İhsan Doğramaci Bilkent University

Introduction to Computer Vision
CS 484

Assignment: Homework #3

Author:

Yusuf Dalva
Bilkent ID: 21602867

April 16, 2020

Table of Contents

Loading Images

In order to organize the dataset given for the assignment the images provided are organized as a dataset. The implementation of this Dataset class is given as follows:

```
class Dataset:
    ## The images are assumed as in the local directory,
    ## Change path names accordingly
    def __init__(self):
        ## Import images for part 1
        edge_image_names = ['edge1.png','edge2.png', 'edge3.png']
        self.edge_images = []
        for image_name in edge_image_names:
            image = mpimg.imread(image_name)
            self.edge_images.append(rgb2gray(image))
            print("Imported " + image_name + ": Original Dimesions " + str(image.shape))
        ## Import images for part 2
        hough_image_names = ['hough.png', 'hough2.png', 'hough3.png']
        self.hough_images = []
        for image_name in hough_image_names:
            image = mpimg.imread(image_name)
            self.hough_images.append(rgb2gray(image))
            print("Imported " + image_name + ": Original Dimensions " + str(image.shape))
        ## Import hough_result
        result_image_name = 'hough_reS.png'
        image = mpimg.imread(result_image_name)
        self.hough_result = (rgb2gray(image))
        print("Imported " + result_image_name + ": Original Dimensions " + str(image.shape))
        print("INFO: Dataset created")
    def get_edge_images(self):
        return self.edge_images
    def get_hough_images(self):
        return self.hough_images
    def get_hough_result(self):
        return self.hough_result
dataset = Dataset()
```

In this implementation, each of the images are stored in attributes that are mapped to their usages. The image samples that will be used for edge detection are under an attribute **edge_images** and for Hough Transform, the name of the attribute is **hough_images**. Also to access the image that is provided to test the Hough Transform algorithm, which contains the result of applying that algorithm, is stored as **hough_result**. Note that it is assumed that the images are in the same directory with the notebook file, to operate please change the locations of image accordingly.

1 Edge Detection

In this part of the assignment, the Edge Detection task will be attempted with different appraches. The approaches used are

- Applying Sobel Operators
- Applying Prewitt Operators
- Canny Edge Detector

Both the mechanics behind these algorithm and a comparison regarding these methods will be provided in this report.

1.1 Edge Detection with Sobel and Prewitt Operators

For this part of the assignment, Sobel and Prewitt operators will be used for edge detection task. The following two subsections are dedicated to applying these two operators.

As mentioned in the definition of edges, the edge refers to significant change in pixel values in an image. From the perspective of human eye, this significant change is most likely to be found with the derivation concept. As the image is a matrix of pixels where the pixel values are ordered in a discrete manner, the discrete derivative concepts is being used for derivation. The equation for derivation is as follows:

$$f'(x) = f(x+1) - f(x) \tag{1}$$

This equation follows from the fact that the distance between two pixel values is 1, in one dimensional space. For the 2 dimensional case, which is valid for image domain, the equation for the first derivative is as follows:

$$f'(x,y) = [\frac{\partial f(x,y)}{\partial x} \frac{\partial f(x,y)}{\partial y}] \tag{2}$$

The partial derivative mentioned is also calculated in a similar manner as it is in the 1 dimensional version. Following these two equations, the masks regarding the Sobel and Prewitt operators are formed.

1.1.1 Applying Sobel Operators

The Sobel operator follows the approach of discrete differentiation. After applying vertical and horizontal Sobel masks to the image, the image is going to be the gradient vector for that image, the direction of the vector is horizontal or vertical depending on which operation is applied. As it is illustrated in the masks for Sobel operator, these filters have more focus on the pixels that are closer to the center of the window. The Sobel filter masks are given as follows:

$$Sobel_{horizontal} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$Sobel_{vertical} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 2 & 1 \end{bmatrix}$$

Regarding these filters, the following implementation is proposed for applying sobel filters:

```
def apply_sobel(image):
    sobel_horizontal = [
        [-1,-2,-1],
        [0, 0, 0],
        [1, 2, 1]
    ]
    sobel_vertical = [
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1]
    ]
    filter_dims = (3,3)
    vertical_edges = ndimage.filters.convolve(image, sobel_vertical)
    horizontal_edges = ndimage.filters.convolve(image, sobel_horizontal)
    sobel_res = np.sqrt(np.square(horizontal_edges) + np.square(vertical_edges))
    return (sobel_res / sobel_res.max() * 255,
           np.arctan2(vertical_edges, horizontal_edges))
```

To perform the convolution operation efficiently, the convolve method is being used in this implementation. The first version of the implementation used a double for loop approach but after trying out this method, a significant increase in performance is observed. The results of applying Sobel operator is given for the images *edge1.png*, *edge2.png* and *edge3.png*.

- Results for *edge1.png*:

