Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS5343 Computer Vision

Assignment # 1

Student name: Rama Abdlrahman

Student ID: 1191344

Notes:

1- Use this page as a cover for your home work.

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.

Contents

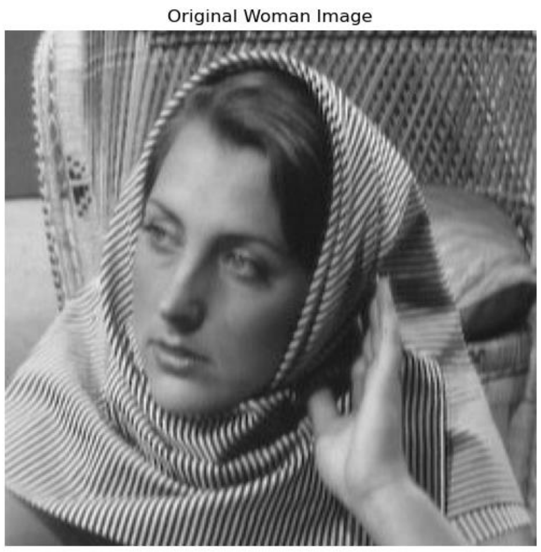
Question 1

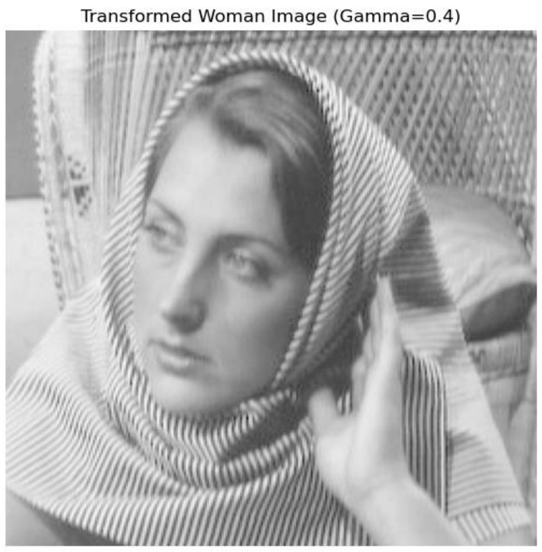
(This question was tested on two images)

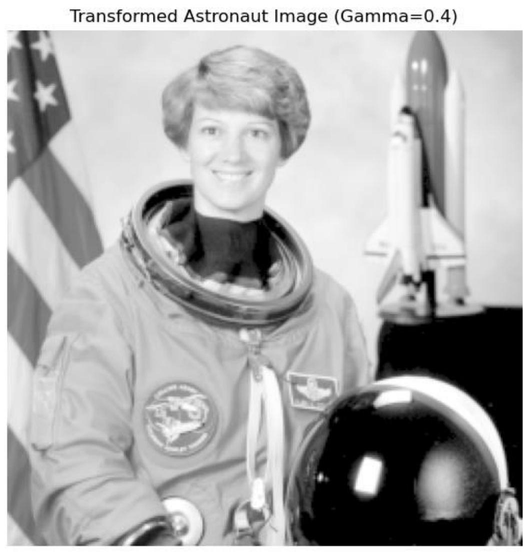
Part 1:

In this part, the two images that were chosen are shown (8-bit gray-level, 256x256 pixels in size).

Part 2:



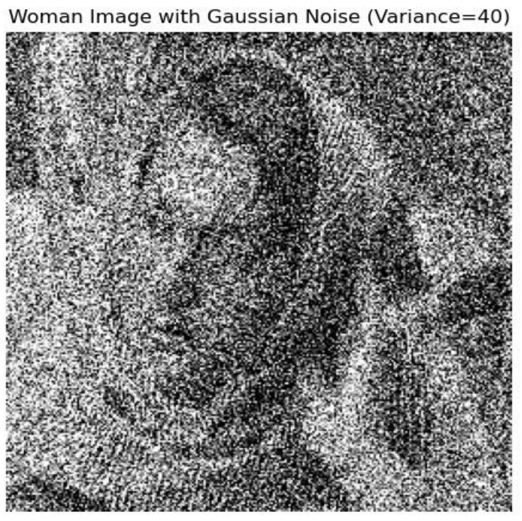


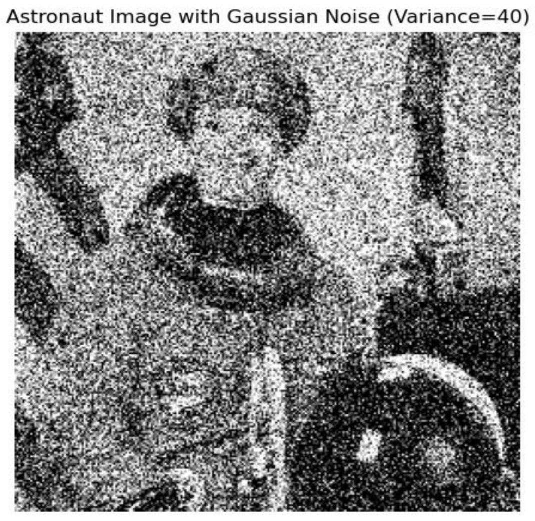


In this part, images underwent a power law transformation with a gamma value of 0.4. This transformation is instrumental in rectifying the contrast of the image. The transformation is expressed as s = c * r^γ, where 's' and 'r' represent the pixel values of the output and input images, respectively. The constant 'c' is set to 1 for simplicity, and γ is the gamma value, which is 0.4.

The output image of this step shows that this transformation tends to enhance the dark regions of the image more than the bright regions, which can be observed in the enhanced contrast and brightness of the darker areas.

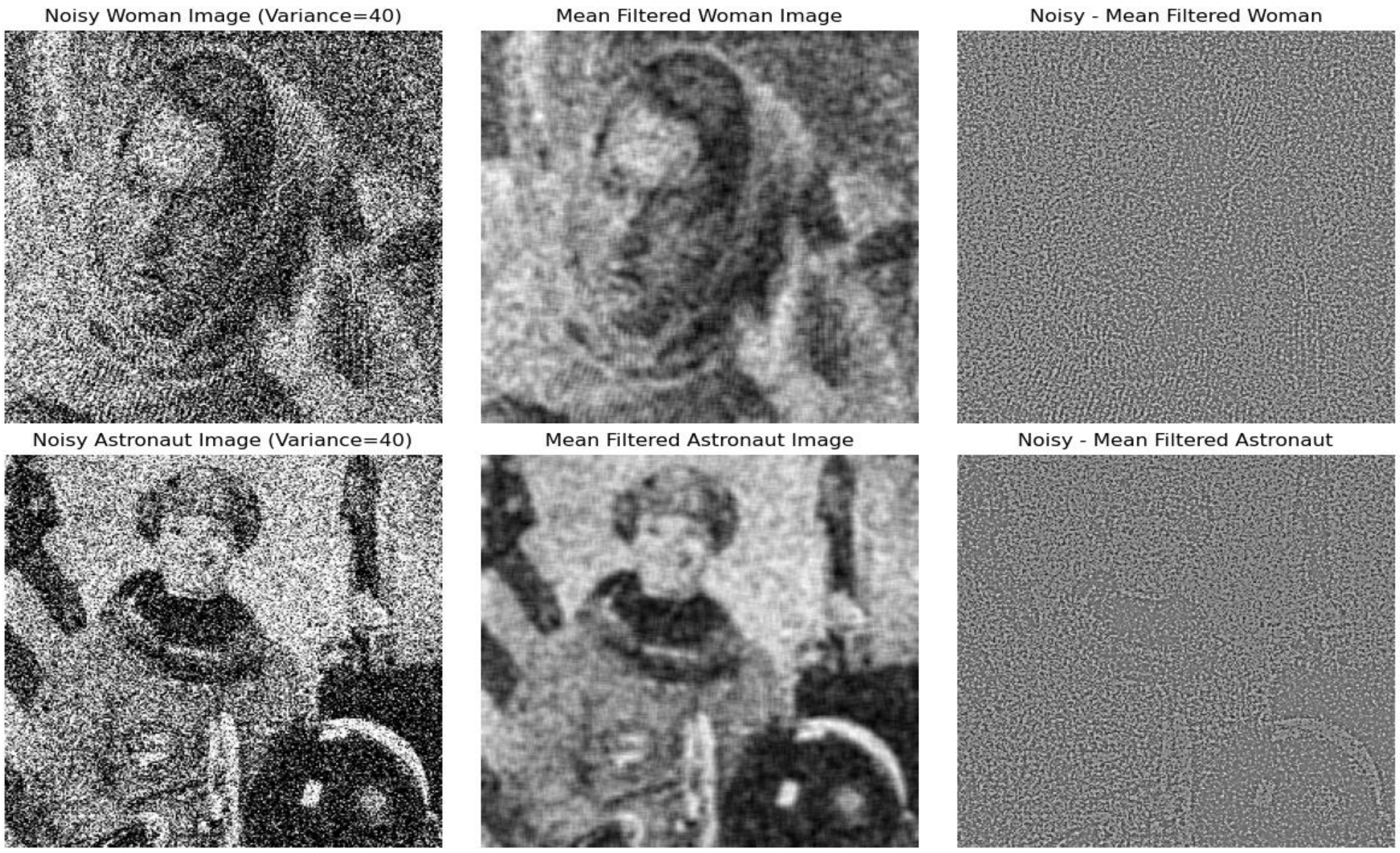
Part 3:





In this section, the images with added Gaussian noise are depicted by the term "noisy images." The presence of this type of noise is characterized by random variations of brightness in the image, representing a common issue encountered in image processing.

Part 4:

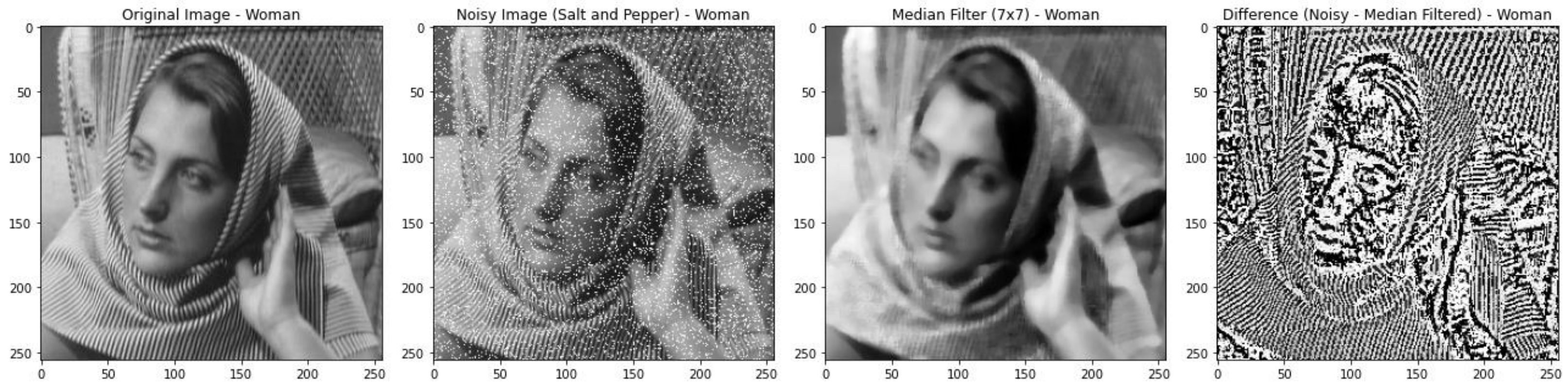

Noisy Woman Image (Variance=40)
Mean Filtered Woman Image
Noisy - Mean Filtered Woman
Noisy Astronaut Image (Variance=40)
Mean Filtered Astronaut Image
Noisy - Mean Filtered Astronaut

