, 1, -1, 0, 1, -1, 0, 1])

show_prewitt_image(image_path_House1, prewitt_x, prewitt_y)
show_prewitt_image(image_path_House2, prewitt_x, prewitt_y)
```

Input and Output Images:

Figure 43 Q2 Part4 House1 input Image

Figure 44 Q2 Part4 House2 input Image

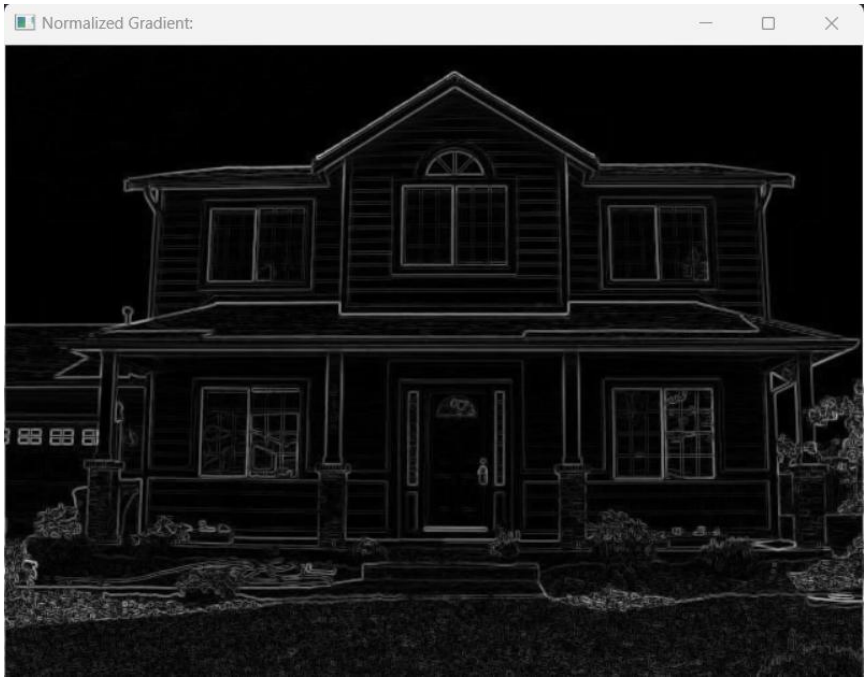

Figure 45 Q2 Part4 House1 Output Image

Figure 46 Q2 Part4 House2 Output Image

Explanation:

In this part, I just tested the function done in part 1 on the prewitt kernels. We can see the output is a little more sharpening than case in the sobel operator, because the prewitt does not make any smoothing for the image, so sobel is better than prewitt.

Q3:

Code:

```python
# For the first image ----> "Noisyimage1.jpg"
# Remove (Not Reduce) the salt and pepper noisy data by the median filter
import cv2

Noisyimage1_path = "Noisyimage1.jpg"
Noisyimage2_path = "Noisyimage2.jpg"


def show_median_image(image_path, filter_size):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image_after_median_filter = cv2.medianBlur(image, filter_size)
    cv2.imshow("Image With Median Filter ----> 5*5 Kernel Size", image_after_median_filter)
    cv2.waitKey(0)

def show_averaging_image(image_path, filter_size):
    input_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image_after_box_filter_with_solt_and_pepper = cv2.boxFilter(input_image, -1, filter_size)
    cv2.imshow("Apply Kernel With Size = {}".format(filter_size), image_after_box_filter_with_solt_and_pepper)
    cv2.waitKey(0)
```

```python
# Apply 5 by 5 averaging filtration
filter_size = (5, 5)
show_averaging_image(Noisyimage1_path, filter_size)
show_averaging_image(Noisyimage2_path, filter_size)