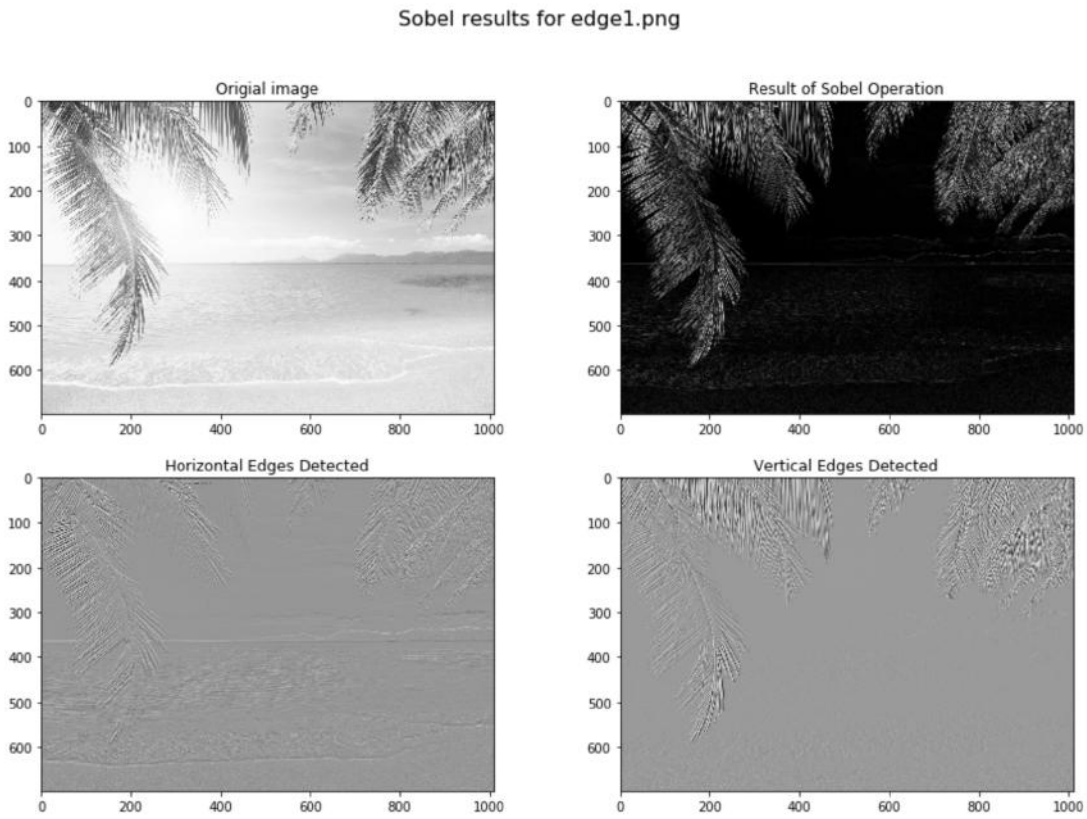

Figure 1: Results of applying Sobel Operator on image edge1.png

- Results for *edge2.png*:

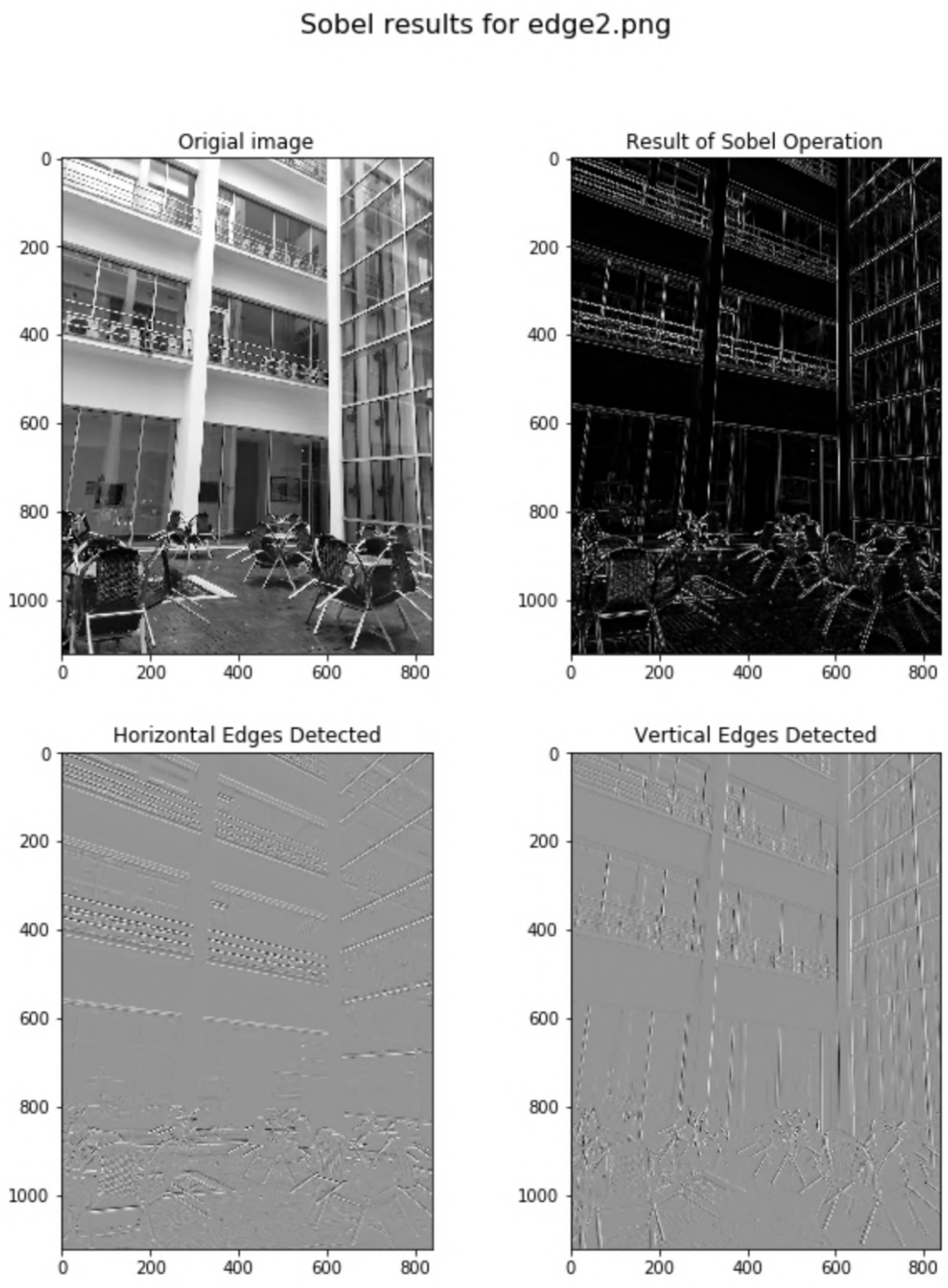

Figure 2: Results of applying Sobel Operator on image edge2.png

- Results for *edge3.png*:

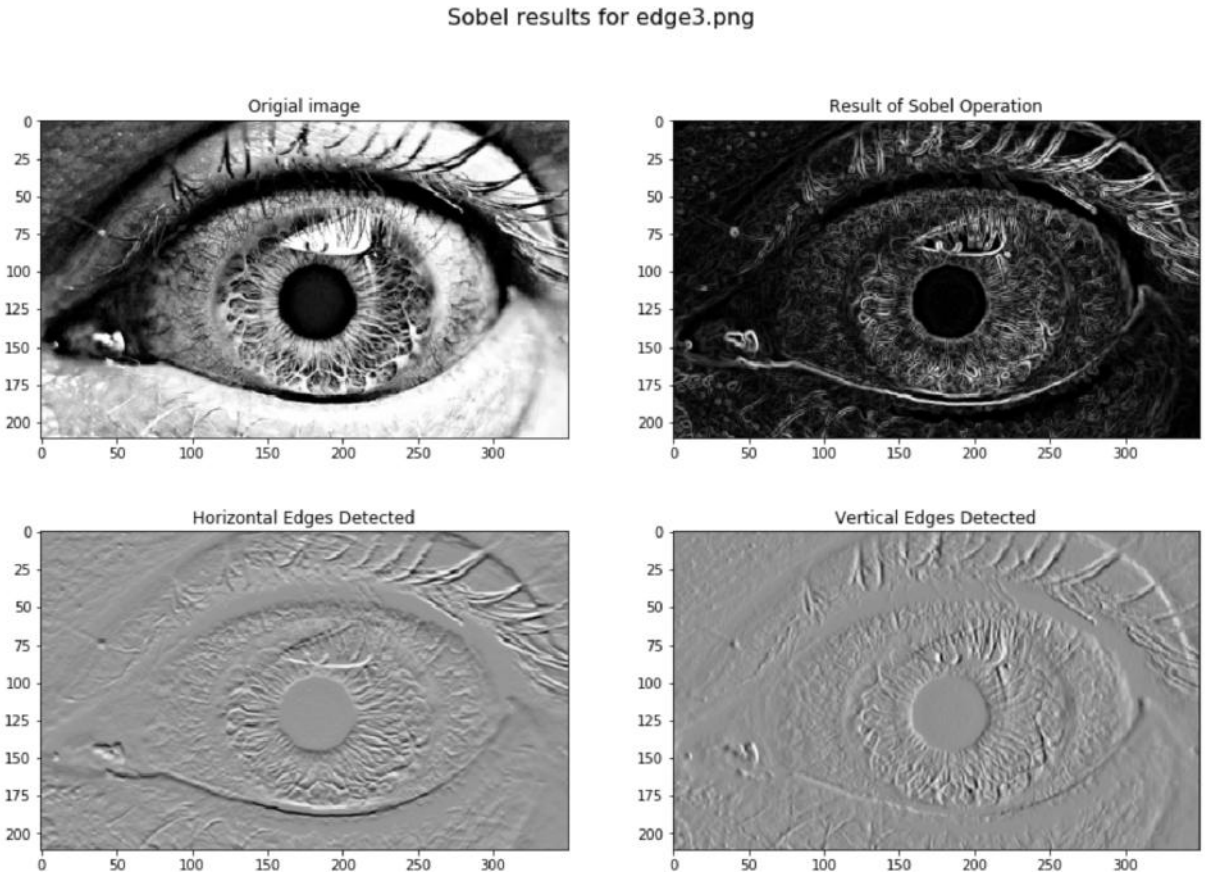

Figure 3: Results of applying Sobel Operator on image edge3.png

1.1.2 Applying Prewitt Operators

Also based on differentiation, the Prewitt operator is also a 3x3 mask that is being applied to the image. Different than the Sobel filter, this mask does not reflect any difference in terms of weight to the pixel that it is being applied to. As this is an operator that is based on discrete approximation, the result of applying this operator gives an approximation of the gradient vector. The masks that are used for prewitt operation is as follows:

$$Prewitt_{horizontal} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$Prewitt_{vertical} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 2 & 1 \end{bmatrix}$$

To apply this filter to an image, the following implementation is proposed, just like in the Sobel operator, the convolve method is being used to perform the convolution operation. Here is the implementation:

```
def apply_prewitt(image):
    prewitt_horizontal = [
        [-1,-1,-1],
        [0, 0, 0],
        [1, 1, 1]
    ]
    prewitt_vertical = [
        [-1, 0, 1],
        [-1, 0, 1],
        [-1, 0, 1]
```

```
]
vertical_edges = ndimage.filters.convolve(image, prewitt_vertical)
horizontal_edges = ndimage.filters.convolve(image, prewitt_horizontal)
prewitt_res = np.sqrt(np.square(horizontal_edges.astype(int)) +
    np.square(vertical_edges.astype(int)))
prewitt_res = prewitt_res / prewitt_res.max() * 255
return (prewitt_res,
       np.arctan2(vertical_edges, horizontal_edges))
```

Following the procedure which is applied for Sobel operator the results obtained after applying the Prewitt operator is as follows:

- Results for *edge1.png*:

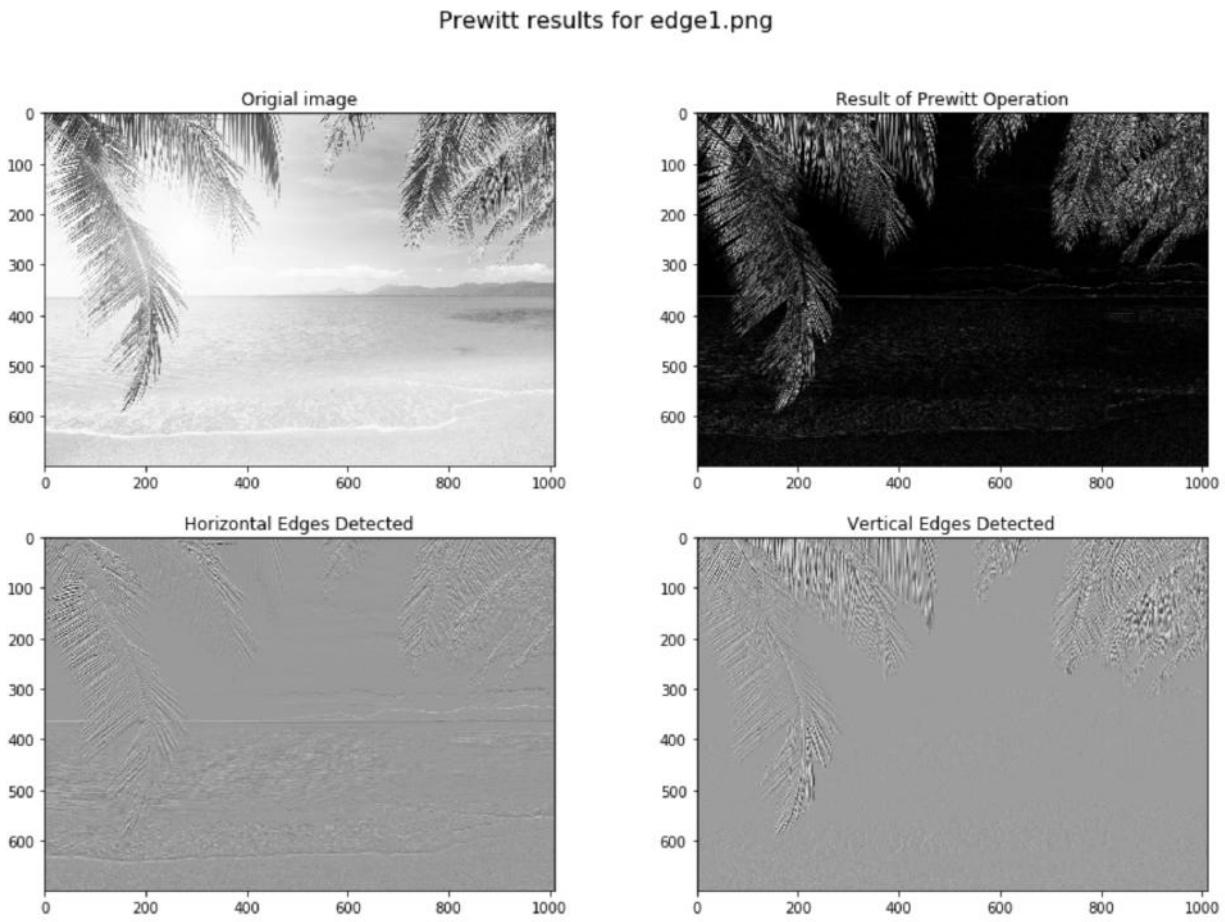

Figure 4: Results of applying Prewitt Operator on image edge1.png

- Results for *edge2.png*:

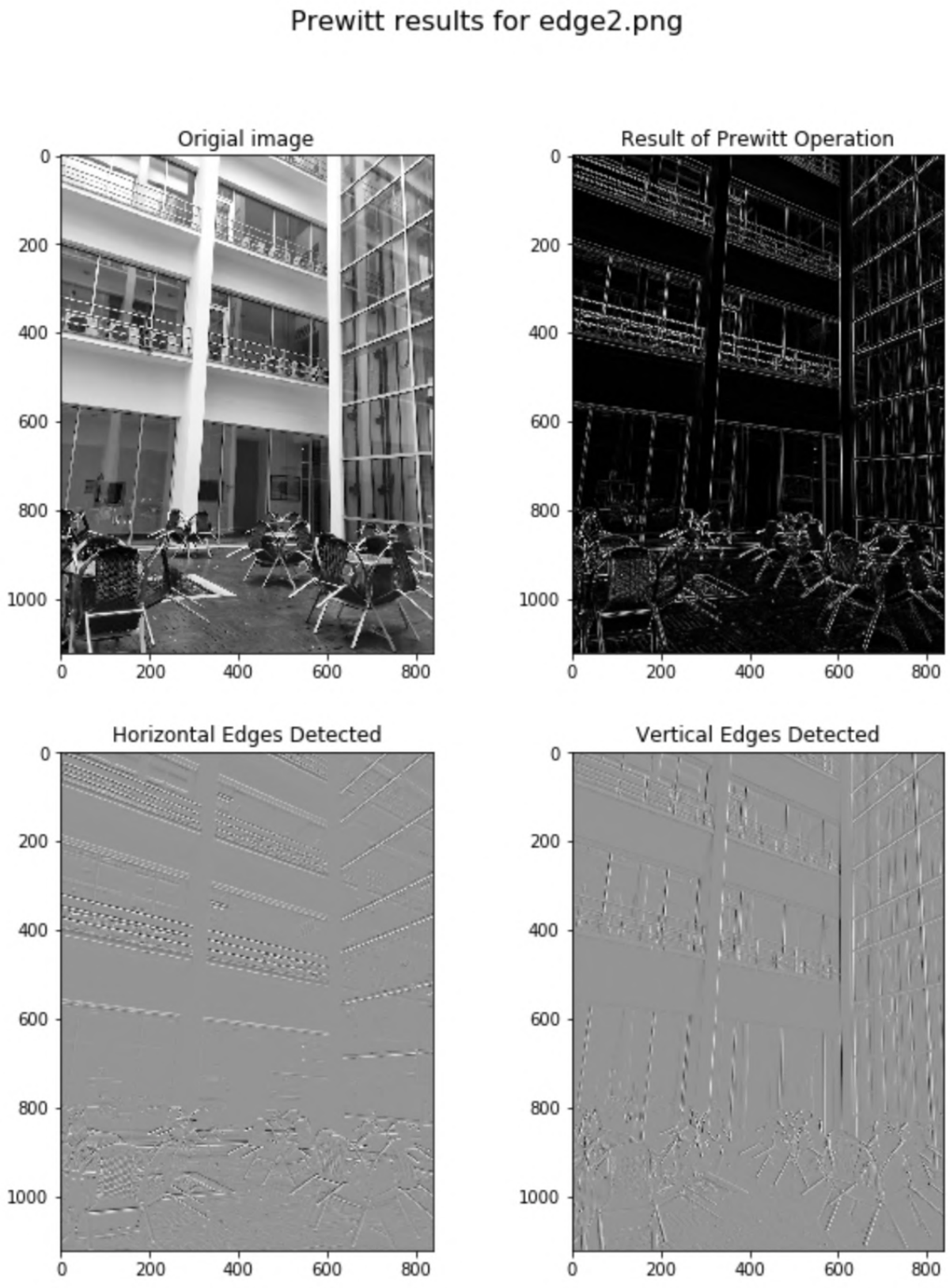

Figure 5: Results of applying Prewitt Operator on image edge2.png

- Results for *edge3.png*:

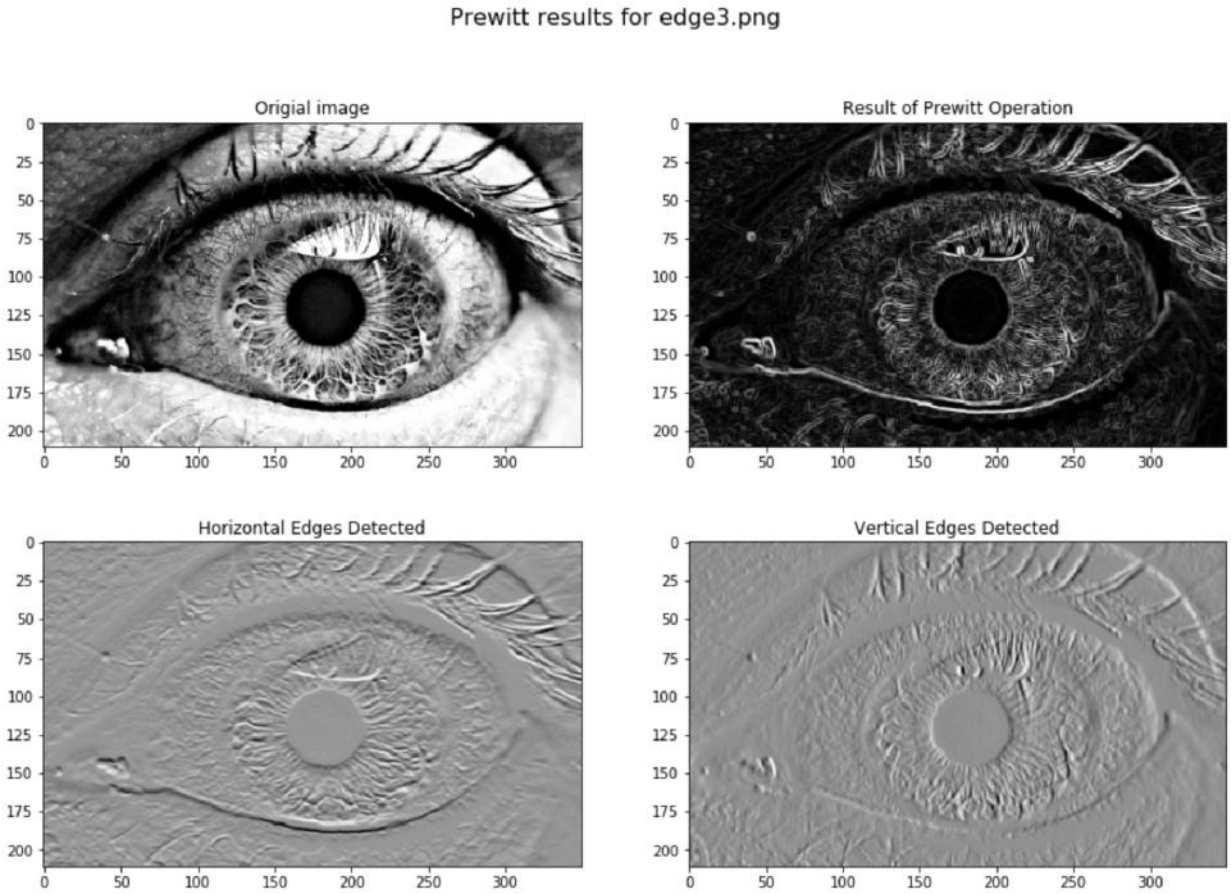

Figure 6: Results of applying Prewitt Operator on image edge3.png

1.1.3 Comparing Sobel and Prewitt Operators

As it is discussed when explaining the masks applied for Sobel and Prewitt operators, they have difference only in terms of the weights given to the closest pixels to the center. As it can be seen from the filters, this difference is reflected in horizontal direction for horizontal filter and in vertical direction in vertical filter. This is reflected to the computation in this way since the gradient is being calculated, which is perpendicular to the variable (pixel values). In order to show this, the horizontal filters for Sobel and Prewitt operations are given as follows:

$$Sobel_{horizontal} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$Prewitt_{horizontal} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

With the resolution that the images are displayed, this difference is not that significant. However these operators cannot be considered as identical as explained in the following discussion

By assigning a higher weights to the pixels that are closer to the center of the mask, the edges that are thinner can be detected rather than the convolution value that treats the pixels around equally, the edge mapping involves more edges for the Sobel operator case ideally, which is not clearly observed for this case[4][1]. However, both of these images are similar in the sense that they are trying to find the first derivative of the matrix, which will approximate to the locations which pixel intensities change significantly(local minima).

1.2 Edge Detection with Canny Edge Detector

For this question, as no limitation is mentioned, the Canny Edge Detector method included in skimage library is used. Using this method the parameters sigma, high threshold and low threshold can all be changed. To explain that these parameters refer to, a brief discussion about the Canny Edge Detector is provided.

1.2.1 Canny Edge Detector Algorithm

Different than the previous methods offered in this report (Applying Sobel and Prewitt operators), Canny Edge Detector follows a multi stage approach. The stages of the algorithm is as follows:

- Noise Reduction
- Gradient Calculation
- Non-Maximum Suppression
- Double Threshold
- Edge Tracking by Hysteresis

1.2.1.1 Noise Reduction As discussed in the lecture, in order to eliminate noise blurring can be applied. In order to reduce the noise one of salt-and-pepper filter, Gaussian filter and Impulse filter will be applied. Considering the results that these filters in order to control the amount of the noise removed (with parameter σ) and the results that the filter results in, makes the Gaussian Filter be the most suitable for this task.