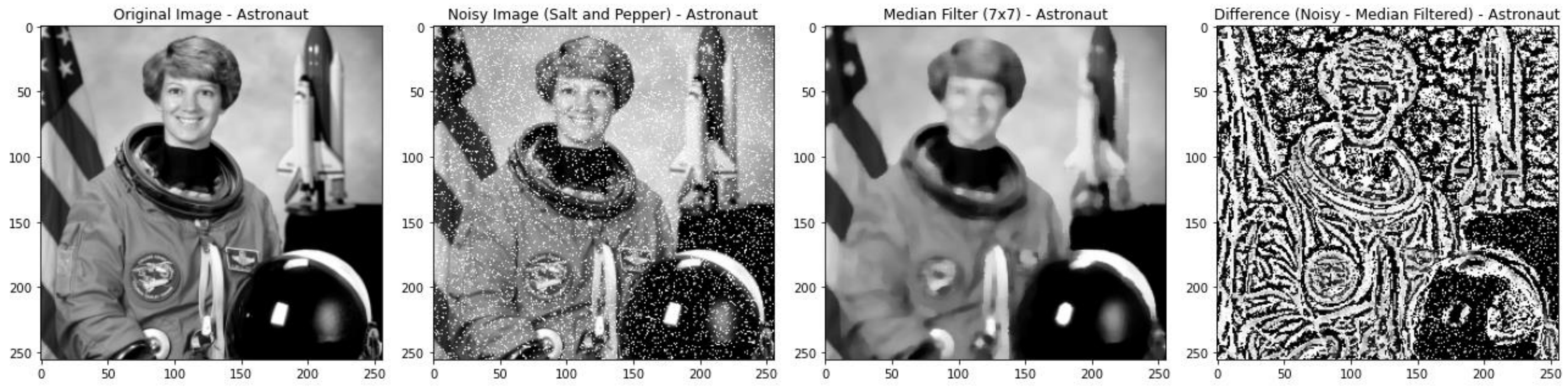
The primary function of the mean filter is to reduce the variance within a local neighborhood of pixels. By replacing each pixel value with the average of its neighbors, the filter effectively diminishes the contribution of outlier values, which are often the result of noise.

In the context of my two images, the mean filter has been applied to a noisy woman image and an astronaut image. The noise, characterized by a Gaussian distribution with a variance of 40, represents a significant degradation of the image quality, obscuring the underlying details. After the application of a 5x5 mean filter, you can observe a discernible improvement in visual clarity. The filter has substantially mitigated the high-frequency noise, which is most readily apparent in the uniform regions of the image. However, the noise is not completely eliminated, as evidenced by the residual variance in the "Noisy - Mean Filtered" images.

One of the primary drawbacks of the mean filter is its indiscriminate averaging effect, which does not differentiate between noise and edges. As a result, while the noise is reduced, there is also a concomitant blurring of edges and details. This effect is noticeable in areas where there should be sharp transitions. The "Noisy - Mean Filtered" images starkly illustrate the trade-off between noise reduction and detail preservation. The gray-scale variations in these difference images highlight the areas where the mean filter has altered the pixel values, with brighter areas indicating greater change.

Part 5:


Original Image - Woman
Noisy Image (Salt and Pepper) - Woman
Median Filter (7x7) - Woman
Difference (Noisy - Median Filtered) - Woman
0
50
100
150
200
250


Original Image - Astronaut
Noisy Image (Salt and Pepper) - Astronaut
Median Filter (7x7) - Astronaut
Difference (Noisy - Median Filtered) - Astronaut
0
50
100
150
200
250

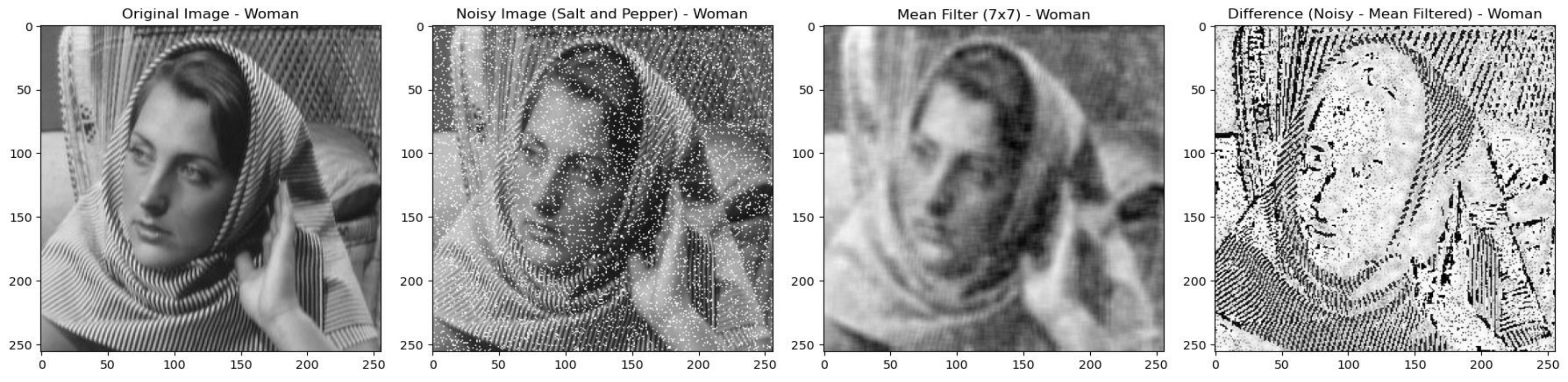
For the woman images, the first one depicts the original image, while the second one showcases the same image with added salt and pepper noise. This particular type of noise is characterized by the random scattering of white and black pixels throughout the image.

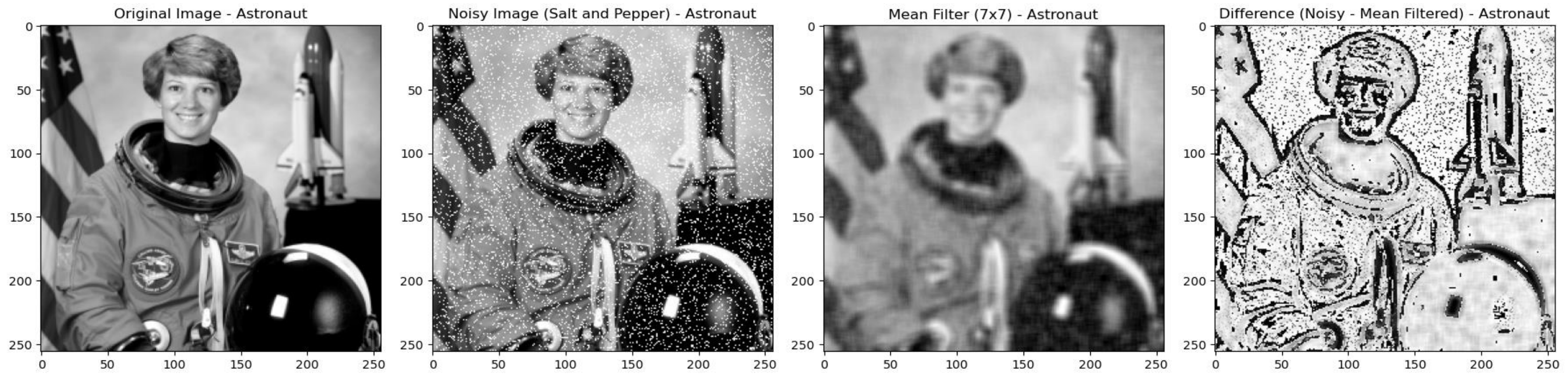
To address this specific noise, median filtering is employed a non-linear digital filtering technique designed for noise removal. The third image reveals the outcome after applying a 7x7 median filter to the noisy image. The median filter proves highly effective in eliminating salt and pepper noise while preserving the image's edges; in fact, this type of filter excels at salt and pepper noise removal.

The filter achieves its efficacy by replacing each pixel's value with the median intensity of the neighboring pixels. The filter traverses the image pixel by pixel, substituting each value with the median intensity derived from the neighboring pixels. The size of the neighborhood is adjustable, as denoted by the '7x7' label, indicating the dimensions of the pixel neighborhood considered by the filter when calculating the median.

The Difference (Noisy - Filtered) image illustrates the pixel-wise contrast between the noisy and filtered images, effectively highlighting the successfully removed noise.

Part 6:


Original Image - Woman
Noisy Image (Salt and Pepper) - Woman
Mean Filter (7x7) - Woman
Difference (Noisy - Mean Filtered) - Woman
0
50
100
150
200
250

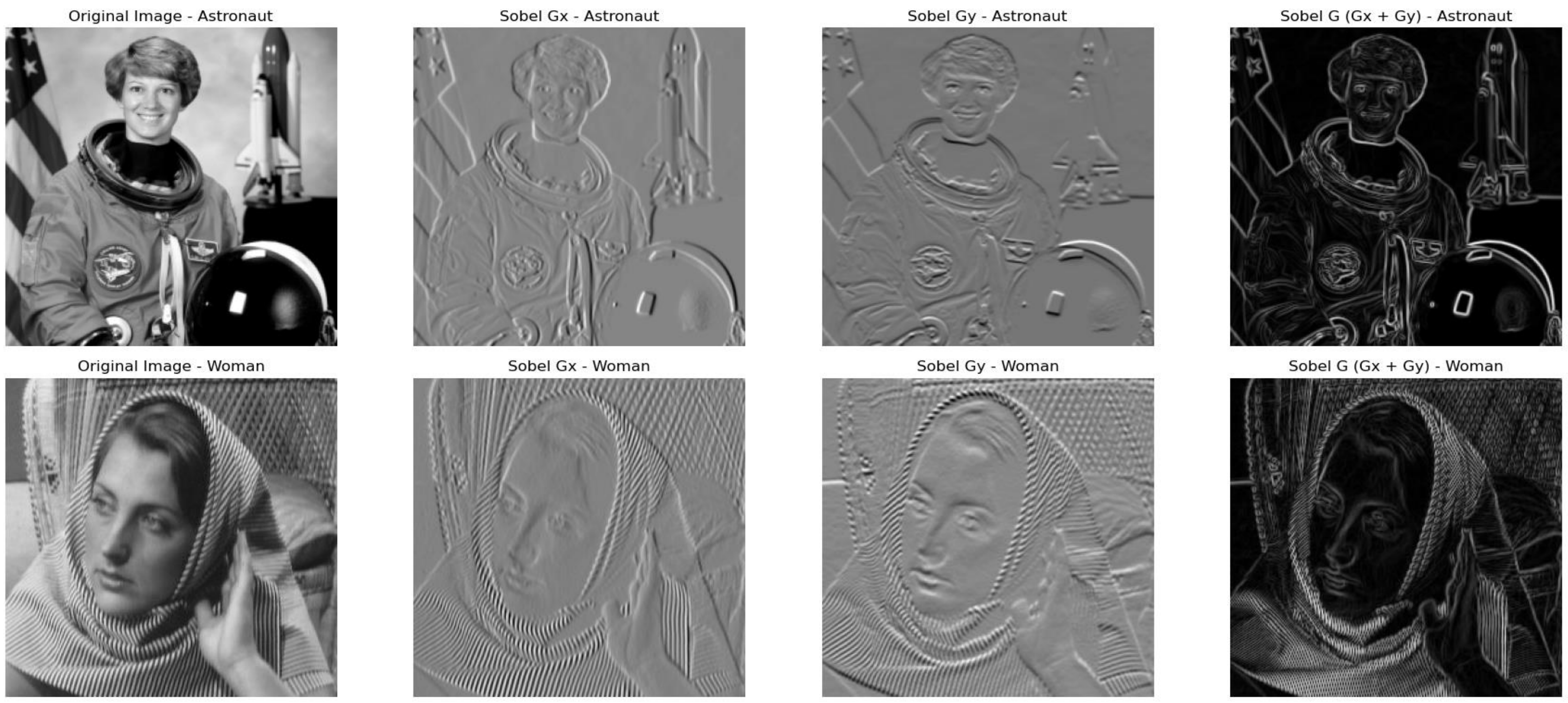

Original Image - Astronaut
Noisy Image (Salt and Pepper) - Astronaut
Mean Filter (7x7) - Astronaut
Difference (Noisy - Mean Filtered) - Astronaut
0
50
100
150
200
250

The images above illustrate the result of applying a 7x7 mean filter to the image with salt and pepper noise. The "Difference (Noisy - Mean Filtered) shows the pixel-wise difference between the noisy and mean-filtered images. It is observed that the mean filter blurs the image, and some noise may still be present.