# Apply 5 by 5 median filtration
show_median_image(Noisyimage1_path, filter_size[0])
show_median_image(Noisyimage2_path, filter_size[0])
```

Figure 47 Q3 Code

Input and Output Images:

Figure 48 Q3 Input Image1

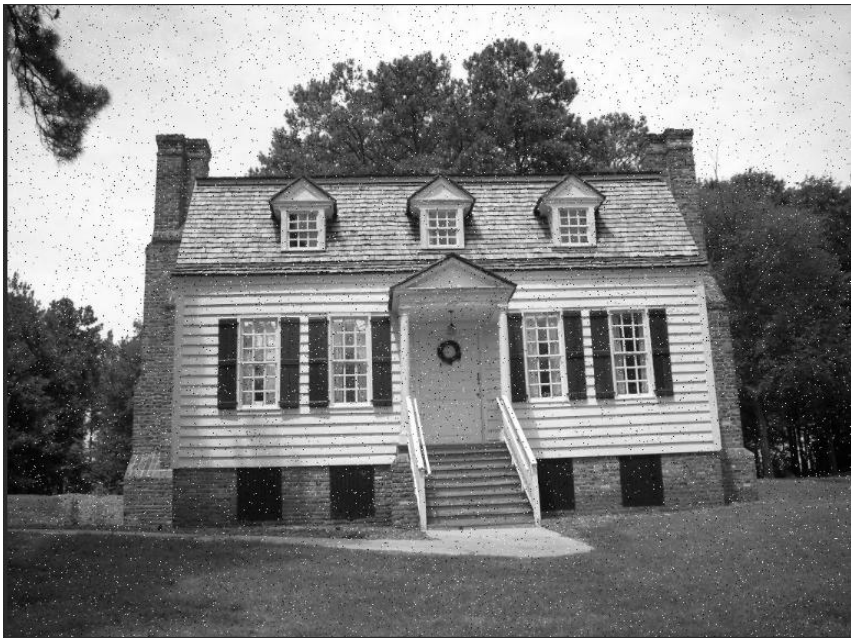

Figure 49 Q3 Input Image2

Figure 50 Q3 Output Image1 With Averaging Filter 5x5

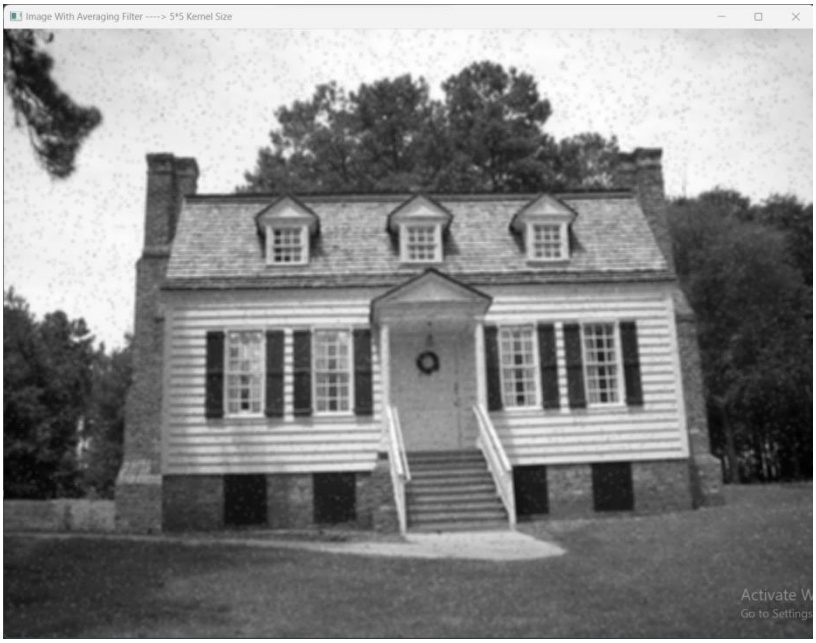

Figure 51 Q3 Output Image2 With Averaging Filter 5x5

Figure 52 Q3 Output Image1 With Median Filter 5x5

Figure 53 Q3 Output Image2 With Median Filter 5x5

Explanation:

In general, the median and mean filters are used for blurring the edges and noise reduction. If you know that the noise in the image is from salt-pepper noise, then you should use median filter, for two reasons: it replace the salt-pepper noise with the median of the kernel after ranking it, so we remove not reduce the noise. The other reason that the median filter, it does not take a long time to find the results, because we do not need any mathematical operation such as dot product between the kernel for a pixel and for the averaging kernel.

Q4:

Part 1 : Stretching The Sobel Magnitude

Code:

```python
# Compute gradient magnitude for attached image "Q4_Image"
import cv2
import numpy as np
import matplotlib.pyplot as plt


# ---------------------------------First Part-------------------------------
def get_sobel_magnitude_image(image_path):
    # Read the image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Apply Sobel filter
    sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

    # Calculate the gradient magnitude
    sobel_magnitude = np.sqrt(sobel_x ** 2 + sobel_y ** 2)
    # Normalize the result to be in the range [0, 255]
    cv2.normalize(sobel_magnitude, sobel_magnitude, 0, 255, cv2.NORM_MINMAX)
    sobel_magnitude_stretched = cv2.convertScaleAbs(sobel_magnitude)
    return sobel_magnitude_stretched
```

