Computer Vision
Exercise 2

Student:
Alireza Amini

1 Edge Detection

In this part I performed edge detection on the images using Prewitt, Kirsch, Marr-Hildreth, and Canny edge detection algorithms.

1.1 Prewitt

To implement the Prewitt edge detection algorithm, I first constructed two 3x3 convolution masks to compute gradients in both horizontal and vertical directions. I then utilized the cv2.filter2D function to apply these masks to the input image, generating separate gradient images. Next, I calculated the magnitude of each pixel's gradient using the combined horizontal and vertical gradient values. This magnitude represents the strength of the edge at that pixel. In the final step, I experimented with various thresholds applied to the gradient magnitude image to isolate significant edges. A threshold value of 50 proved optimal for producing clear and distinct edges within this implementation.

```
def Prewitt(image):
    kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
    kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
    img_prewittx = cv2.filter2D(image, -1, kernelx)
    img_prewitty = cv2.filter2D(image, -1, kernely)
    img_prewitt = img_prewittx + img_prewitty
    img_prewitt = np.uint8(img_prewitt > 50) * 255
    return img_prewitt
```

Figure 1: Implementation of Prewitt

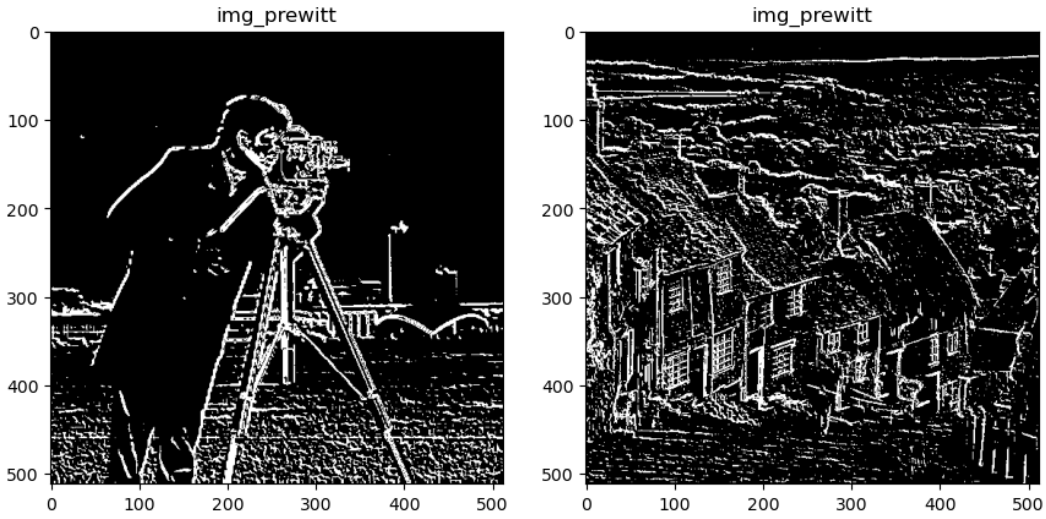

Figure 2: Prewitt on the Camera Man and Goldhill Images

1.2 Kirsch

To implement the Kirsch edge detection algorithm, I first constructed eight 3x3 convolution masks. Each mask is unique, oriented at 45-degree intervals to capture gradients in different directions. Next, I convolved each mask with the input image, using the cv2.filter2D function. This yielded a set of eight "suspicion levels" for each pixel - the gradient magnitudes from each mask. I then identified the maximum gradient across all eight convolutions for each pixel. Finally, I applied a threshold of 210 to extract the edge-detected image.

```python
def Kirsch(image):
    # Kirsch maskes
    kirsch_masks = [
    np.array([[5, 5, 5], [-3, 0, -3], [-3, -3, -3]]),
    np.array([[-3, 5, 5], [-3, 0, 5], [-3, -3, -3]]),
    np.array([[-3, -3, 5], [-3, 0, 5], [-3, -3, 5]]),
    np.array([[-3, -3, -3], [-3, 0, -3], [5, 5, 5]]),
    np.array([[-3, -3, -3], [5, 0, -3], [5, 5, -3]]),
    np.array([[-3, -3, -3], [-3, 0, 5], [-3, 5, 5]]),
    np.array([[5, -3, -3], [5, 0, -3], [5, -3, -3]]),
    np.array([[5, 5, -3], [5, 0, -3], [-3, -3, -3]])
    ]
    # Apply Kirsch masks
    kirsch_responses = [np.abs(cv2.filter2D(image, -1, mask)) for mask in kirsch_masks]
    # Calculate edge strength
    edge_strength = np.max(np.stack(kirsch_responses, axis=-1), axis=-1)
    # Threshold for visualization
    edges = np.uint8(edge_strength > 210) * 255
    return edges
```