The mean filter has smoothed the image, reducing the salt and pepper noise. However, this type of filter tends to blur the image more compared to a median filter, which might be noticeable in the loss of some detail and sharpness (edges).

In the presence of salt and pepper noise, median filters are often preferred over mean filters because median filters are better at preserving edges and details while effectively removing that type of noise.

Part 7:



The images provided illustrate the application of the Sobel edge detection filter on two different grayscale photographs one of an astronaut and the other of a woman. The Sobel filter is utilized in image processing to compute the gradient magnitude of the image intensity at each pixel, a method that accentuates areas of high spatial frequency, often associated with the edges of objects within the image.

For each original image, two distinct Sobel kernels are applied: sobel_kernel_x and sobel_kernel_y. The former is designed to detect edges that are vertical by identifying horizontal gradients, and the latter detects edges that are horizontal by identifying vertical gradients. The results of these operations produce two intermediary images that illustrate the rate of change in image intensity in both the x (horizontal) and y (vertical) directions.

In the provided results, the Sobel filter successfully delineates the contours and details of the subjects. This is evidenced by the highlighted features of the astronaut's suit and the facial features of both individuals, which are rendered with sharp transitions to emphasize the edges.

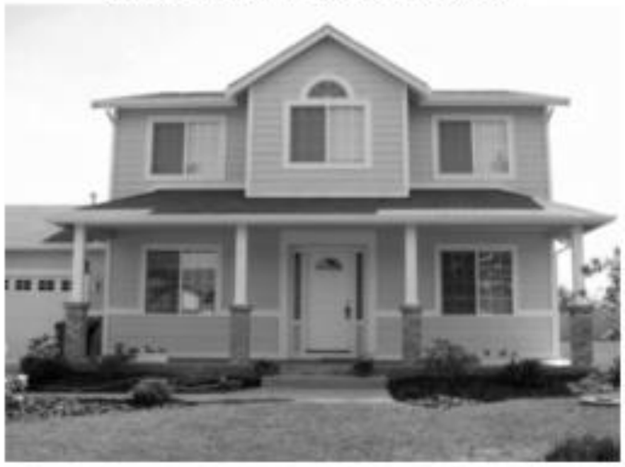
The final step in the Sobel edge detection process involves combining these horizontal and vertical gradients to obtain a comprehensive edge map of the original image. This is achieved using the Euclidean norm to calculate the magnitude of the gradient at each pixel, effectively integrating both directional responses. The images on the far right for both the astronaut and the woman showcase this combined response, known as the sobel_response. Here, the intensity of the edges is proportional to the gradient magnitude, with brighter regions signifying stronger edges. The sobel_response images exhibit a pronounced outline of the subjects, effectively capturing the essence of the original photograph through the stark contrast of its edges.

Question 2

Part 1:

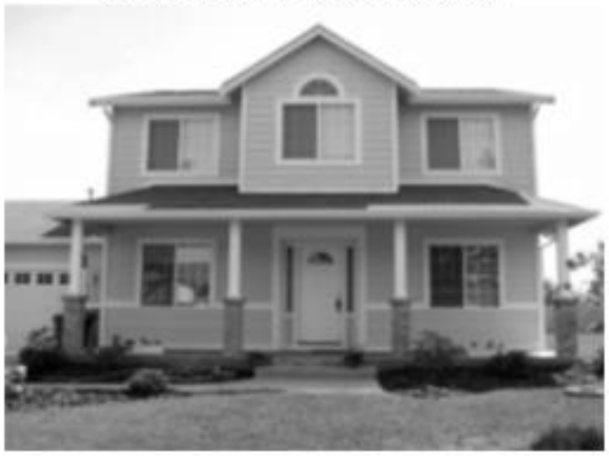
House1 Original

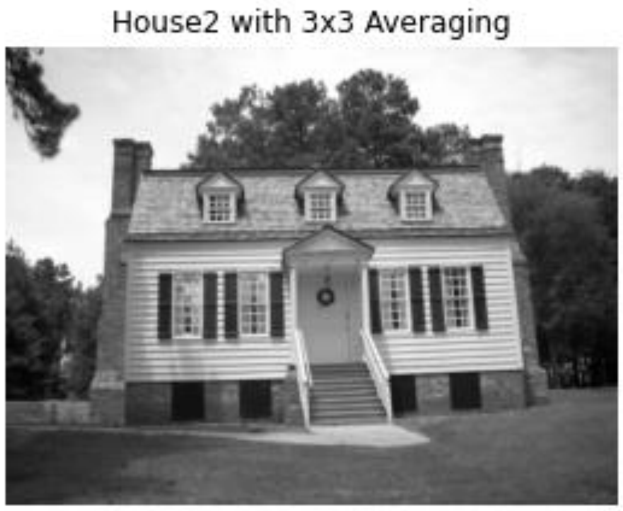
House1 with 3x3 Averaging

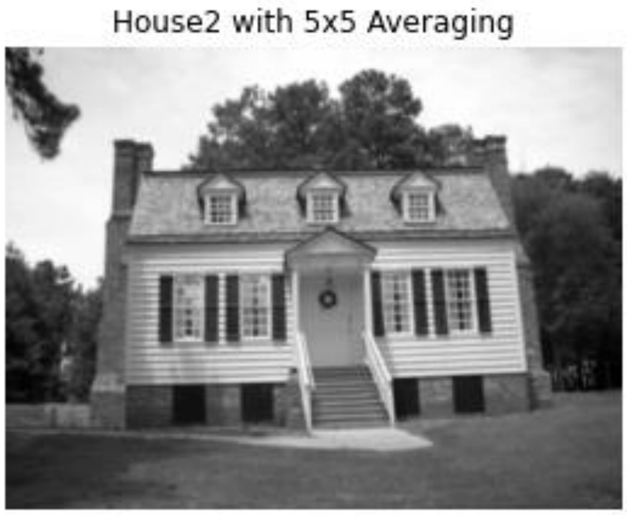


House1 Original

House1 with 5x5 Averaging





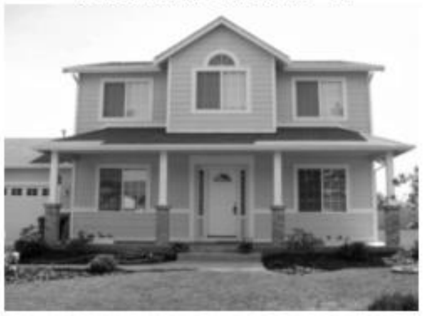


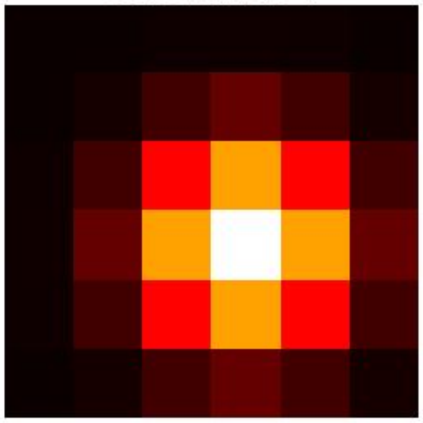
In this part, the image has been processed with a 3x3 averaging filter. As you can see in the filtered image, the details are slightly blurred compared to the original image on the left, due to the averaging effect of the kernel. This kernel effectively reduces noise and smoothens the image by averaging the pixel values within each 3x3 area.

The 5x5 averaging filter has been also applied to the image, and the result is displayed on the right side. This filter has a larger area than the 3x3, which means it averages more pixels together. As a result, the image becomes even more blurred (stronger blur effect), with finer details becoming less distinct compared to the original image on the left. This effect is useful for removing more noise from the image but at the cost of losing detail.

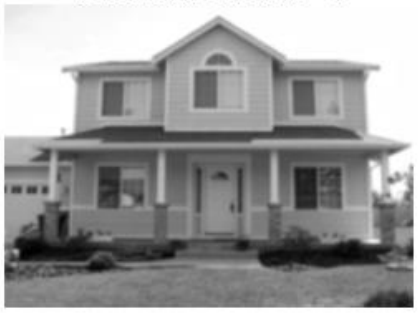
Part 2:

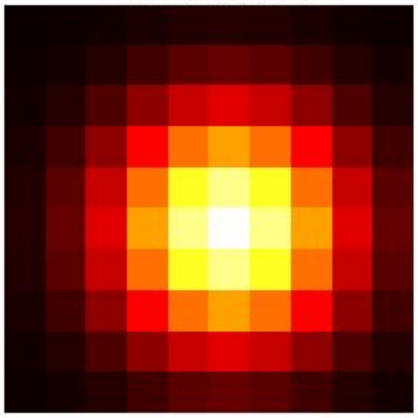
House1 Original


House1 with Gaussian σ=1

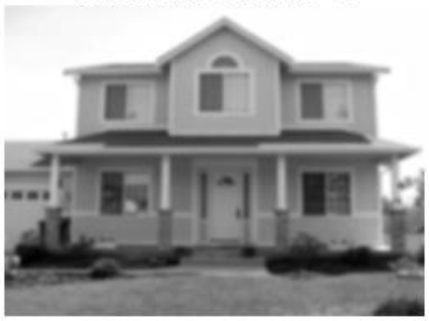

Gaussian Kernel σ=1

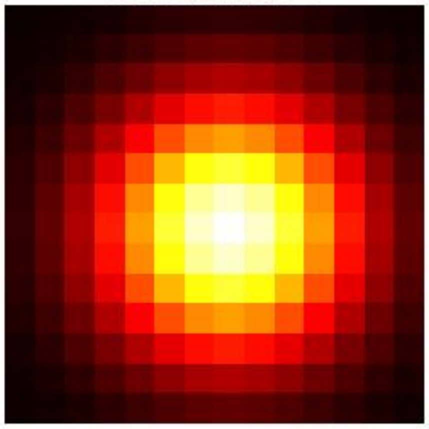
House1 Original

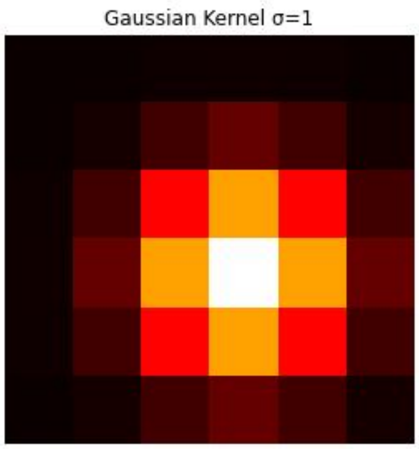

House1 with Gaussian σ=2

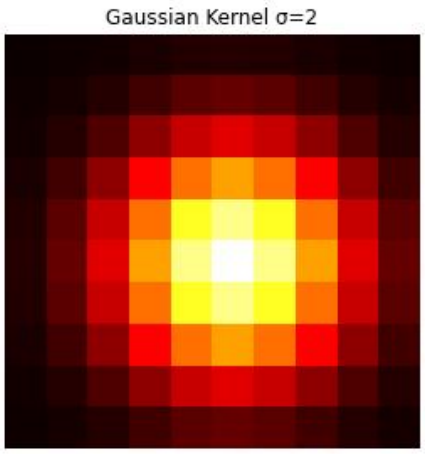

Gaussian Kernel σ=2

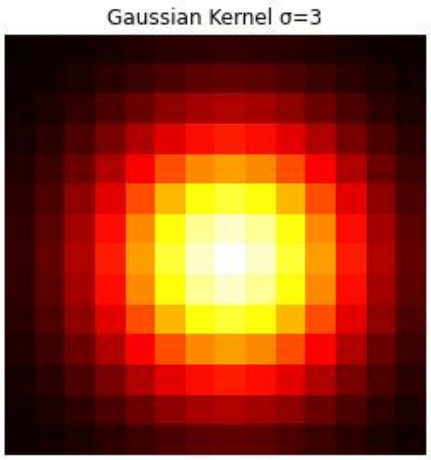
House1 Original


House1 with Gaussian σ=3


Gaussian Kernel σ=3





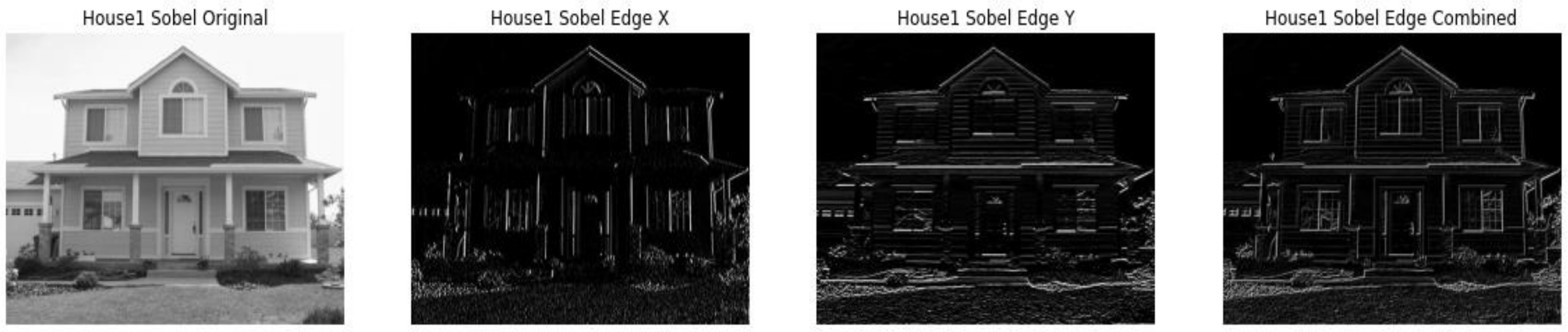


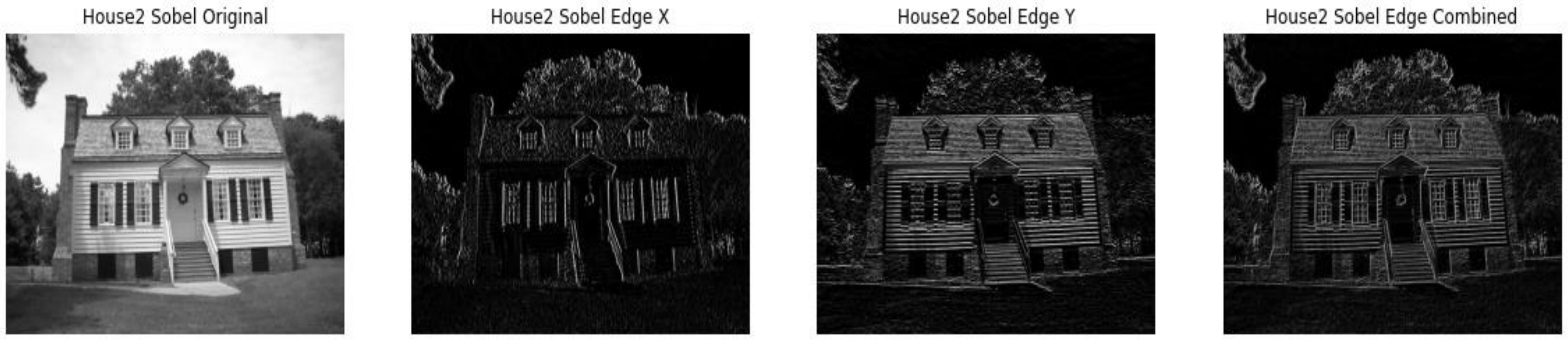
Above are the images processed with Gaussian filters for different values of σ. On the most right, you can see the Gaussian kernels for σ=1, σ=2, and σ=3. These kernels are visualized using a heat map where brighter colors represent higher weights. On the center, the corresponding filtered images show the smoothing effect of the Gaussian filter, with the effect becoming more pronounced as σ increases. The higher the value of

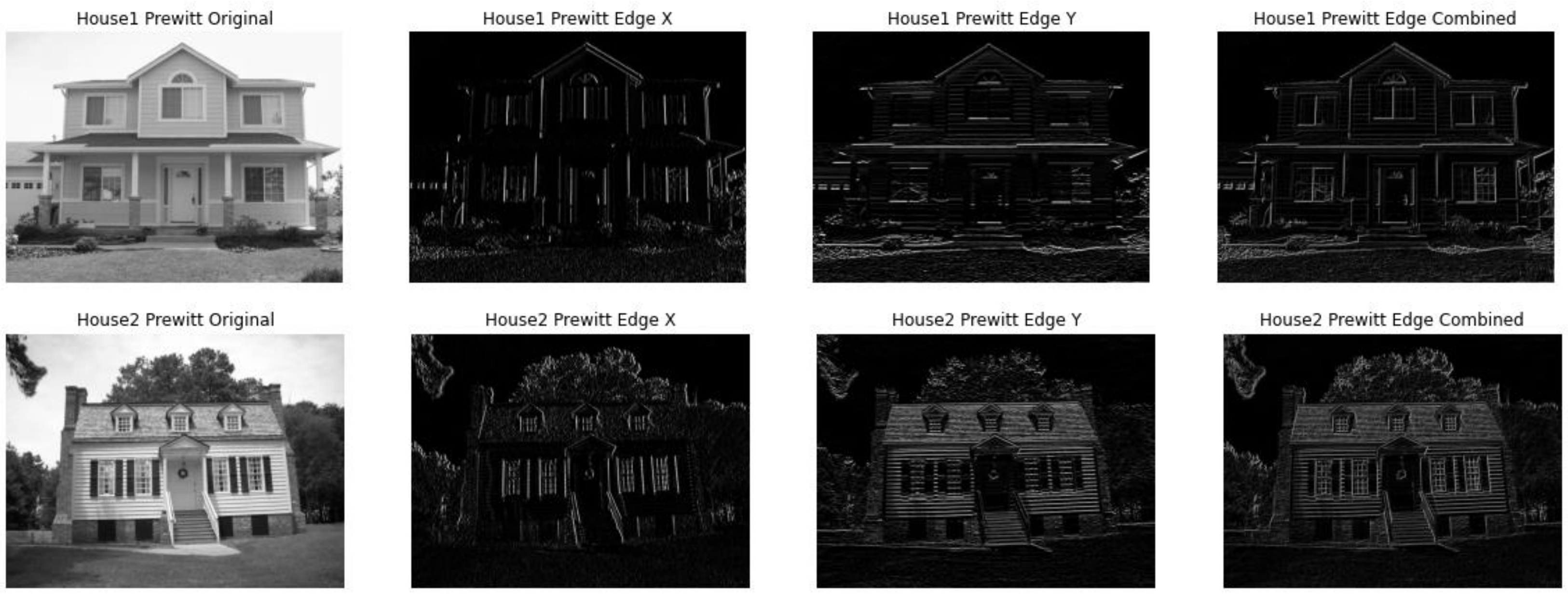
σ, the greater the smoothing, as the filter incorporates a larger neighborhood of pixels in the averaging process.

It is observed from the results that applying Gaussian Kernel with $\sigma=1$ will produce a mild blur, smoothing out the smallest details and noise. The size of this kernel is $(2*1+1)\times(2*1+1)(2*1+1)\times(2*1+1)$, which is 3x3. Gaussian Kernel with $\sigma=2$ is larger and will produce a more pronounced blur than the $\sigma=1$ kernel. It will smooth out medium-sized details and noise. The size of this kernel is $(2*2+1)\times(2*2+1)(2*2+1)\times(2*2+1)$, which is 5x5. Gaussian Kernel with $\sigma=3$: is even larger and will result in a significant blur, affecting large areas of the image. It's suitable for smoothing over larger details and reducing more noise. The size of this kernel is $(2*3+1)\times(2*3+1)(2*3+1)\times(2*3+1)$, which is 7x7.

Part 3 & Part 4 :







The first image in each set is the original grayscale picture of a house. Sobel Edge X: image represents the edges detected in the horizontal direction. The Sobel operator works by convolving a kernel with the original image to calculate the gradient of the image intensity at each pixel. This operation highlights regions of the image where the intensity changes most rapidly in the horizontal direction, which typically correspond to vertical edges.

Sobel Edge Y image shows the edges detected in the vertical direction. It uses a kernel that is rotated 90 degrees from the one used for detecting horizontal edges. This image emphasizes changes in intensity in the vertical direction, highlighting horizontal edges.

Sobel Edge Combined image combines the results of the horizontal and vertical edge detection. This is typically done by computing the magnitude of the gradient vector at each pixel, which combines both the X and Y gradients. The result is a more complete edge map that shows all the prominent edges in the image.

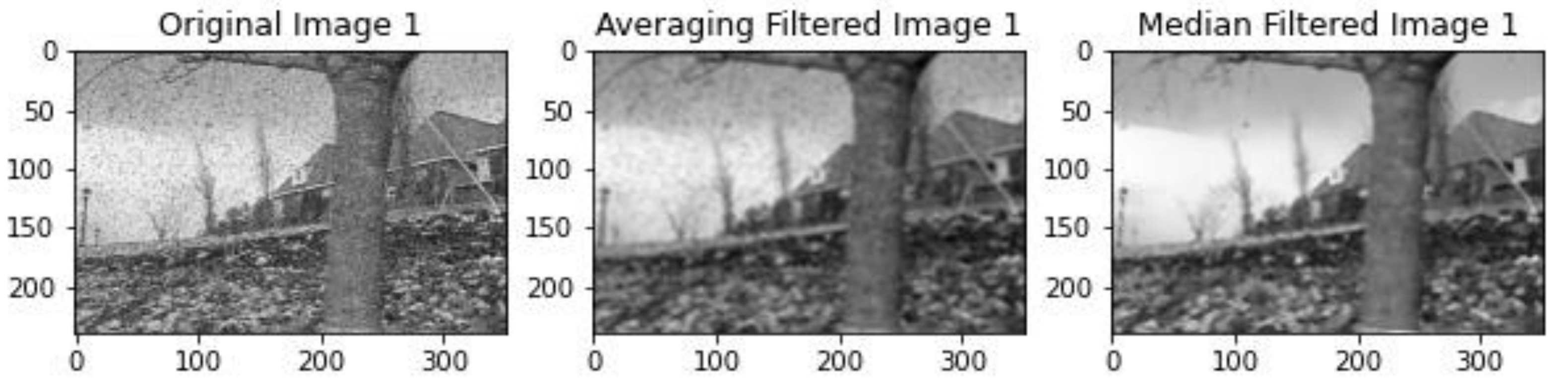
Prewitt Edge X image depicts the outcome of applying the Prewitt operator to detect horizontal changes in intensity. The Prewitt operator uses a 3x3 kernel that emphasizes horizontal transitions in brightness. This results in the detection of vertical edges within the image, as areas of high horizontal gradient are highlighted.