Here the parameter $\sigma(sigma)$ shows the standard deviation of the values in the filter applied. The equation for the Gaussian filter is as follows[2]:

$$H_{ij} = \frac{1}{2\pi\sigma^2} exp(\frac{(i-(k+1)^2)^2 + (j-(k+1)^2)^2}{2\sigma^2}), 1 < i, j < (2k+1) \tag{3}$$

1.2.1.2 Gradient Calculation After the noise removal stage, the gradients corresponding the edges will be found. As explained in the part related with Sobel and Prewitt operations, calculating the gradients is similar to finding edges due to nature of differentiation. Since the edges represents the places where the pixel intensity changes significantly, the differentiation operation is analogous to finding the edges. In the Canny Edge Detection algorithm referred for explanation Sobel Operators are used. After finding horizontal and vertical gradients, the gradient of the image corresponding to the edges will be found in the following way:

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{4}$$

$$\theta_G = \tan^{-1}(\frac{G_y}{G_x}) \tag{5}$$

Here the symbol G_y corresponds to the vertical gradient and G_x corresponds to the horizontal gradient.

1.2.1.3 Non-Maximum Suppression In the ideal scenario, the image where edges are detected should have thin edges. In order to achieve this, the method non-maximum suppression has been applied as the third step of the algorithm. According to this principle, the algorithm finds the points having the maximum value in edge direction in gradient intensity matrix firstly. Then in this direction, only the pixel that has the most intensity is kept, if pixels have equal intensity then all of the pixels are kept. More details about this method can be found in the bibliography[2].

1.2.1.4 Double Threshold In the idea of double threshold, the image after non-maximum suppression has three kinds of pixel values. These pixel values (classes) are:

- Strong: Values above the high threshold (T_{high})
- Non-relevant: Values below the low threshold (T_{low})
- Weak: Values in the scale (T_{low}, T_{high}) - between the threshold values

Here the pixels that are marked as strong are the ones that are very likely to contribute to the edges of the image. The weak pixels here are not likely to contribute to edges in the same likelihood with strong pixels but they can contribute to an edge. Here the non-relevant pixels are the ones that it is certain that they will not contribute to an edge.

1.2.1.5 Edge Tracking by Hysteresis As the final stage of the algorithm, the hysteresis procedure is applied. This procedure will transform weak pixels to strong pixels conditionally. The input to this part is the mapping done by double thresholding. The condition of transforming weak pixels into strong ones are as follows: If a weak pixel has a neighborhood relationship with a strong pixel then the weak pixel is transformed into a strong pixel. If not, the pixel will be considered irrelevant to the edges in the image. Here the neighborhood window is considered as a 3x3 window[2].

1.2.2 Trying out Different Parameters

For a logical ordering in the report, first the parameters tried for optimal parameters are shown. The results will be given in the order

- edge1.png
- edge2.png
- edge3.png

1.2.2.1 edge3.png For this image, 4 parameter sets for Canny Edge Detector has been attempted. The parameters are as follows:

σ	T_{low}	T_{high}
1	0.05	0.15
2	0.07	0.15
2	0.03	0.2
3	0.05	0.2

Here the floating points are corresponding to grayscale pixels in the range [0,1]. The results obtained after plotting with these hyperparameters are as follows:

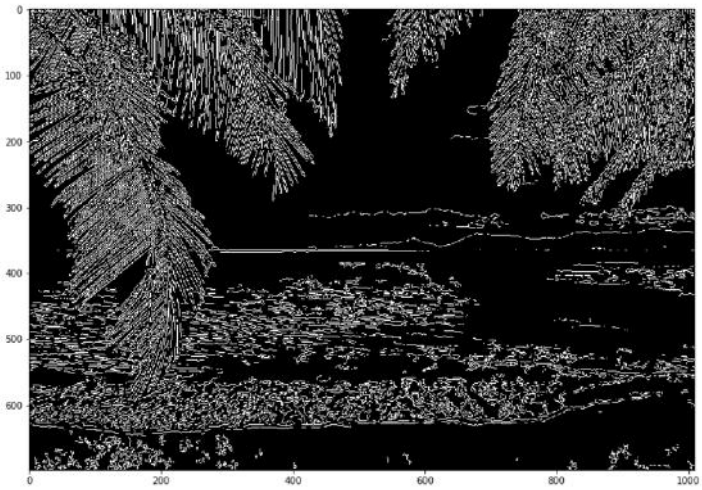

Figure 7: Trying Canny Edge Detector on edge1.png with $\sigma = 1$, $T_{low} = 0.05$ and $T_{high} = 0.15$

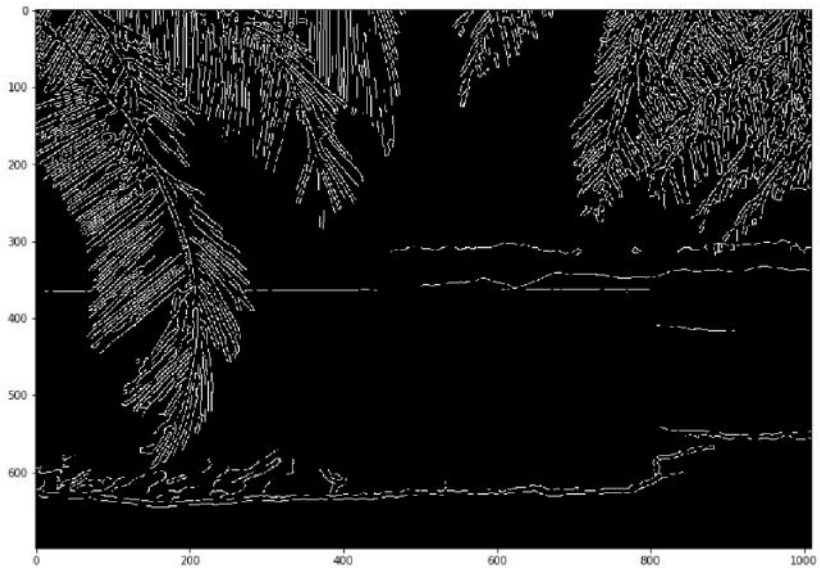

Figure 8: Trying Canny Edge Detector on edge1.png with $\sigma = 2$, $T_{low} = 0.07$ and $T_{high} = 0.15$