Figure 3: Implementation of Kirsch

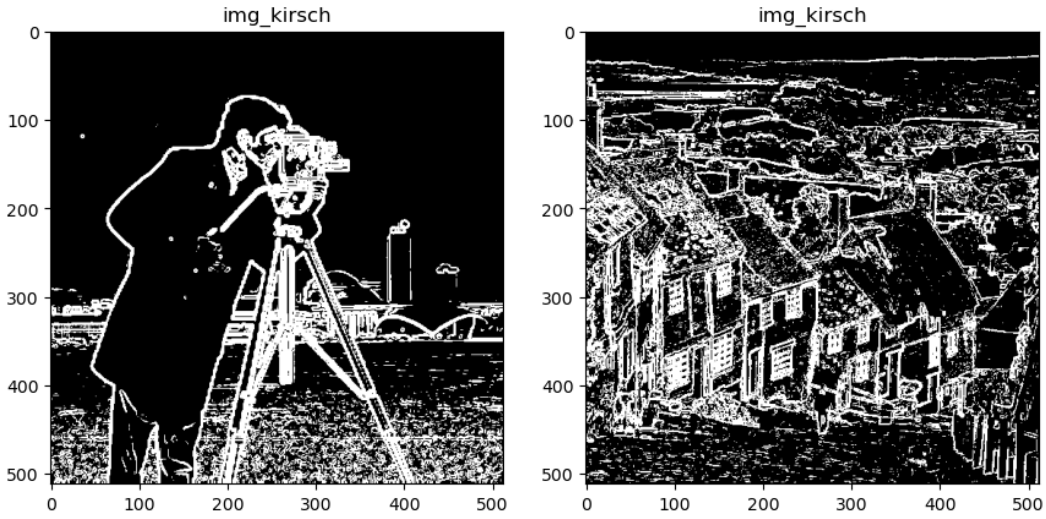

Figure 4: Kirsch on the Camera Man and Goldhill Images

1.3 Marr-Hildreth

For the Marr-Hildreth algorithm, a Gaussian filter is first applied to remove noise, then the second derivative or Laplacian is applied. Subsequently, zero-crossing operation is applied to the second derivative image to obtain stronger edges, and finally, edges are obtained by thresholding.

```python
def Mar_Hildreth(image):
    blurred = cv2.GaussianBlur(image, (5, 5), 0)
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)

    edges = cv2.morphologyEx(np.uint8(np.abs(laplacian)), cv2.MORPH_CLOSE, np.ones((3, 3)))
    edges = np.uint8(edges > 10) * 255
    return edges
```

Figure 5: Implementation of Marr-Hildreth

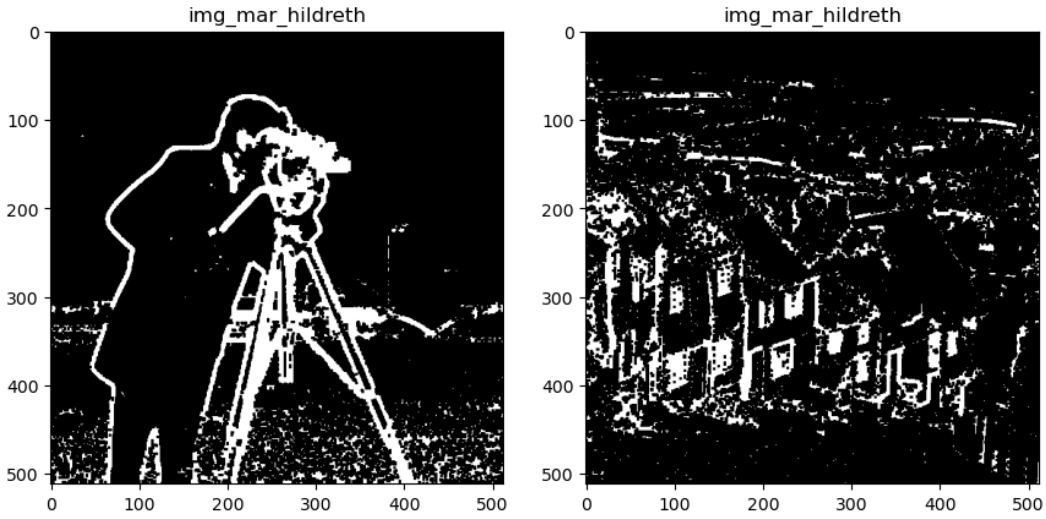

Figure 6: Marr-Hildreth on the Camera Man and Goldhill Images

1.4 Canny

To implement the Canny edge detection algorithm, I utilized the **cv2.Canny** function, which efficiently transforms an input image into its edge-detected counterpart. I choosed 100 and 200 as the minVal and maxVal parameters which are used in the double threshold step. The image is first convolved with a Gaussian filter to rubdue noise. Next, the gradient of the image is calculated using Sobel operators and the gradient magnitude and orientation, too Non-Maximum Suppression then refines the edges, preserving only the local maxima in the gradient direction to ensure that each edge is represented by a single, distinct line. A double thresholding process decisively classifies pixels based on their gradient magnitudes: those exceeding a high threshold are deemed strong edges, while those falling below a low threshold are dismissed as non-edges. Finally, weak edges are given a chance to prove their worth. Only those gracefully connecting to strong edges are granted a place in the final edge-detected image, resulting in a cohesive and refined representation of the image's edges.

```
def Canny(image):
    img_canny = cv2.Canny(image,100,200)
    return img_canny
```

Figure 7: Implementation of Canny

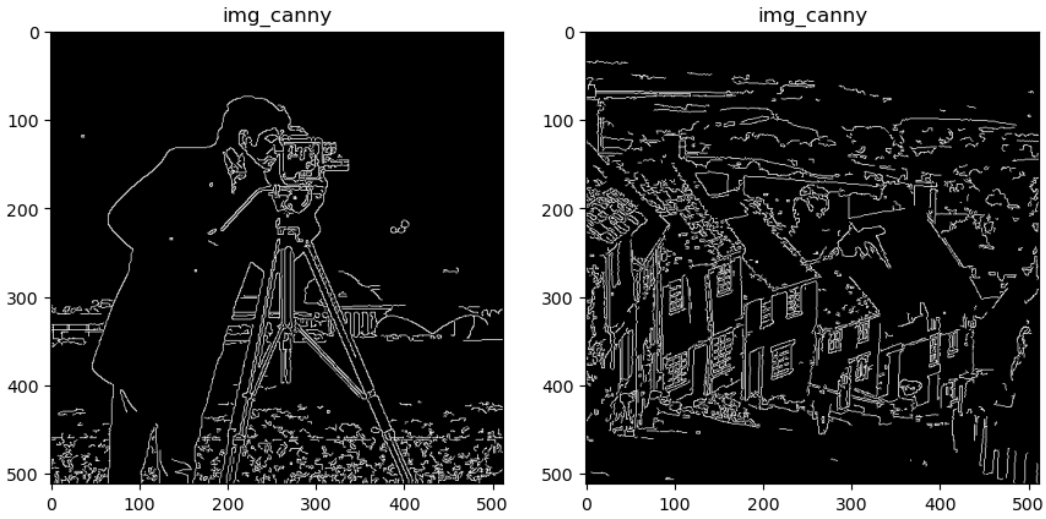

Figure 8: Canny on the Camera Man and Goldhill Images

By visually comparing the edges obtained from the four applied algorithms on the images, it can be concluded that the Canny algorithm is less affected by noise present in the images compared to the other algorithms. It also detects better and thinner edges, which is due to the Gaussian filter applied in the first step of the algorithm as well as non-maximum suppression and double thresholding aiding in finding thinner and better edges.