```python
def show_sobel_image(image):
    cv2.imshow("Sobel Image:", image)
    cv2.waitKey(0)
```

```python
image_path = "Q_4.jpg"
# Show the stretched magnitude image
sobel_magnitude_stretched = get_sobel_magnitude_image(image_path)
show_sobel_image(sobel_magnitude_stretched)
```

Figure 54 Q4 Part1 Code

Input and Output Images:

Figure 55 Q4 Part1 Input Image

Figure 56 Q4 Part 1 Output Image

Explanation:

I used the ready sobel function to get the gradient of x and y, for both magnitude and phase. In this part, I got the magnitude for the gradient, normalize it to the range [0,255] and stretched it by the function ***convertScaleAbs().*** I used the min-max normalization technique.

Part 2 : Compute the histogram of gradient magnitude

Code:

```
# ---------------------------------Second Part-----------------------------------
def show_mag_histogram(gradient):
    gradient_histogram, bins = np.histogram(gradient.flatten(), bins=256)
    gradient_histogram = gradient_histogram / sum(gradient_histogram)
    # Plot the histogram
    plt.bar(bins[:-1], gradient_histogram, width=1, color='red')
    plt.title('Image Magnitude Histogram')
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.show()

# Show the histogram of the magnitude
show_mag_histogram(sobel_magnitude_stretched)
```

Figure 57 Q4 Part 2 Code

Image Magnitude Histogram:

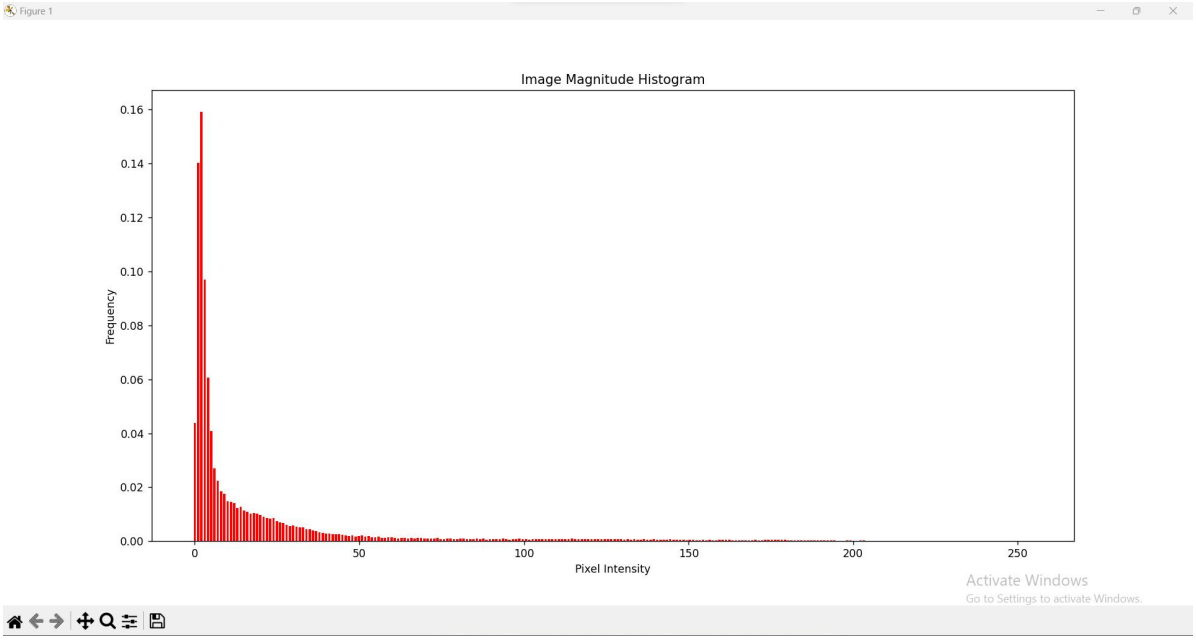

Figure 58 Q4 Part2 Image Magnitude Histogram

Explanation:

In this part I calculated the histogram for the image output from the previous part, normalize it by divided by the summation of the whole image and show the results by matplotlib library. We can notice from the histogram that the image tends to be more dark than white, and this is sensible, because the gradient magnitude of it is darker.

Part 3 : Compute the gradient orientation

Code:

```python
# ---------------------------------Third Part-----------------------------------
def get_sobel_orientation_image(image_path):
    # Read the image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Apply Sobel filter
    sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