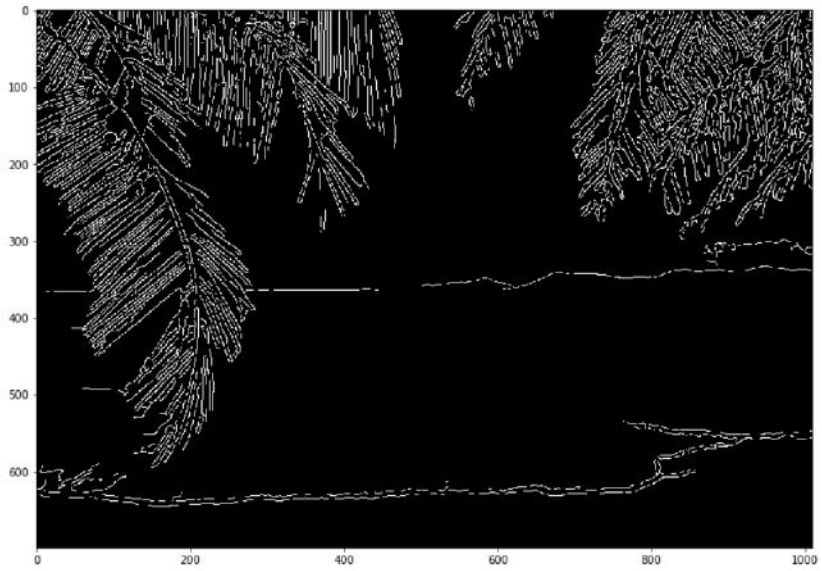

Figure 9: Trying Canny Edge Detector on edge1.png with $\sigma = 2$, $T_{low} = 0.03$ and $T_{high} = 0.2$

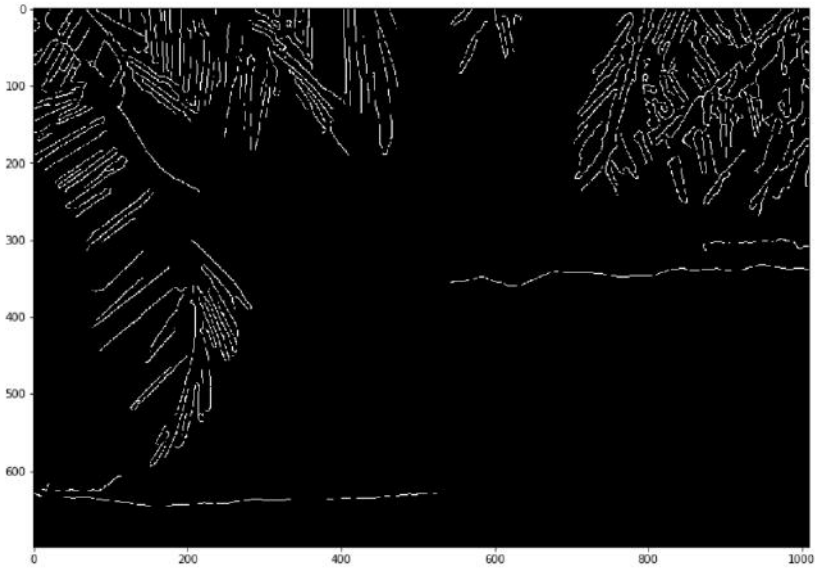

Figure 10: Trying Canny Edge Detector on edge1.png with $\sigma = 3$, $T_{low} = 0.05$ and $T_{high} = 0.2$

1.2.2.2 edge2.png For this image, 4 parameter sets for Canny Edge Detector has been attempted. The parameters are as follows:

σ	T_{low}	T_{high}
1	0.05	0.15
2	0.07	0.09
2	0.09	0.15
3	0.09	0.11

By applying these the following results are obtained:

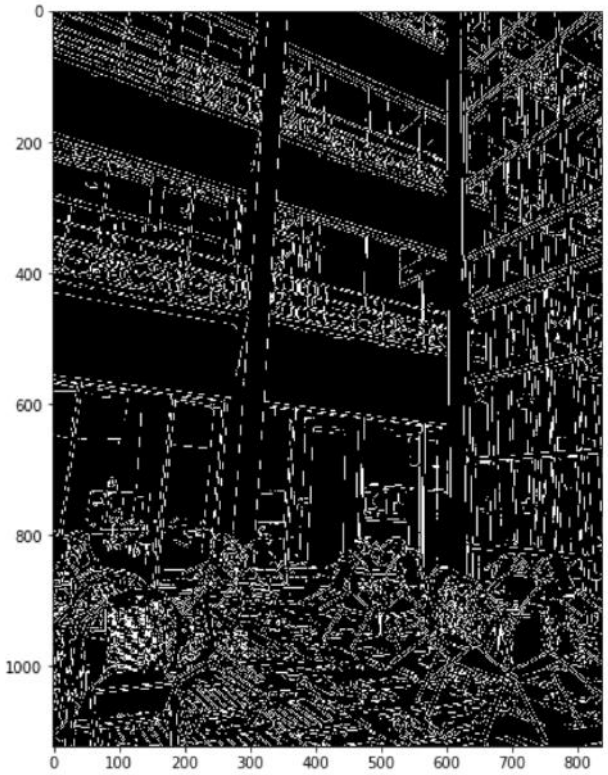

Figure 11: Trying Canny Edge Detector on edge2.png with $\sigma = 1$, $T_{low} = 0.05$, $T_{high} = 0.15$

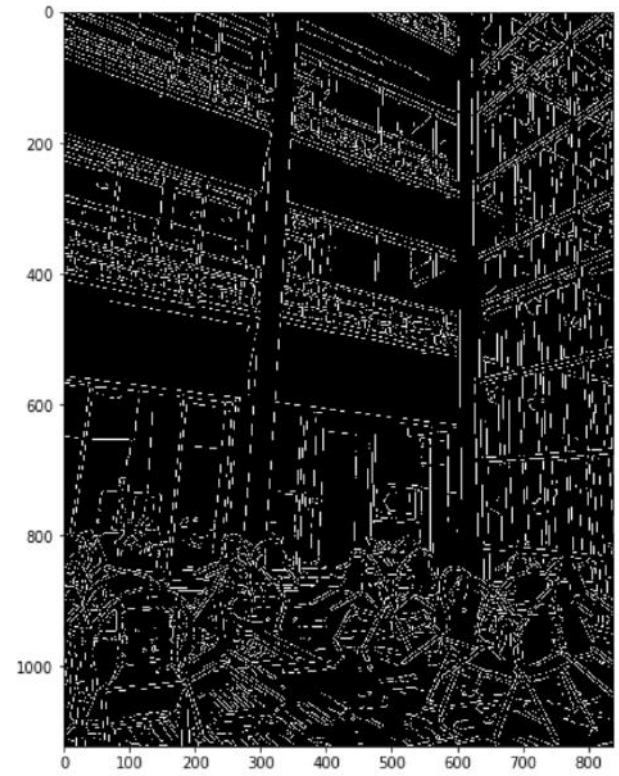

Figure 12: Trying Canny Edge Detector on edge2.png with $\sigma = 2$, $T_{low} = 0.07$, $T_{high} = 0.09$