2 Color Segmentation

In this section, I first convert the image to the HSV color space. Then, I feed the H, S, and V values to the **cv2.kmeans** function to perform image segmentation by clustering the image pixels. For each image, we used k values of 8, 16, and 32 as the number of clusters. The results are shown below:

```
path = "/content/drive/MyDrive/images/2/"
colors = [8, 16, 32]
for i in range(1, 4):
  image_path = path + str(i) + ".jpg"
  image = cv2.imread(image_path)
  height, width, _ = image.shape
  plt_imshow("Original", image)
  img = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
  h = img[:, :, 0]
  s = img[:, :, 1]
  v = img[:, :, 2]
  h_color = h.reshape((-1, 1))
  s_color = s.reshape((-1, 1))
  v_color = v.reshape((-1, 1))
  Z = np.column_stack((h_color, s_color, v_color))
  # Convert Z to float32
  Z = np.float32(Z)
  # Set K-means criteria
  criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
  for color in colors:
    # Perform K-means clustering
    segmented, label, center = cv2.kmeans(Z, color, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
    center = np.uint8(center)
    segmented = center[label.flatten()]
    segmented = segmented.reshape((height, width, 3))[:, :, :3]
    segmented = cv2.cvtColor(segmented, cv2.COLOR_HSV2RGB)
    plt_imshow("Segmented using {} Colors".format(color), segmented)
    print("PSNR: {}".format(psnr(img, segmented)))
    print("MSE: {}".format(mse(img, segmented)))
```

Figure 9: Color Segmentation Implementation

For segmented images, PSNR and MSE metrics were calculated. It was expected that as the number of segments increased, the value of MSI would decrease and the value of PSNR would increase, which only happened for the first image. This may be because the first image has many colors, so increasing the number of segments makes the segmented image closer to the original. However, for the second and third images, it produces additional colors. Another reason could be that segmentation was not done properly with suitable methods or features, and PSNR and MSE mertrics may not be good indicators of segmentation quality.