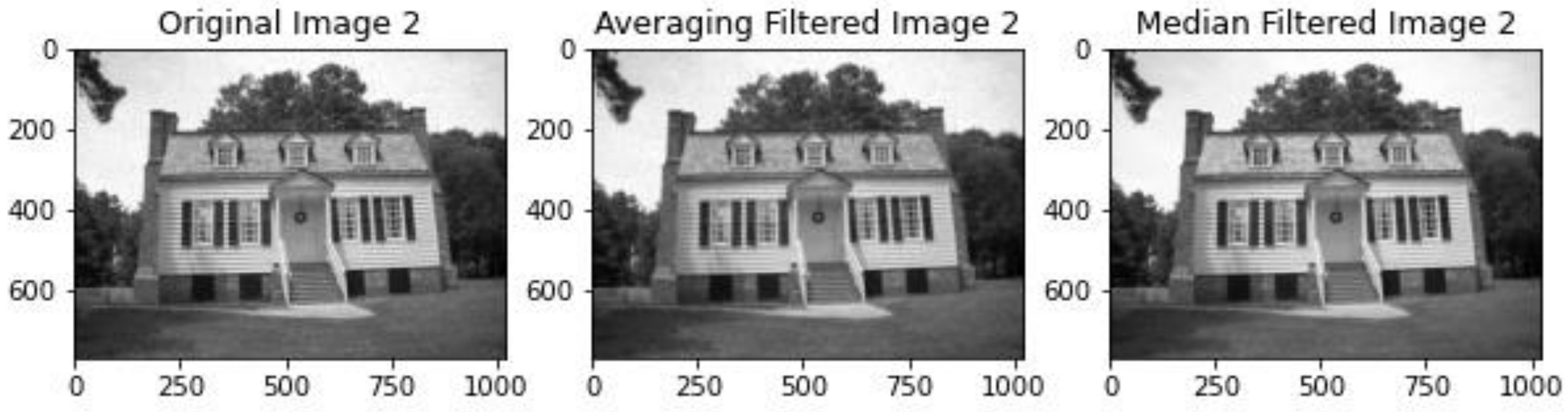
Prewitt Edge Y image illustrates the edges detected in the vertical direction using the Prewitt operator. The kernel used for this direction is designed to emphasize vertical changes in brightness, and thus, it detects horizontal edges.

Prewitt Edge Combined image combines the horizontal and vertical edge detections. This is done by calculating the vector magnitude of the gradients at each pixel, using both the X and Y gradient images. This process enhances the visibility of all the significant edges within the image, regardless of their orientation.

The Prewitt operator bears resemblance to the Sobel operator in that both are edge detection tools using convolution kernels; however, the Prewitt operator employs a kernel that assigns equal weight to all contributing pixels, resulting in less emphasis on those pixels that are in close proximity to the target pixel. This characteristic is evident in the edge magnitude images produced by both operators, where the intensity of the detected edges is represented by brighter regions indicating stronger edge responses. While the Prewitt operator tends to place less focus on minute edge details in comparison with the Sobel operator, it benefits from a reduced sensitivity to image noise, making it more robust in noisy environments. Nonetheless, this robustness may come at the cost of diminished precision in the identification of subtle edge variations.

Question 3





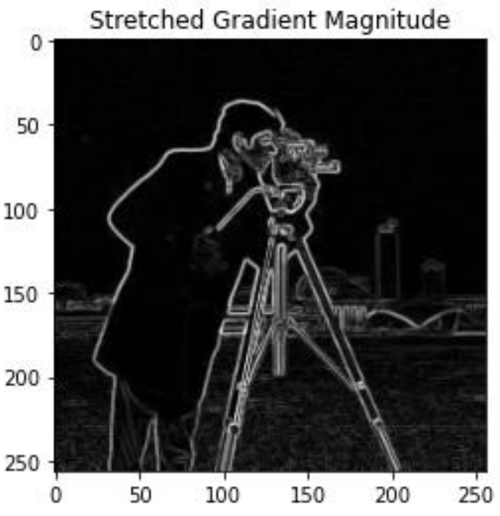
From the median filter outputs, it's evident that there's a significant reduction in the salt-and-pepper noise. The median filter works better than the averaging filter for this type of noise because it is a non-linear process. Unlike the averaging filter which simply computes the mean of the pixel values within the filter window (including the noise pixels, which can be extreme values), the median filter sorts the values and selects the median value. This effectively ignores the extreme values caused by the noise.

In the case of salt-and-pepper noise, which consists of random occurrences of black and white pixels, the averaging filter will include these extreme values in its computation, often resulting in a less effective noise reduction. In contrast, the median filter is more robust as it selects the median value, which is less likely to be an extreme value. Consequently, the median filter can maintain the edge sharpness while removing noise, which is particularly beneficial for images with this type of corruption.

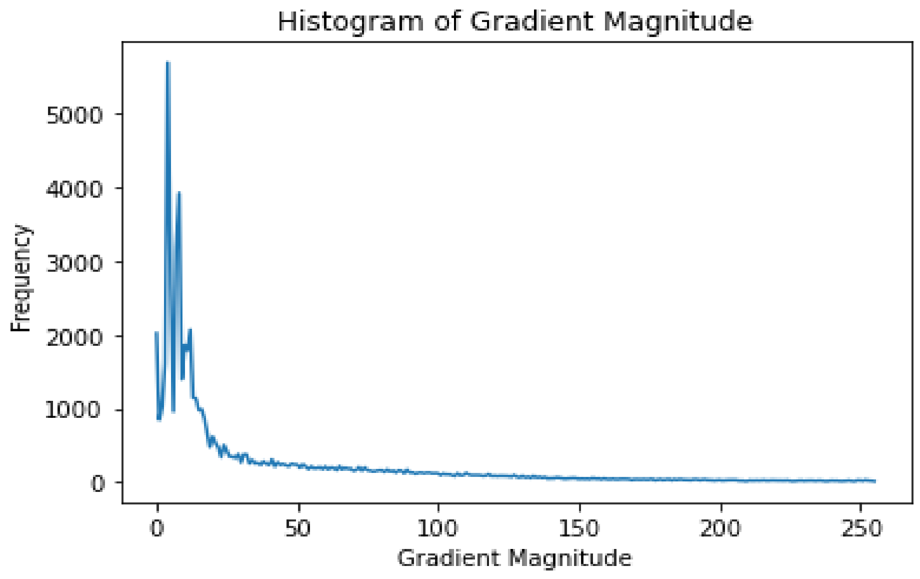
The averaging filter image shows a reduction in noise, but it also has a blurring effect on the details of the image. This is because the averaging filter smooths out the pixel values, which can diminish the sharpness of the image.

The median filter image, on the other hand, has reduced noise without the same level of blurring. The details and edges in the image are more preserved compared to the averaging filter. This is due to the median filter's ability to exclude the extreme outlier values that represent the noise. It does so by selecting the median value from the neighborhood of each pixel, which is less sensitive to outliers than the simple mean used by the averaging filter.

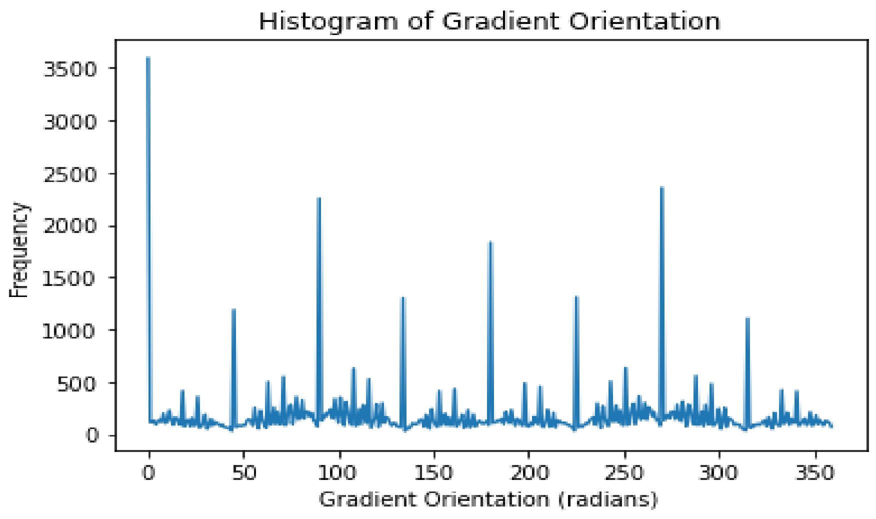
Question 4



This image shows the edges within the original image as detected by the Sobel operator, which is used to find the gradient magnitude of an image. By stretching the gradient magnitude, the contrast of the edges is enhanced to span the full range of 0 to 255, making the edges appear more distinct. The white lines indicate where the edges are sharpest or most pronounced.



The histogram illustrates the distribution of gradient magnitudes across the image. The x-axis represents the gradient magnitude, and the y-axis shows the frequency of each magnitude within the image. A sharp peak close to the origin suggests that there are many pixels with low gradient values, indicating areas of uniform intensity in the image. As the gradient magnitude increases, the frequency decreases, indicating fewer pixels with high contrast or strong edges.



This histogram representing the orientation of the gradients. The x-axis represents the angle of the gradient vector in radians, ranging from 0 to just under 2π radians. The y-axis shows how often each angle occurs. Spikes in the histogram indicate preferred orientations, which can be associated with the dominant directions of edges in the image. For example, vertical edges would produce a peak around 0 or π radians, and horizontal edges would produce a peak around $\pi/2$ or $3\pi/2$ radians.