    # Calculate the gradient Orientation
    sobel_orientation = np.arctan2(sobel_y, sobel_x)
    cv2.normalize(sobel_orientation, sobel_orientation, 0, 180, cv2.NORM_MINMAX)
    sobel_orientation_stretched = cv2.convertScaleAbs(sobel_orientation)
    return sobel_orientation_stretched

# Show the stretched orientation image
sobel_orientation_stretched = get_sobel_orientation_image(image_path)
show_sobel_image(sobel_orientation_stretched)
```

Figure 59 Q4 Part 3 Code

Input and Output Images:

Figure 60 Q4 Part 3 Input Image

Figure 61 Q4 Part 3 Output Image

Explanation:

In this part, I did the same as part 2 above. I calculated the gradient x and y of the image, then take arctan(y/x) for the gradient to find the orientation gradient. Then I stretched it by the same function above and finally showed it. We can notice that the maximum value of the pixel can get is pi/2, because arctan(inf) = pi/2, for this reason the image color as above.

Part 4 : Compute the Histogram orientation

Code:

```
# ---------------------------------Forth Part-----------------------------------
def show_ori_histogram(gradient):
    gradient_histogram, bins = np.histogram(gradient.flatten(), bins=180)
    # Plot the histogram
    plt.bar(bins[:-1], gradient_histogram, width=0.7, color='blue')
    plt.title('Image Orientation Histogram')
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.show()


# Show the histogram of the orientation
show_ori_histogram(sobel_orientation_stretched)
```

```
# Show the histogram of the orientation
show_ori_histogram(sobel_orientation_stretched)
```

Figure 62 Q4 Part 4 Code

Image Orientation Histogram:

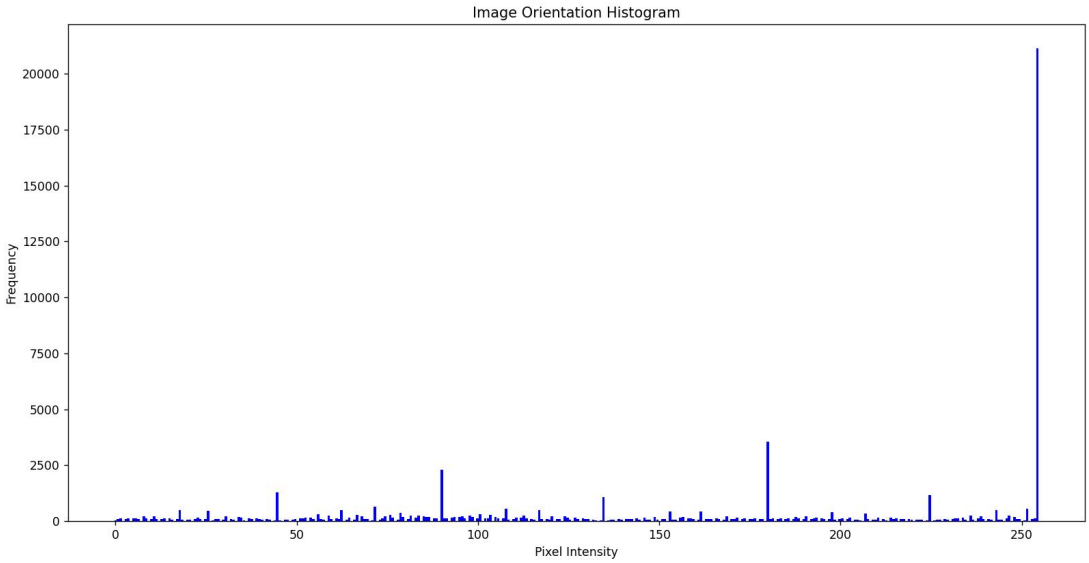

Figure 63 Q4 Part 4 Gradient Orientation Histogram

Explanation:

In this part, I plotted the histogram of the gradient orientation. The procedure of it is as the previous one.

Q5:

Code:

```
# Loading walk_1.jpg and walk_2.jpg images
import cv2


def show_the_difference_between_two_images(image_1_path, image_2_path):
    image_1 = cv2.imread(image_1_path, cv2.IMREAD_GRAYSCALE)
    image_2 = cv2.imread(image_2_path, cv2.IMREAD_GRAYSCALE)
    resulting_image_after_subtraction = image_1 - image_2
    cv2.imshow("Image 1 - Image 2 =", resulting_image_after_subtraction)
    cv2.waitKey(0)
```