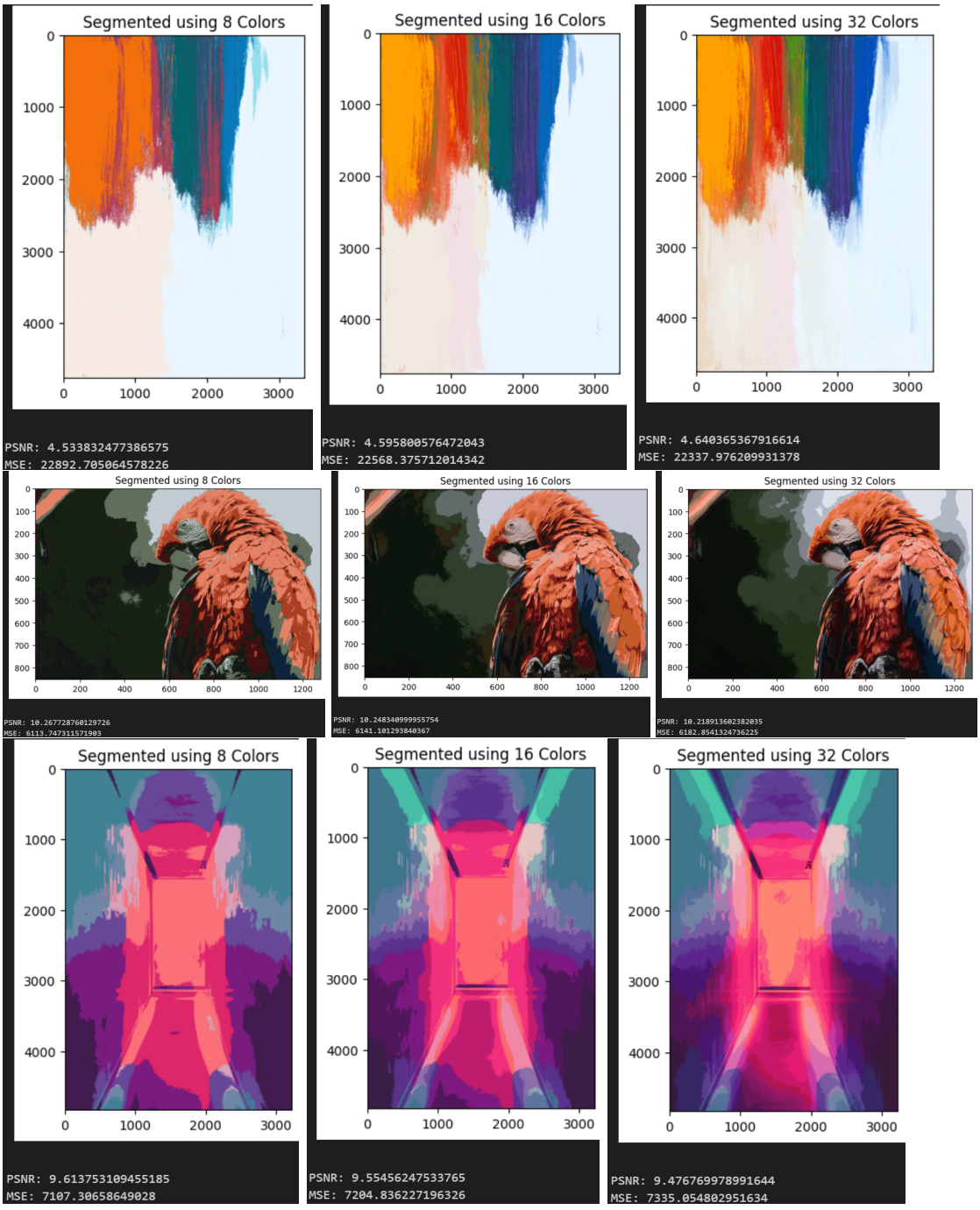

Figure 10: Results of Color Segmentation

3 Texture-based Segmentation

In this section, I first created a bank of Gabor filters using the cv2.getGaborKernel function with different parameters such as theta, sigma, and gamma. Then, I convolved them with the original image, and finally, I fed the resulting convolved images to the K-means algorithm for clustering the pixels based on the detected textures to segment them into different regions. As seen in the output, each image is divided into multiple regions in terms of texture to distinguish between different areas.

```python
def texture_segment(image, K):
    image1 = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    df = pd.DataFrame()
    num = 1
    for theta in range(2):
        thete = theta / 4. *np.pi
        for sigma in (3, 5):
            for lamda in np.arange(0, np.pi, np.pi /4.):
                for gamma in (0.8, 0.9):
                    gabor_label = 'Gabor' + str(num)
                    kernel = cv2.getGaborKernel((3, 3), sigma, theta, lamda, gamma, 0, ktype=cv2.CV_32F)
                    fimage = cv2.filter2D(image1, cv2.CV_8UC3, kernel)
                    filterd_image = fimage.reshape(-1)
                    df[gabor_label] = filterd_image
                    num += 1

    features = np.float32(df.to_numpy())
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    segmented, label = cv2.kmeans(features, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
    segmented = np.round(label.flatten().reshape((image1.shape[0], image1.shape[1])) * (255 / (K - 1)))

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8), sharex=True, sharey=True)
    ax1.axis('off')
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    ax1.imshow(image)
    ax1.set_title('Original image')
    ax2.axis('off')
    ax2.imshow(segmented, cmap='gray')
    ax2.set_title('Segmented')
    plt.show()
```

Figure 11: Implementation of texture-based Segmentation

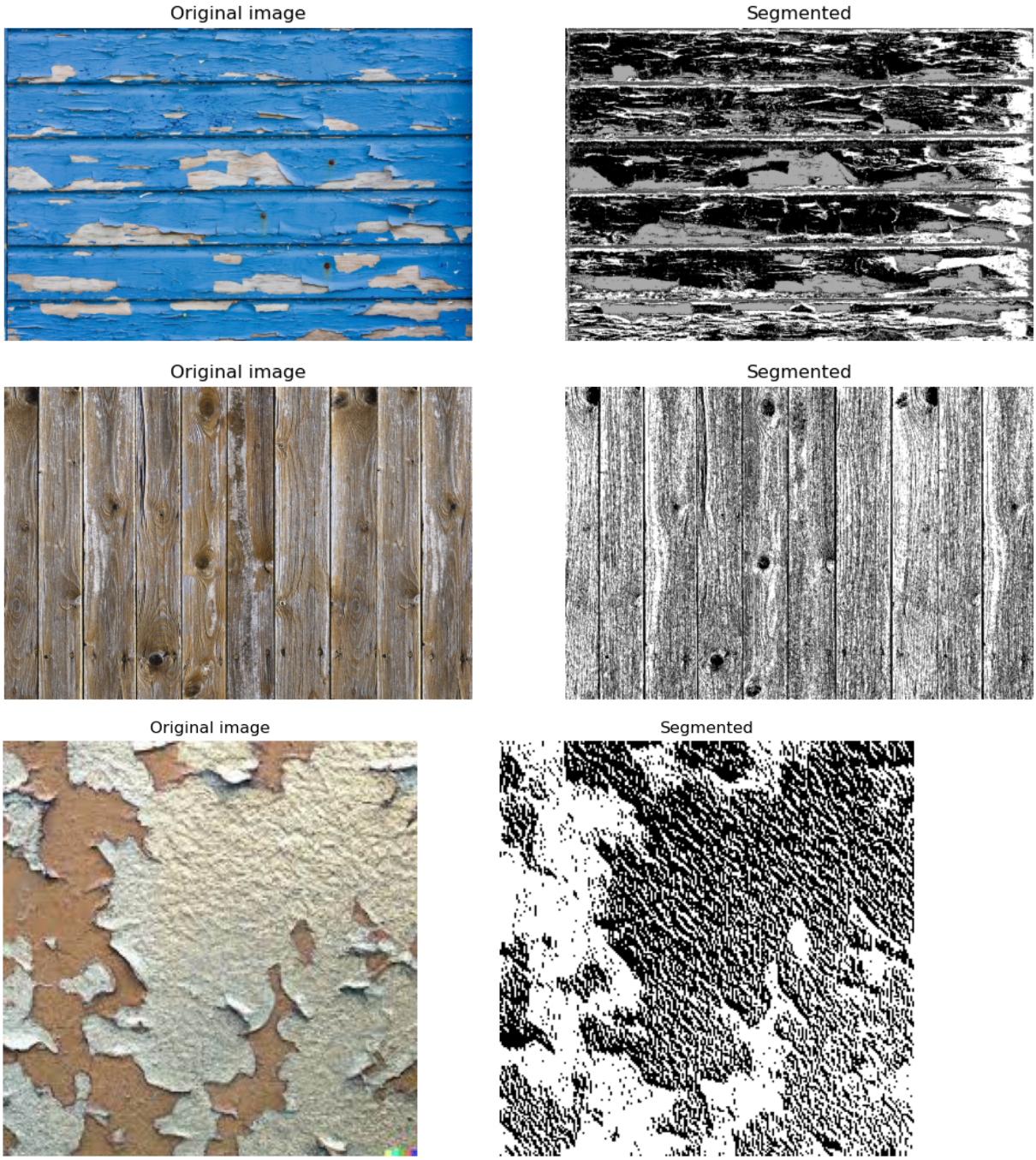

Figure 12: Results of texture-based Segmentation

Furthermore, parameters such as sigma, theta, and window size for Gabor filters were tested with different values, and the results are observable in the following section. As observed, increasing the window size and sigma value of the Gabor filter makes it more sensitive to textural regions and can better detect them. However, the theta parameter, as it determines the angle, should be chosen appropriately based on the different angles present in the original image.