Question 5

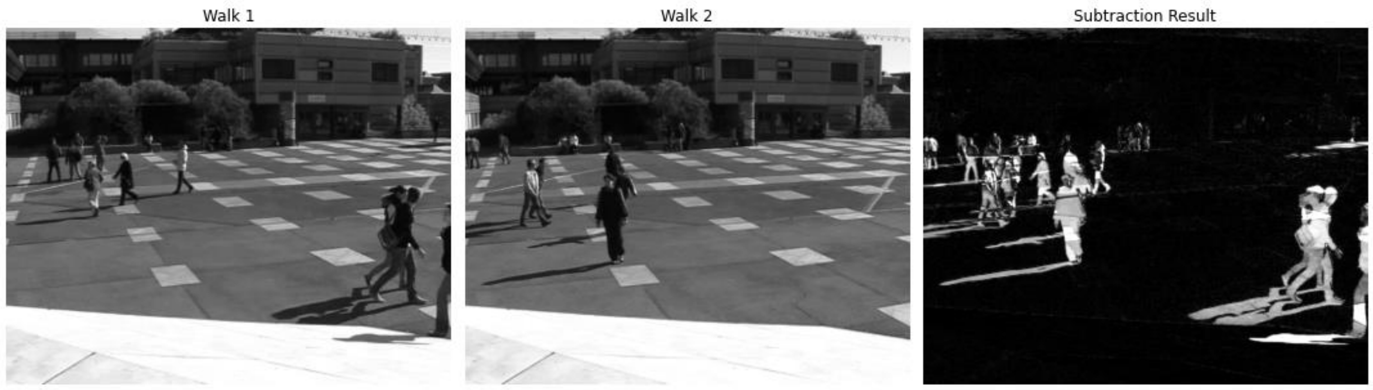
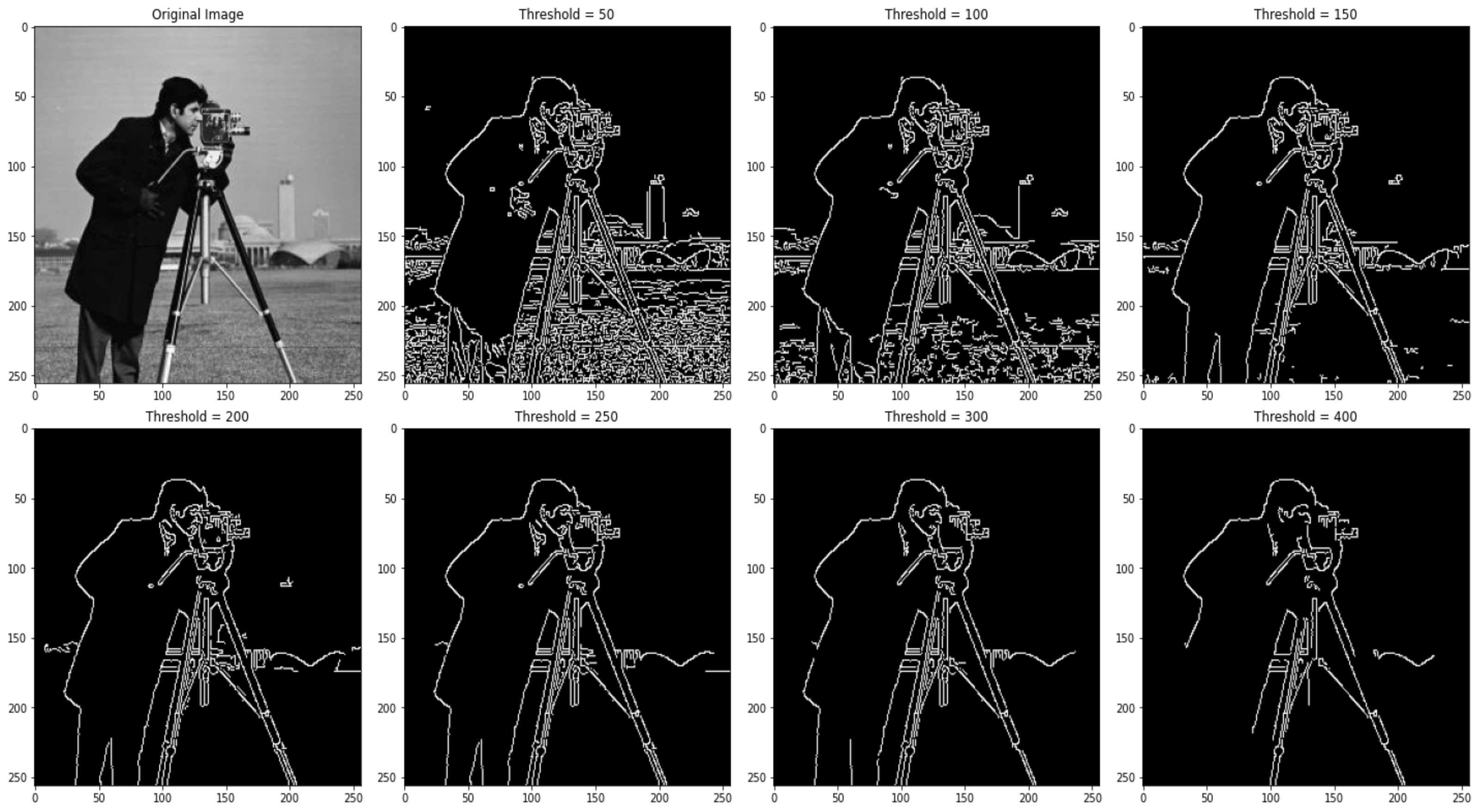


Image differencing is a technique used to find differences between two images. By converting images to grayscale and subtracting them, changes that have occurred between the captures of the two images can be highlighted. The result above shows the difference between the pixel values of the two images, and highlight those differences.

The first subplot shows the first image (walk_1) in grayscale. The second subplot shows the second image (walk_2) in grayscale. The third subplot shows the result of subtracting the second image from the first.

In the subtraction result, areas that appear darker indicate little to no change between the two images, while lighter areas represent regions where there has been more significant change. This technique is often used in motion detection, where differences between consecutive frames indicate movement.

Question 6



Explaining the results:

Low Thresholds (e.g., 50, 100): At lower threshold values, the algorithm is more sensitive and detects more edges. This includes weaker or finer edges, which may be noise or less significant features in the image. This can lead to a noisier edge image with many small edges that may not necessarily correspond to the actual edges in the real world.

Moderate Thresholds (e.g., 150, 200): As the threshold value increases, the algorithm starts to ignore weaker edges and only the stronger, more prominent edges are detected.

This usually results in a cleaner edge map that represents the significant features of the image.

High Thresholds (e.g., 250, 300, 400): Very high threshold values may cause the algorithm to miss even some significant edges, especially if they are not very pronounced. This can lead to an incomplete edge map where only the most distinct edges in the image are detected, and potentially important structural information may be lost.

From the results, it is observed that the progressive simplification of the edge map as the threshold increases. At the highest threshold values, only the most prominent edges of the figure and the tripod remain, while finer details of the background no longer visible. Very high threshold values can lead to loose important information.

Appendices

Q1 Part1:

```
from skimage import data, img_as_ubyte

import matplotlib.pyplot as plt

from skimage.transform import resize

import cv2

from skimage.util import random_noise

from skimage.filters.rank import mean

from skimage.morphology import square


# Load an example grayscale image

woman_img = cv2.imread('img.jpg', cv2.IMREAD_GRAYSCALE)


# Resize the woman image to 256x256 pixels

woman_img_resized = resize(woman_img, (256, 256), anti_aliasing=True)


# Convert the woman image to 8-bit

woman_image = img_as_ubyte(woman_img_resized)


# Load another example grayscale image

astronaut_img = data.astronaut()


# Convert the astronaut image to grayscale

astronaut_img_gray = cv2.cvtColor(astronaut_img, cv2.COLOR_RGB2GRAY)
```

```
# Resize the astronaut image to 256x256 pixels

astronaut_img_resized = resize(astronaut_img_gray, (256, 256), anti_aliasing=True)



# Convert the astronaut image to 8-bit

astronaut_image = img_as_ubyte(astronaut_img_resized)



# Display the images side by side

plt.figure(figsize=(12, 6))



plt.subplot(1, 2, 1)

plt.imshow(woman_image, cmap='gray')

plt.title('Original 8-bit Gray-Level Woman Image (256x256)')

plt.axis('off')



plt.subplot(1, 2, 2)

plt.imshow(astronaut_image, cmap='gray')

plt.title('Resized 8-bit Gray-Level Astronaut Image (256x256)')

plt.axis('off')



plt.show()
```

Q1 Part2:

```
import matplotlib.pyplot as plt

import numpy as np
```

```
# Apply power-law transformation with gamma=0.4
gamma = 0.4
transformed_woman_image = np.power(woman_image / 255.0, gamma) * 255.0
transformed_woman_image = np.uint8(transformed_woman_image)
transformed_astronaut_image = np.power(astronaut_image / 255.0, gamma) * 255.0
transformed_astronaut_image = np.uint8(transformed_astronaut_image)

# Display the original and transformed woman images side by side
plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.title('Original Woman Image')
plt.imshow(woman_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Transformed Woman Image (Gamma=0.4)')
plt.imshow(transformed_woman_image, cmap='gray')
plt.axis('off')

plt.show()

# Display the original and transformed astronaut images side by side
plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.title('Original Astronaut Image')

plt.imshow(astronaut_image, cmap='gray')

plt.axis('off')


plt.subplot(1, 2, 2)

plt.title('Transformed Astronaut Image (Gamma=0.4)')

plt.imshow(transformed_astronaut_image, cmap='gray')

plt.axis('off')


plt.show()
```

Q1 Part3:

```
# Zero-mean Gaussian noise with a variance of 40 to the original image


# Normalizing the variance because the image is 8-bit

variance = 40 / 255


#Add random noise ,the mode is set to 'gaussian', indicating that Gaussian noise will be
applied.

gaussian_noise_woman_image = random_noise(woman_image, mode='gaussian',
var=variance)

gaussian_noise_astronaut_image = random_noise(astronaut_image, mode='gaussian',
var=variance)


plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)

plt.imshow(gaussian_noise_woman_image, cmap='gray')

plt.title('Woman Image with Gaussian Noise (Variance=40)')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(gaussian_noise_astronaut_image, cmap='gray')

plt.title('Astronaut Image with Gaussian Noise (Variance=40)')

plt.axis('off')

plt.show()
```

Q1 Part4:

```
# Apply a 5x5 mean filter to the noisy woman image

mean_filtered_woman_image = cv2.blur(gaussian_noise_woman_image, (5, 5))


# Apply a 5x5 mean filter to the noisy astronaut image

mean_filtered_astronaut_image = cv2.blur(gaussian_noise_astronaut_image, (5, 5))



# Display the images

plt.figure(figsize=(12, 8))


plt.subplot(2, 3, 1)

plt.imshow(gaussian_noise_woman_image, cmap='gray')

plt.title('Noisy Woman Image (Variance=40)')
plt.axis('off')


plt.subplot(2, 3, 2)

plt.imshow(mean_filtered_woman_image, cmap='gray')

plt.title('Mean Filtered Woman Image')

plt.axis('off')


plt.subplot(2, 3, 3)

plt.imshow(gaussian_noise_woman_image - mean_filtered_woman_image, cmap='gray')

plt.title('Noisy - Mean Filtered Woman')

plt.axis('off')


plt.subplot(2, 3, 4)

plt.imshow(gaussian_noise_astronaut_image, cmap='gray')

plt.title('Noisy Astronaut Image (Variance=40)')

plt.axis('off')


plt.subplot(2, 3, 5)

plt.imshow(mean_filtered_astronaut_image, cmap='gray')

plt.title('Mean Filtered Astronaut Image')

plt.axis('off')


plt.subplot(2, 3, 6)

plt.imshow(gaussian_noise_astronaut_image - mean_filtered_astronaut_image,
cmap='gray')

plt.title('Noisy - Mean Filtered Astronaut')

plt.axis('off')
plt.tight_layout()

plt.show()
```