```
walk_1_path = "walk_1.jpg"
walk_2_path = "walk_2.jpg"

show_the_difference_between_two_images(walk_1_path, walk_2_path)
```

Figure 64 Q5 Code

Input and Output Images:

Figure 65 Q5 Input Image 1

Figure 66 Q5 Input Image 2

Figure 67 Q5 Output Image

Explanation:

First of all, I converted the two images into greyscale, subtracted image 2 from image 1. We can see the output image is the differences between the first image , the second image and dissolved for intersection regions. And the explanation as this: we have three cases here,

If image 1 [current pixel] > image 2 [current pixel], but not small difference, the output pixel will represents image 1 (detail in image 1)

If image 1 [current pixel] < image 2 [current pixel], but not small difference, the output pixel will represents image 2 pixel (detail in image 2)

If image 1 [current pixel] = image 2 [current pixel], with small difference, the output will dissolves (detail will dissolves).

Thus, the output image will be details in image 1 + details in image 2 + dissolving for intersection details.

We can notice that there is nor difference between the difference between the pixels (0,3), (0,252) and so on. Because we take the 2's complement for the result if it is negative. So in the two cases (0,3), (0,252), the output will be blurred (dissolved), but with different colors.

Q6:

Code:

```python
import cv2

image_path = "Q_4.jpg"
def apply_canny_detector(image_path, low_threshold, high_threshold):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image_after_canny_detector = cv2.Canny(image, low_threshold, high_threshold)
    cv2.imshow("Image With Canny Detector:", image_after_canny_detector)
    cv2.waitKey(0)
```

```python
# Test the Canny edge detector for three values of [Low threshold, High threshold]
apply_canny_detector(image_path, 5, 20)  # Low threshold
apply_canny_detector(image_path, 50, 90)  # Medium threshold
apply_canny_detector(image_path, 120, 220)  # High Threshold
```

Input and Output Images:

Figure 68 Q6 Input Image

Figure 69 Q6 Canny Detector With TL = 5 and TH = 20

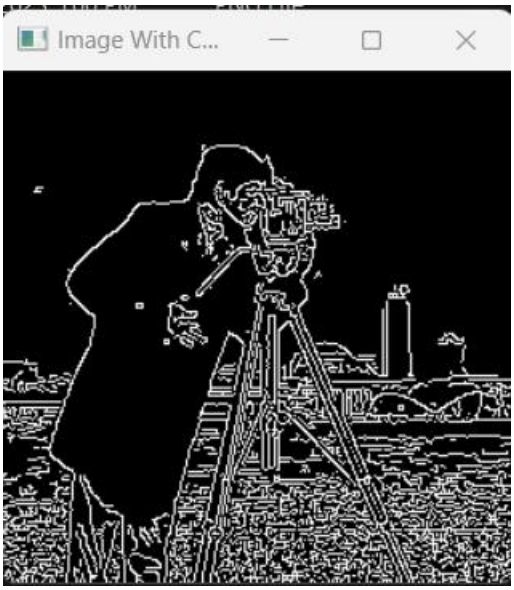

Figure 70 Q6 Canny Detector With TL = 50 and TH = 90

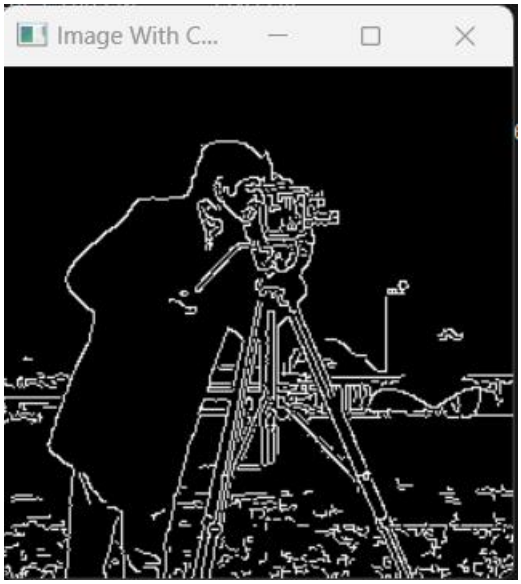

Figure 71 Q6 Canny Detector With TL = 120 and TH = 220

Explanation:

In this part, I used the ready API for the canny detector with different low threshold and high threshold. We notice that when we increase TL and TH only the strong edges are remain. The main factor for edges detection is TL, TH is used for example to connect the broken pixels done by TL, with same direction.