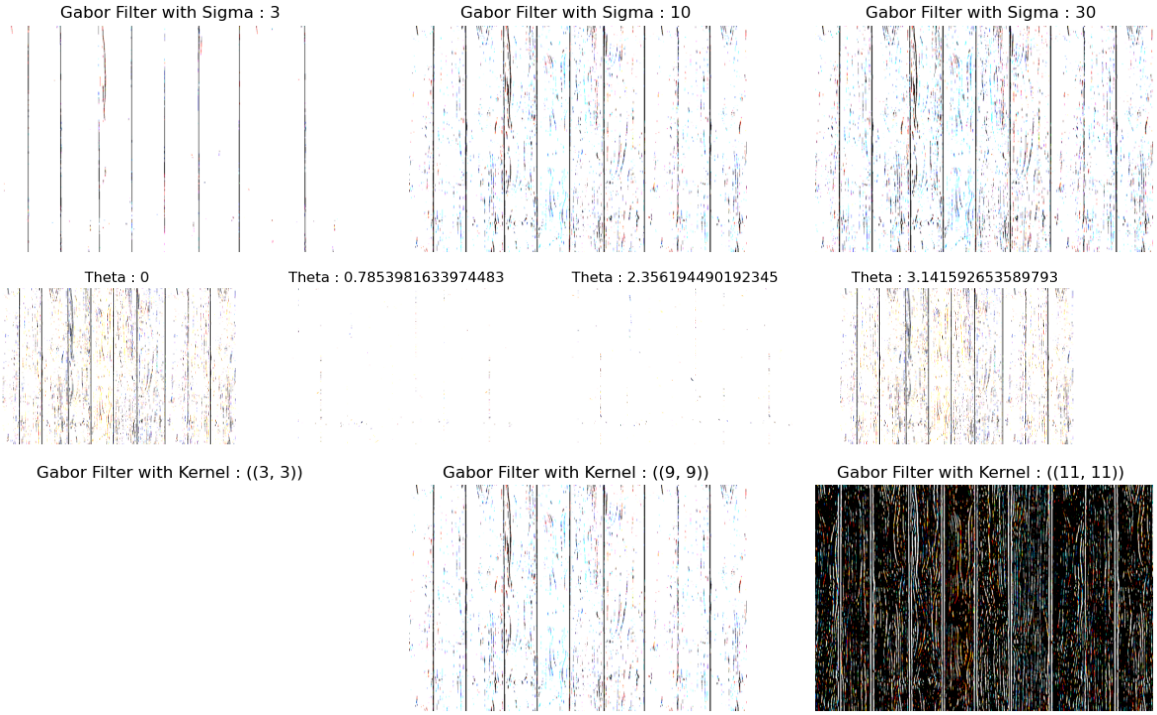

Figure 13: Results of texture-based Segmentation

4 Histogram of Oriented Gradients (HOG)

The Histogram of Oriented Gradients (HOG) algorithm computes image descriptors by first dividing the image into small spatial regions called cells. For each cell, gradients are computed using the Sobel filters. These gradients are then quantized into a set of discrete orientation bins, and the magnitude of each gradient is distributed among these bins based on its orientation. The descriptor for each cell is constructed by concatenating the histogram of gradient orientations over all pixels within the cell. To enhance invariance to illumination and contrast variations, cells are grouped into larger spatial regions called blocks, and the histograms of these blocks are normalized. Finally, the descriptors of all blocks are concatenated to form the final feature vector, which represents the local object appearance and shape within the image.
I implemented the Histogram of Oriented Gradients (HOG) using the hog function from the skimage library, which has three main parameters:

- 1.The number of orientation bins in the histogram
- 2.The number of pixels per cell
- 3.The number of cells per block

```
def HOG(image, orientation, pixels_per_cell, cells_per_block):
    fd, hog_image = hog(image, orientation, pixels_per_cell,
                        cells_per_block, visualize=True, channel_axis=-1)

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), sharex=True, sharey=True)

    ax1.axis('off')
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    ax1.imshow(image, cmap=plt.cm.gray)
    ax1.set_title('Input image')

    # Rescale histogram for better display
    hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

    ax2.axis('off')
    ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
    ax2.set_title('Histogram of Oriented Gradients')
    plt.show()
```