Q1 Part5:

```
# Add salt and pepper noise with density=0.1

noise_density = 0.1

salt_pepper_mask = np.random.rand(*woman_image.shape) < noise_density

salt_noise = salt_pepper_mask * 255

pepper_noise = salt_pepper_mask * 0


# Applying noise to the woman image

noisy_woman_image = np.clip(woman_image + salt_noise + pepper_noise, 0,
255).astype(np.uint8)


# Applying noise to the astronaut image

salt_pepper_mask_astronaut = np.random.rand(*astronaut_image.shape) < noise_density

salt_noise_astronaut = salt_pepper_mask_astronaut * 255

pepper_noise_astronaut = salt_pepper_mask_astronaut * 0

noisy_astronaut_image = np.clip(astronaut_image + salt_noise_astronaut +
pepper_noise_astronaut, 0, 255).astype(np.uint8)


# Apply a 7x7 median filter to the noisy images

median_filtered_woman_image = cv2.medianBlur(noisy_woman_image, 7)

median_filtered_astronaut_image = cv2.medianBlur(noisy_astronaut_image, 7)


# Displaying the images
```

```
plt.figure(figsize=(18, 12))


plt.subplot(2, 4, 1)
plt.title('Original Image - Woman')
plt.imshow(woman_image, cmap='gray')


plt.subplot(2, 4, 2)
plt.title('Noisy Image (Salt and Pepper) - Woman')
plt.imshow(noisy_woman_image, cmap='gray')


plt.subplot(2, 4, 3)
plt.title('Median Filter (7x7) - Woman')
plt.imshow(median_filtered_woman_image, cmap='gray')


plt.subplot(2, 4, 4)
plt.title('Difference (Noisy - Median Filtered) - Woman')
plt.imshow(noisy_woman_image - median_filtered_woman_image, cmap='gray')


plt.subplot(2, 4, 5)
plt.title('Original Image - Astronaut')
plt.imshow(astronaut_image, cmap='gray')


plt.subplot(2, 4, 6)
plt.title('Noisy Image (Salt and Pepper) - Astronaut')
plt.imshow(noisy_astronaut_image, cmap='gray')
plt.subplot(2, 4, 7)
plt.title('Median Filter (7x7) - Astronaut')
plt.imshow(median_filtered_astronaut_image, cmap='gray')

plt.subplot(2, 4, 8)
plt.title('Difference (Noisy - Median Filtered) - Astronaut')
plt.imshow(noisy_astronaut_image - median_filtered_astronaut_image, cmap='gray')

plt.tight_layout()
plt.show()
```

Q1 Part6:

```
# Apply a 7x7 mean filter
mean_filter_image_1 = cv2.blur(noisy_woman_image, (7, 7))

mean_filter_image_2 = cv2.blur(noisy_astronaut_image, (7, 7))

# Displaying the images
plt.figure(figsize=(18, 12))

plt.subplot(2, 4, 1)
plt.title('Original Image - Woman')
plt.imshow(woman_image, cmap='gray')

plt.subplot(2, 4, 2)
plt.title('Noisy Image (Salt and Pepper) - Woman')
plt.imshow(noisy_woman_image, cmap='gray')


plt.subplot(2, 4, 3)

plt.title('Mean Filter (7x7) - Woman')

plt.imshow(mean_filter_image_1, cmap='gray')


plt.subplot(2, 4, 4)

plt.title('Difference (Noisy - Mean Filtered) - Woman')

plt.imshow(noisy_woman_image - mean_filter_image_1, cmap='gray')


plt.subplot(2, 4, 5)

plt.title('Original Image - Astronaut')

plt.imshow(astronaut_image, cmap='gray')


plt.subplot(2, 4, 6)

plt.title('Noisy Image (Salt and Pepper) - Astronaut')

plt.imshow(noisy_astronaut_image, cmap='gray')


plt.subplot(2, 4, 7)

plt.title('Mean Filter (7x7) - Astronaut')

plt.imshow(mean_filter_image_2, cmap='gray')


plt.subplot(2, 4, 8)

plt.title('Difference (Noisy - Mean Filtered) - Astronaut')

plt.imshow(noisy_astronaut_image - mean_filter_image_2, cmap='gray')
plt.tight_layout()

plt.show()
```

Q1 Part7:

```
# Sobel kernel for horizontal and vertical edges

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

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


# Convolution function

def convolve(image, kernel):

    # Get the height and width of the input

    height, width = image.shape


    # Get the size of the kernel

    kernel_size = len(kernel)


    # Initialize the output array with zeros

    output = np.zeros((height - kernel_size + 1, width - kernel_size + 1))


    # Iterate over the pixels of the output array

    for i in range(output.shape[0]):

        for j in range(output.shape[1]):

            # Convolution operation by element-wise multiplication and summation

            output[i, j] = np.sum(image[i:i + kernel_size, j:j + kernel_size] * kernel)


    return output


# Apply Sobel filter for horizontal and vertical edges
```

```
sobel_x_woman = convolve(woman_image, sobel_kernel_x)
sobel_y_woman = convolve(woman_image, sobel_kernel_y)

sobel_x_astronaut = convolve(astronaut_image, sobel_kernel_x)
sobel_y_astronaut = convolve(astronaut_image, sobel_kernel_y)



# Combine horizontal and vertical responses
sobel_response_woman = np.sqrt(sobel_x_woman**2 + sobel_y_woman**2)

sobel_response_astronaut = np.sqrt(sobel_x_astronaut**2 + sobel_y_astronaut**2)

# Displaying the results for both images using plt.subplot
plt.figure(figsize=(18, 8))

# Display for astronaut image
plt.subplot(2, 4, 1)
plt.imshow(astronaut_image, cmap='gray')
plt.title('Original Image - Astronaut')
plt.axis('off')

plt.subplot(2, 4, 2)
plt.imshow(sobel_x_astronaut, cmap='gray')
plt.title('Sobel Gx - Astronaut')
plt.axis('off')
plt.subplot(2, 4, 3)

plt.imshow(sobel_y_astronaut, cmap='gray')

plt.title('Sobel Gy - Astronaut')

plt.axis('off')


plt.subplot(2, 4, 4)

plt.imshow(sobel_response_astronaut, cmap='gray')

plt.title('Sobel G (Gx + Gy) - Astronaut')

plt.axis('off')


# Display for woman image

plt.subplot(2, 4, 5)

plt.imshow(woman_image, cmap='gray')

plt.title('Original Image - Woman')

plt.axis('off')


plt.subplot(2, 4, 6)

plt.imshow(sobel_x_woman, cmap='gray')

plt.title('Sobel Gx - Woman')

plt.axis('off')


plt.subplot(2, 4, 7)

plt.imshow(sobel_y_woman, cmap='gray')

plt.title('Sobel Gy - Woman')

plt.axis('off')
plt.subplot(2, 4, 8)

plt.imshow(sobel_response_woman, cmap='gray')

plt.title('Sobel G (Gx + Gy) - Woman')

plt.axis('off')


plt.tight_layout()

plt.show()
```