Figure 14: HOG Implementation

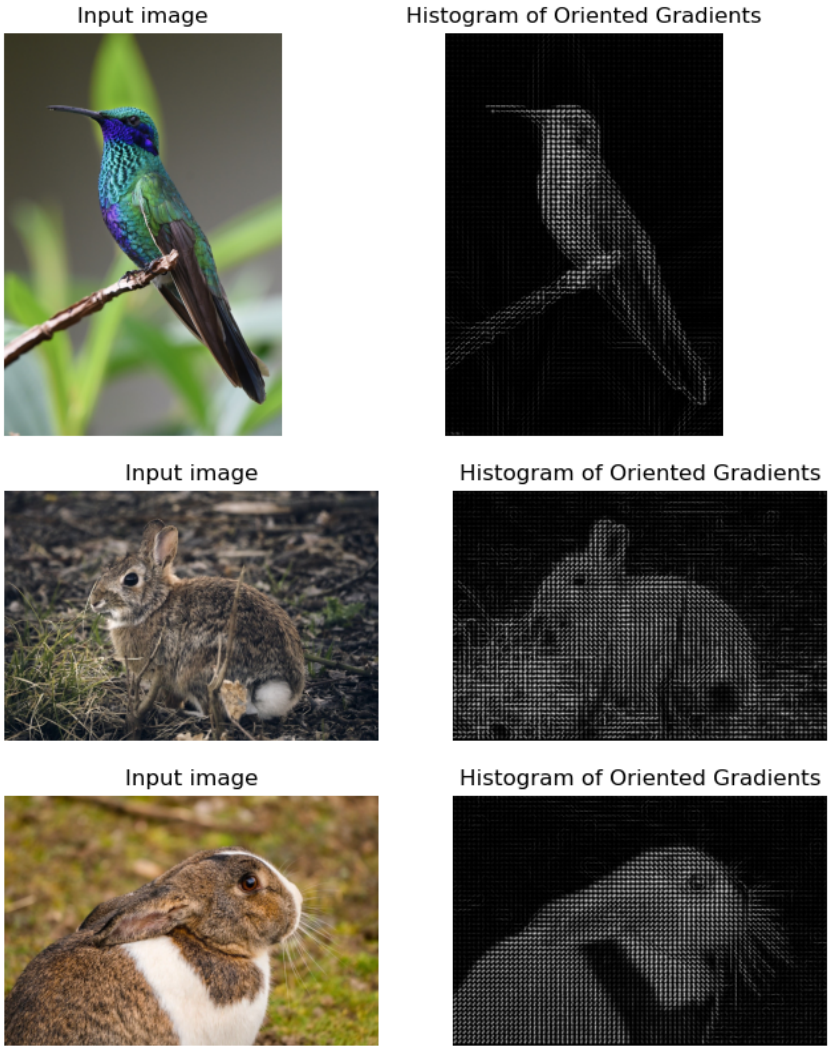

Figure 15: HOG Results with parameters: orientation = 12, pixel per cell = 16, ce1l per block = 1

As it is obvious from the result, higher number of orientation bins result in more detailed images with sharper edges and finer details. On the other side, lower number of orientation bins result in less detailed images with smoother edges and lower resolution. This is because fewer orientations capture less information about the image gradients. However, low orientations are less sensitive to noise in the image.

Figure 16: HOG Results, The Number of Orientation Bins: 2, 12, 100

When the value of pixel per cell is high, each cell in the image covers a larger portion of the image. This can improve the accuracy in detecting features and patterns but may result in loss of finer details. Conversely, when the pixel per cell value is low, cells respond more accurately to the image and capture smaller details. This can enhance the accuracy in detecting fine details but may result in larger output images and sometimes unnecessary resource consumption.

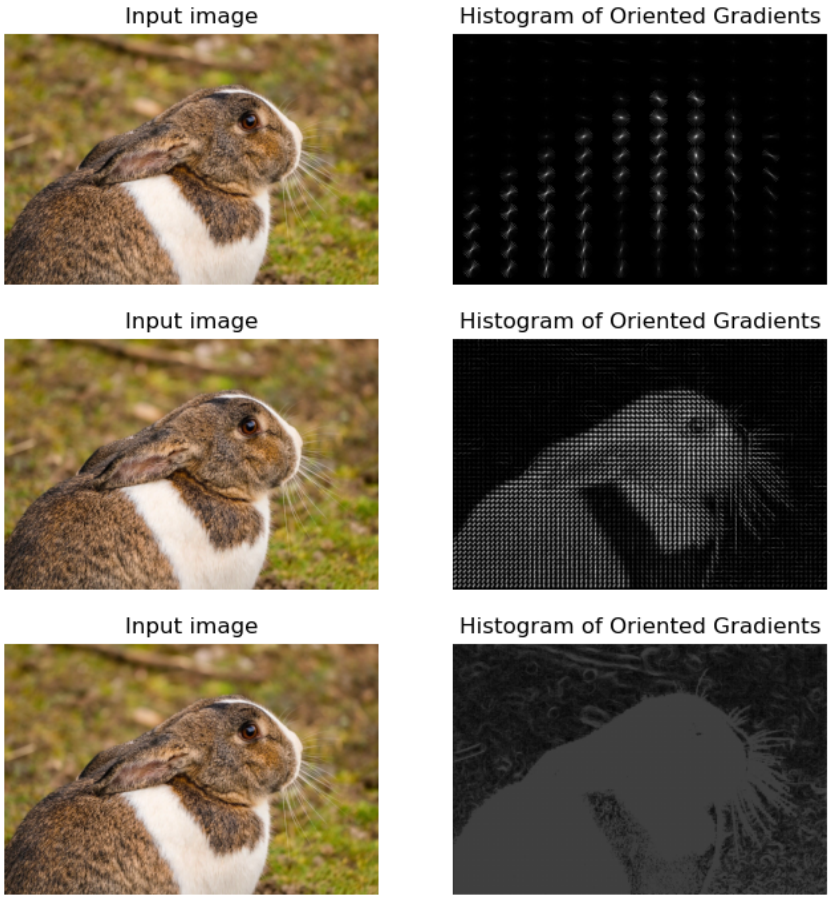

Figure 17: HOG Results, The Number of Oixels per Cell: 4×4, 16×16, 64×64

Accoring to the result representing 1 cell per block and a block size of 32 by 32, When the value of cells per block is higher, it means a larger local area is considered for normalization. This can help in capturing more global patterns and reducing the effect of local variations in the image. However, it may also lead to a loss of finer details within each block. On the other side, when the value of cells per block is lower, it means a smaller local area is considered for normalization. This can lead to better sensitivity to local variations and finer details within each block. However, it may also increase the sensitivity to noise and result in a less stable representation.