Q2:

```
import matplotlib.pyplot as plt

import cv2

import numpy as np


# Load the images

house1_img = cv2.imread('House1.jpg', cv2.IMREAD_GRAYSCALE)

house2_img = cv2.imread('House2.jpg', cv2.IMREAD_GRAYSCALE)


# Apply the convolution filter using the filter2D function from OpenCV

def myImageFilter(input_image, kernel):

    # The function uses a border type that reflects the border elements of the image

    result_image = cv2.filter2D(input_image, -1, kernel,
borderType=cv2.BORDER_REFLECT)

    return result_image




# Define the kernels

averaging_kernel_3x3 = np.ones((3, 3), np.float32) / 9
```

```
averaging_kernel_5x5 = np.ones((5, 5), np.float32) / 25

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

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

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

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


# Function to generate Gaussian kernel

def gaussian_kernel(size, sigma=1):

    size = int(size) // 2 * 2 + 1  # Ensure odd size for symmetry

    x, y = np.mgrid[-size//2:size//2+1, -size//2:size//2+1]

    normal = 1 / (2.0 * np.pi * sigma**2)

    g = np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal

    return g


# Function to display images

def display_images(original, filtered, title1='Original', title2='Filtered'):

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)

    plt.imshow(original, cmap='gray')

    plt.title(title1)

    plt.axis('off')


    plt.subplot(1, 2, 2)

    plt.imshow(filtered, cmap='gray')

    plt.title(title2)

    plt.axis('off')
    plt.show()


# Apply 3x3 averaging filter to House1 and display
house1_avg_3x3 = myImageFilter(house1_img, averaging_kernel_3x3)
house1_avg_3x3_normalized = cv2.normalize(house1_avg_3x3, None, 0, 255,
cv2.NORM_MINMAX)


display_images(house1_img, house1_avg_3x3_normalized, 'House1 Original', 'House1
with 3x3 Averaging')


# Apply 5x5 averaging filter to House1 and display
house1_avg_5x5 = myImageFilter(house1_img, averaging_kernel_5x5)
house1_avg_5x5_normalized = cv2.normalize(house1_avg_5x5, None, 0, 255,
cv2.NORM_MINMAX)


display_images(house1_img, house1_avg_5x5_normalized, 'House1 Original', 'House1
with 5x5 Averaging')



# Apply 3x3 averaging filter to House2 and display
house2_avg_3x3 = myImageFilter(house2_img, averaging_kernel_3x3)
house2_avg_3x3_normalized = cv2.normalize(house2_avg_3x3, None, 0, 255,
cv2.NORM_MINMAX)


display_images(house2_img, house2_avg_3x3_normalized, 'House2 Original', 'House2
with 3x3 Averaging')


# Apply 5x5 averaging filter to House2 and display
```

```
house2_avg_5x5 = myImageFilter(house2_img, averaging_kernel_5x5)

house2_avg_5x5_normalized = cv2.normalize(house2_avg_5x5, None, 0, 255,
cv2.NORM_MINMAX)

display_images(house2_img, house2_avg_5x5_normalized, 'House2 Original', 'House2
with 5x5 Averaging')




# Function to display images with Gaussian kernel heatmap

def display1(original, filtered, kernel, title1='Original', title2='Filtered', title3='Kernel'):

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)

    plt.imshow(original, cmap='gray')

    plt.title(title1)

    plt.axis('off')


    plt.subplot(1, 3, 2)

    plt.imshow(filtered, cmap='gray')

    plt.title(title2)

    plt.axis('off')


    plt.subplot(1, 3, 3)

    plt.imshow(kernel, cmap='hot')

    plt.title(title3)

    plt.axis('off')


    plt.show()
```

```
# Apply Gaussian filter with sigma=1, 2, 3 to House1 and House 2 and display

sigmas = [1, 2, 3]

for sigma in sigmas:

    # (6*sigma + 1) x (6*sigma + 1) kernel

    size = int(2*np.ceil(2*sigma)+1)

    gaussian_kernel_sigma = gaussian_kernel(size, sigma)

    house1_gaussian = myImageFilter(house1_img, gaussian_kernel_sigma)

    house1_gaussian_normalized = cv2.normalize(house1_gaussian, None, 0, 255,
cv2.NORM_MINMAX)

    display1(house1_img, house1_gaussian_normalized, gaussian_kernel_sigma, 'House1
Original', f'House1 with Gaussian σ={sigma}', f'Gaussian Kernel σ={sigma}')


for sigma in sigmas:

    size = int(2*np.ceil(2*sigma)+1)

    gaussian_kernel_sigma = gaussian_kernel(size, sigma)

    house2_gaussian = myImageFilter(house2_img, gaussian_kernel_sigma)

    house2_gaussian_normalized = cv2.normalize(house2_gaussian, None, 0, 255,
cv2.NORM_MINMAX)

    display1(house2_img, house2_gaussian_normalized, gaussian_kernel_sigma, 'House2
Original', f'House2 with Gaussian σ={sigma}', f'Gaussian Kernel σ={sigma}')




# Function to display four images side by side

def display_four_images(img1, img2, img3, img4, title1='Image 1', title2='Image 2',
title3='Image 3', title4='Image 4'):

    plt.figure(figsize=(20, 10))  # Adjust the size to fit four images
    plt.subplot(1, 4, 1)
    plt.imshow(img1, cmap='gray')
    plt.title(title1)
    plt.axis('off')

    plt.subplot(1, 4, 2)
    plt.imshow(img2, cmap='gray')
    plt.title(title2)
    plt.axis('off')

    plt.subplot(1, 4, 3)
    plt.imshow(img3, cmap='gray')
    plt.title(title3)
    plt.axis('off')

    plt.subplot(1, 4, 4)
    plt.imshow(img4, cmap='gray')
    plt.title(title4)
    plt.axis('off')

    plt.show()


def normalize_and_clip_image(image):
    if image.size == 0:
        return image
    image = image - image.min()

    image = image / image.max()

    image = (image * 255).astype(np.uint8)

    return image



# Function to apply edge detection

def apply_edge_detection(input_image, kernel_x, kernel_y, title):

    edge_x = myImageFilter(input_image, kernel_x)

    edge_y = myImageFilter(input_image, kernel_y)

    edge_combined = np.hypot(edge_x, edge_y)

    edge_combined = normalize_and_clip_image(edge_combined)

    display_four_images(input_image, edge_x, edge_y, edge_combined,

                         f'{title} Original', f'{title} Edge X', f'{title} Edge Y', f'{title} Edge
Combined')



apply_edge_detection(house1_img, sobel_kernel_x, sobel_kernel_y, 'House1 Sobel')

apply_edge_detection(house2_img, sobel_kernel_x, sobel_kernel_y, 'House2 Sobel')

apply_edge_detection(house1_img, prewitt_kernel_x, prewitt_kernel_y, 'House1
Prewitt')

apply_edge_detection(house2_img, prewitt_kernel_x, prewitt_kernel_y, 'House2
Prewitt')

# Apply Sobel and Prewitt filters on normalized images

house1_sobel_x = myImageFilter(house1_img, sobel_kernel_x)

house1_sobel_y = myImageFilter(house1_img, sobel_kernel_y)

house2_sobel_x = myImageFilter(house2_img, sobel_kernel_x)
```

```
house2_sobel_y = myImageFilter(house2_img, sobel_kernel_y)


# Apply Prewitt edge detection to House1 and display

house1_prewitt_x = myImageFilter(house1_img, prewitt_kernel_x)

house1_prewitt_y = myImageFilter(house1_img, prewitt_kernel_y)

house2_prewitt_x = myImageFilter(house2_img, prewitt_kernel_x)

house2_prewitt_y = myImageFilter(house2_img, prewitt_kernel_y)


# Calculate the gradient magnitude for Sobel

house1_sobel_combined = np.sqrt(house1_sobel_x**2 + house1_sobel_y**2)

house2_sobel_combined = np.sqrt(house2_sobel_x**2 + house2_sobel_y**2)


# Calculate the gradient magnitude for Prewitt

house1_prewitt_combined = np.sqrt(house1_prewitt_x**2 + house1_prewitt_y**2)

house2_prewitt_combined = np.sqrt(house2_prewitt_x**2 + house2_prewitt_y**2)


# Normalize the images

house1_sobel_x_normalized = cv2.normalize(house1_sobel_x, None, 0, 255,
cv2.NORM_MINMAX)

house1_sobel_y_normalized = cv2.normalize(house1_sobel_y, None, 0, 255,
cv2.NORM_MINMAX)

house1_sobel_combined_normalized = cv2.normalize(house1_sobel_combined, None, 0,
255, cv2.NORM_MINMAX)


house1_prewitt_x_normalized = cv2.normalize(house1_prewitt_x, None, 0, 255,
cv2.NORM_MINMAX)

house1_prewitt_y_normalized = cv2.normalize(house1_prewitt_y, None, 0, 255,
cv2.NORM_MINMAX)
```

```
house1_prewitt_combined_normalized = cv2.normalize(house1_prewitt_combined,
None, 0, 255, cv2.NORM_MINMAX)



house2_sobel_x_normalized = cv2.normalize(house2_sobel_x, None, 0, 255,
cv2.NORM_MINMAX)

house2_sobel_y_normalized = cv2.normalize(house2_sobel_y, None, 0, 255,
cv2.NORM_MINMAX)

house2_sobel_combined_normalized = cv2.normalize(house2_sobel_combined, None, 0,
255, cv2.NORM_MINMAX)



house2_prewitt_x_normalized = cv2.normalize(house2_prewitt_x, None, 0, 255,
cv2.NORM_MINMAX)

house2_prewitt_y_normalized = cv2.normalize(house2_prewitt_y, None, 0, 255,
cv2.NORM_MINMAX)

house2_prewitt_combined_normalized = cv2.normalize(house2_prewitt_combined,
None, 0, 255, cv2.NORM_MINMAX)



# Display Sobel results for img1

display_four_images(house1_img, house1_sobel_x_normalized,
house1_sobel_y_normalized, house1_sobel_combined_normalized, 'House1 Original',
'House1 Sobel X', 'House1 Sobel Y', 'House1 Sobel Combined')



# Display Prewitt results  for img1

display_four_images(house1_img, house1_prewitt_x_normalized,
house1_prewitt_y_normalized, house1_prewitt_combined_normalized, 'House1
Original', 'House1 Prewitt X', 'House1 Prewitt Y', 'House1 Prewitt Combined')



# Display Sobel results  for img2

display_four_images(house2_img, house2_sobel_x_normalized,
house2_sobel_y_normalized, house2_sobel_combined_normalized, 'House2 Original',
'House2 Sobel X', 'House2 Sobel Y', 'House2 Sobel Combined')
```

```
# Display Prewitt results  for img2

display_four_images(house2_img, house1_prewitt_x_normalized,
house2_prewitt_y_normalized, house2_prewitt_combined_normalized, 'House2
Original', 'House2 Prewitt X', 'House2 Prewitt Y', 'House2 Prewitt Combined')
```

Q3:

```
import matplotlib.pyplot as plt

import cv2




# Read the noisy images

image1 = cv2.imread('Noisyimage1.jpg', cv2.IMREAD_GRAYSCALE)

image2 = cv2.imread('Noisyimage2.jpg', cv2.IMREAD_GRAYSCALE)


# Apply 5 by 5 Averaging filter

averaging_filtered1 = cv2.blur(image1, (5, 5))

averaging_filtered2 = cv2.blur(image2, (5, 5))


# Apply 5 by 5 Median filter

median_filtered1 = cv2.medianBlur(image1, 5)

median_filtered2 = cv2.medianBlur(image2, 5)


# Display the original and filtered images

plt.figure(figsize=(10, 8))


plt.subplot(231), plt.imshow(image1, cmap='gray'), plt.title('Original Image 1')

plt.subplot(232), plt.imshow(averaging_filtered1, cmap='gray'), plt.title('Averaging
Filtered Image 1')
```

```
plt.subplot(233), plt.imshow(median_filtered1, cmap='gray'), plt.title('Median Filtered Image 1')


plt.subplot(234), plt.imshow(image2, cmap='gray'), plt.title('Original Image 2')

plt.subplot(235), plt.imshow(averaging_filtered2, cmap='gray'), plt.title('Averaging Filtered Image 2')

plt.subplot(236), plt.imshow(median_filtered2, cmap='gray'), plt.title('Median Filtered Image 2')


plt.show()
```

Q4:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt




# Load the image

image1 = cv2.imread('Q_4.jpg', cv2.IMREAD_GRAYSCALE)


# Compute gradient magnitude using Sobel gradients

gradient_x = cv2.Sobel(image1, cv2.CV_64F, 1, 0, ksize=3)

gradient_y = cv2.Sobel(image1, cv2.CV_64F, 0, 1, ksize=3)

gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)


# Stretch the gradient magnitude for better visualization
```

```
gradient_magnitude_stretched = cv2.normalize(gradient_magnitude, None, 0, 255, cv2.NORM_MINMAX)



# Display the stretched gradient magnitude

plt.imshow(gradient_magnitude_stretched, cmap='gray')

plt.title('Stretched Gradient Magnitude')

plt.show()



# Compute histogram of gradient magnitude

hist, bins = np.histogram(gradient_magnitude, bins=256, range=[0, 256])


# Display histogram of gradient magnitude

plt.plot(hist)

plt.title('Histogram of Gradient Magnitude')

plt.xlabel('Gradient Magnitude')

plt.ylabel('Frequency')

plt.show()


# Compute gradient orientation

gradient_orientation = np.arctan2(gradient_y, gradient_x)


# Convert angles to the range [0, 2*pi)

gradient_orientation = (gradient_orientation + 2 * np.pi) % (2 * np.pi)


# Compute histogram of gradient orientation
```

```
hist_orientation, bins_orientation = np.histogram(gradient_orientation, bins=360,
range=[0, 2*np.pi])


# Display histogram of gradient orientation
plt.plot(hist_orientation)
plt.title('Histogram of Gradient Orientation')
plt.xlabel('Gradient Orientation (radians)')
plt.ylabel('Frequency')
plt.show()
```

Q5:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt


# Load the images
walk_1_path = 'walk_1.jpg'
walk_2_path = 'walk_2.jpg'


# Check if the files exist and read them
try:
    walk_1 = cv2.imread(walk_1_path)
    walk_2 = cv2.imread(walk_2_path)
    if walk_1 is None or walk_2 is None:
        raise ValueError("One of the images didn't load correctly. Check the file paths.")
except Exception as e:
    error_message = str(e)
```

```
# If images are loaded successfully
if 'error_message' not in locals():
    # Convert images to grayscale
    walk_1_gray = cv2.cvtColor(walk_1, cv2.COLOR_BGR2GRAY)
    walk_2_gray = cv2.cvtColor(walk_2, cv2.COLOR_BGR2GRAY)

    # Ensure the images have the same size
    walk_2_gray = cv2.resize(walk_2_gray, (walk_1_gray.shape[1],
walk_1_gray.shape[0]))

    # Subtract walk_2.jpg from walk_1.jpg
    result = cv2.absdiff(walk_1_gray, walk_2_gray)

    # Normalize the result to ensure visibility
    result_normalized = cv2.normalize(result, None, 0, 255, cv2.NORM_MINMAX)

    # Create a gray background image with the same size as the result
    background = np.full_like(result_normalized, 200, dtype=np.uint8)

    # Combine the subtracted result with the background
    result_with_background = cv2.addWeighted(result_normalized, 1, background, 0.7,
0.9)

    # Prepare to show the images using matplotlib
    images_to_show = {
        'Walk 1': walk_1_gray,
        'Walk 2': walk_2_gray,
        'Subtraction Result': result_with_background
    }
else:
    images_to_show = {'Error': error_message}

# Display the images using matplotlib
plt.figure(figsize=(15, 5))
for i, (title, img) in enumerate(images_to_show.items()):
    plt.subplot(1, len(images_to_show), i + 1)
    plt.imshow(img, cmap='gray')
    plt.title(title)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Q6:

```
import cv2
import matplotlib.pyplot as plt

# Load the image in grayscale
image1 = cv2.imread('Q_4.jpg', cv2.IMREAD_GRAYSCALE)

# Test different values of 'Threshold'
threshold_values = [50, 100, 150, 200, 250, 300, 350, 400]
```

```
# Create subplots for better display
rows, cols = 2, 4
plt.figure(figsize=(20, 10))

# Display the original image
plt.subplot(rows, cols, 1)
plt.imshow(image1, cmap='gray')
plt.title('Original Image')

# Apply Canny edge detector for different thresholds
for i, threshold in enumerate(threshold_values):
    edges = cv2.Canny(image1, threshold, 2 * threshold)

    # Display the result
    plt.subplot(rows, cols, i + 2 if i < 7 else 8)
    plt.imshow(edges, cmap='gray')
    plt.title(f'Threshold = {threshold}')

# Show the plots
plt.tight_layout()
plt.show()
```