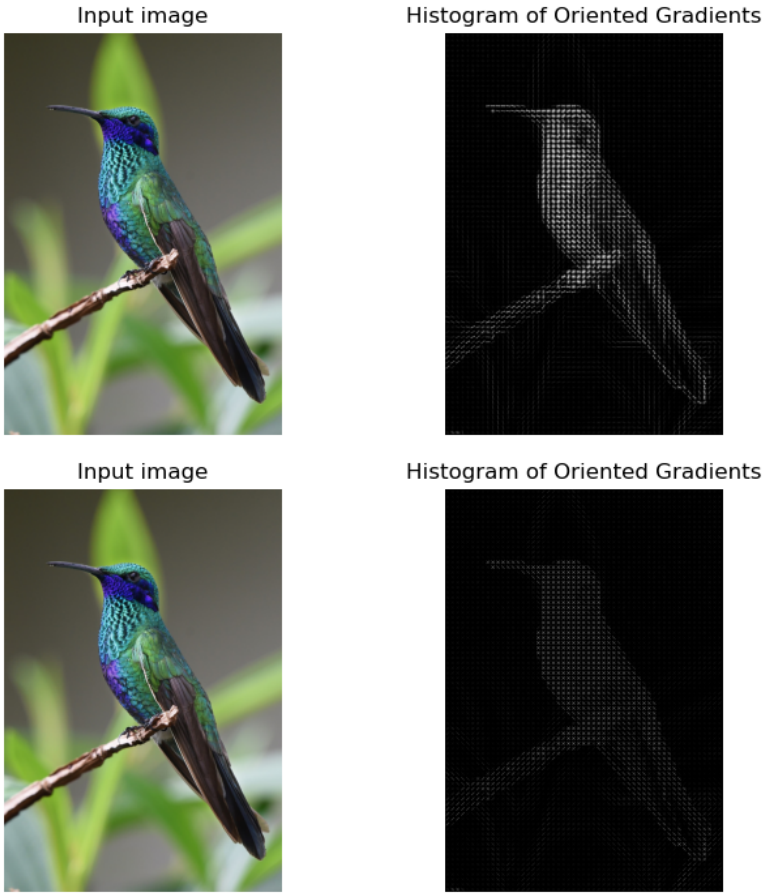

Figure 18: HOG Results, The Number of Pixels per Cell: 1, 32×32

5 Sift Algorithm for Scene Stitching

The **stitch_img** function receives two images and the sift object to perfom image stiching. Initially, I convert the input images to grayscale . Next, it detects keypoints and computes the corresponding descriptors for each image using **sift.detectAndCompute**. I then utilize **cv2.BFMatcher** to perform brute-force matching of descriptors between the two images. After filtering ambiguous matches with a ratio test, I estimate the perspective transformation between the images using **cv2.findHomography**, which leverages the Random Sample Consensus (RANSAC) algorithm. Finally, I warp the second image onto the first using **cv2.warpPerspective**, blending overlapping regions to create a seamless stitched panorama.

```python
def stich_img(img1, img2, sift):
    # find the keypoints and descriptors with SIFT
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    kp1, des1 = sift.detectAndCompute(img1_gray, None)
    kp2, des2 = sift.detectAndCompute(img2_gray,None)

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des2,des1, k=2)

    # Apply ratio test
    good = []
    for m in matches:
        if m[0].distance < 0.5*m[1].distance:
            good.append(m)
            matches = np.asarray(good)

    if len(matches[:,0]) >= 4:
        src = np.float32([ kp2[m.queryIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)
        dst = np.float32([ kp1[m.trainIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)
        H, masked = cv2.findHomography(src, dst, cv2.RANSAC, 5.0)
    else:
        raise AssertionError("Can not find enough keypoints.")
    dst = cv2.warpPerspective(img2, H,(img1.shape[1] + img2.shape[1], img1.shape[0]))
    dst[0:img1.shape[0], 0:img1.shape[1]][img1 != 0] = img1[img1 != 0]
    return dst
```

Figure 19: Scene Stitching Implementation

Therefore, utilizing the function **stitch_img**, initially, the right image is stitched with the center image. Subsequently, the output of these two images is sent back to the **stitch_img** function along with the left image to produce the final panoramic image.

```python
#left
img1 = cv2.imread("D:/Uni/BCV/Exercise_2/images/5/sl.jpg")

#center
img2 = cv2.imread("D:/Uni/BCV/Exercise_2/images/5/sm.jpg")

#right
img4 = cv2.imread("D:/Uni/BCV/Exercise_2/images/5/sr.jpg")

sift = cv2.SIFT_create(nfeatures = 0, nOctaveLayers = 40, contrastThreshold = 0.04, edgeThreshold = 200, sigma = 5,
                       enable_precise_upscale = False)

img3 = stich_img(img1, img2, sift)
final_image = stich_img(img3, img4, sift)
final_image = final_image[0:final_image.shape[0], 0:1200]
plt_imshow("Panaroma Image", final_image)
```

Figure 20: Scene Stitching Implementation

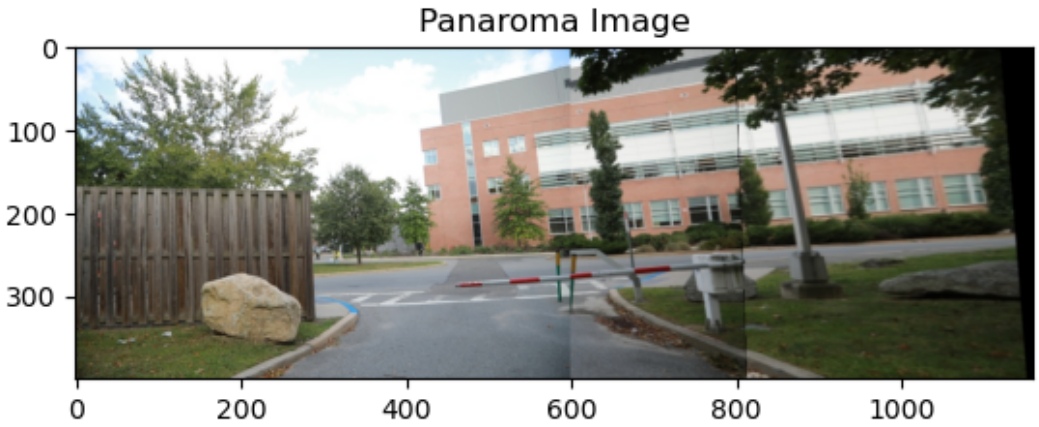

Figure 21: Result of Scene Stitching

One parameter in sift Algorithm is sigma which is the gaussian scale parameter for creating the scale-space pyramid. According to the result, a lower sigma value might be more suitable if the images contain finer details or if preserving sharp edges is crucial. Lower sigma values allow for finer-scale features to be detected and can result in more precise keypoint localization. On the other side, a high sigma value can lead to poor stitching results due to oversmoothing of the input images.

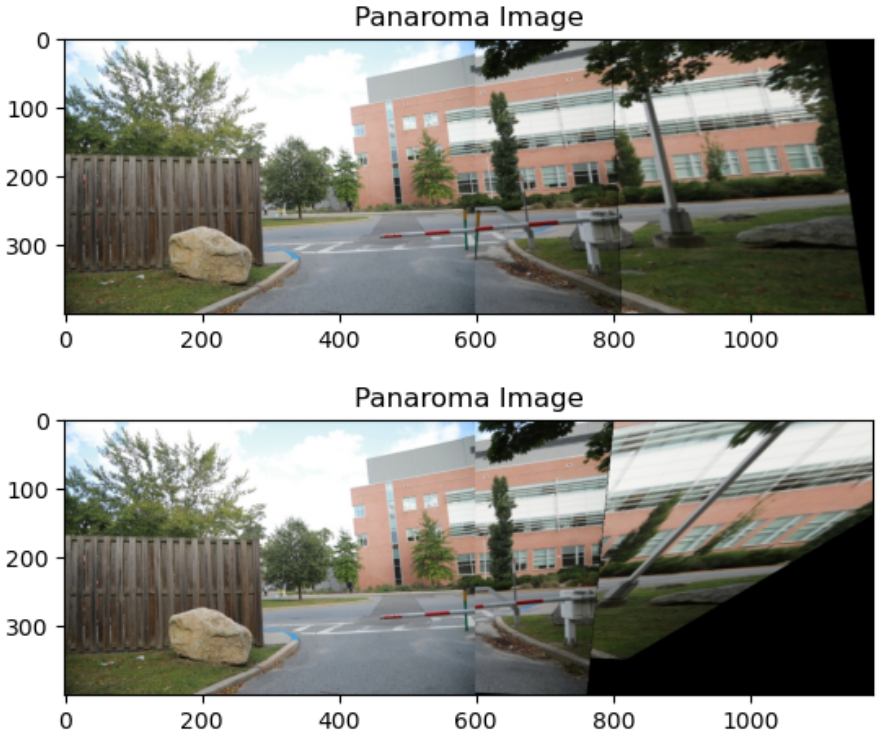

Figure 22: Sigma = 1, Sigma = 25

Another parameter is contrastThreshold which determines the minimum contrast required for a keypoint to be detected. According to the result, lower values of contrastThreshold will result in more keypoints being detected, including those with lower contrast, but higher values of contrastThreshold will lead to fewer keypoints being detected, as it requires keypoints to have higher contrast to be considered and lead to poor stitching results.

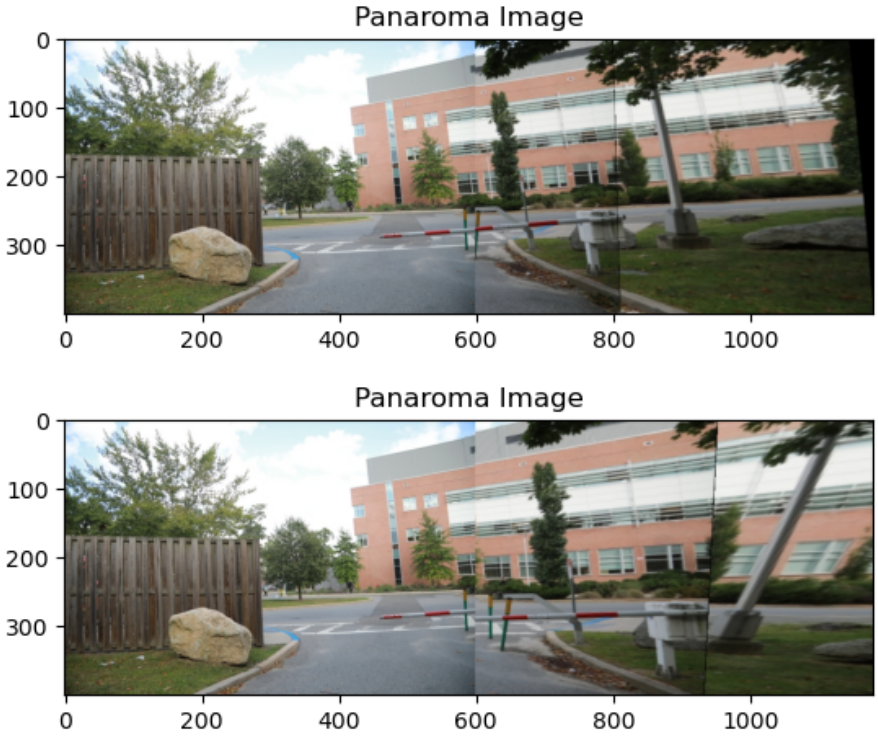

Figure 23: contrastThreshold = 0.001, contrastThreshold = 0.28