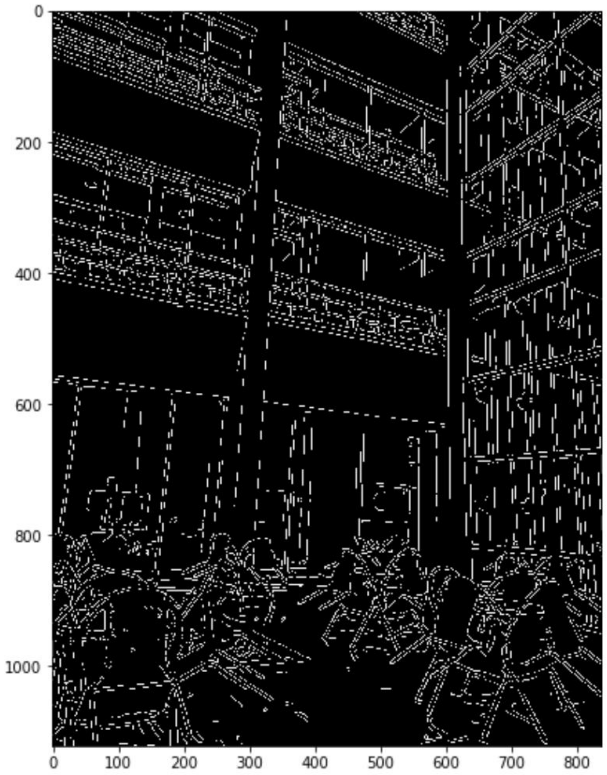

Figure 13: Trying Canny Edge Detector on edge2.png with $\sigma = 2$, $T_{low} = 0.09$, $T_{high} = 0.15$

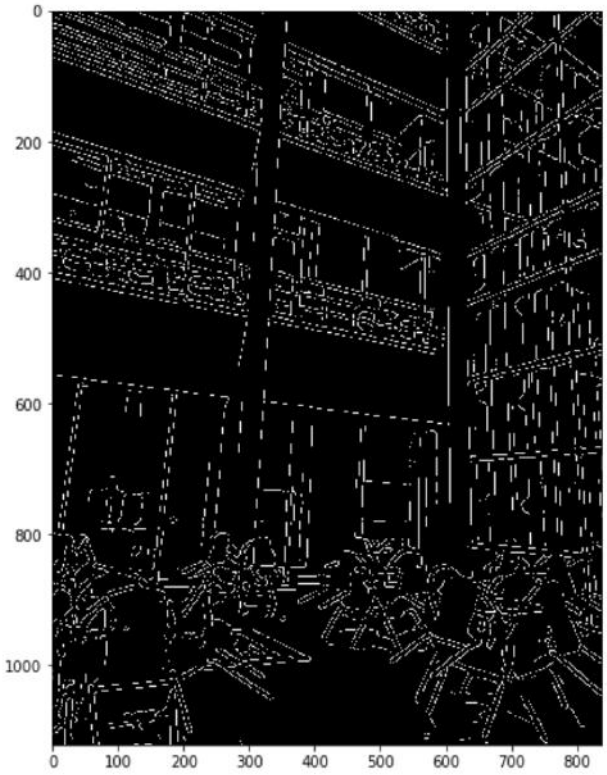

Figure 14: Trying Canny Edge Detector on edge2.png with $\sigma = 3$, $T_{low} = 0.09$, $T_{high} = 0.11$

1.2.2.3 edge3.png For this image, 4 parameter sets for Canny Edge Detector has been attempted. The parameters are as follows:

σ	T_{low}	T_{high}
2	0.05	0.15
3	0.05	0.15
3	0.05	0.2
4	0.03	0.2

The results after applying these hyper-parameters are as follows:

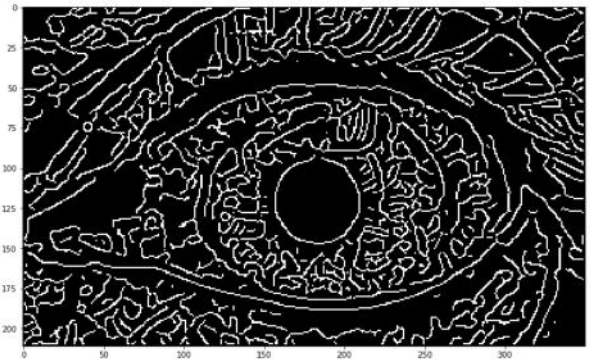

Figure 15: Trying Canny Edge Detector on edge3.png with $\sigma = 2$, $T_{low} = 0.05$, $T_{high} = 0.15$

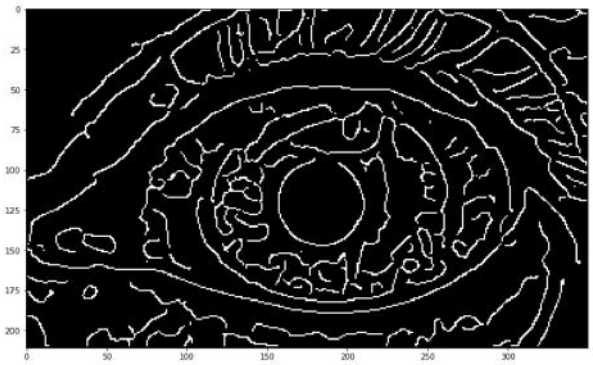

Figure 16: Trying Canny Edge Detector on edge3.png with $\sigma = 3$, $T_{low} = 0.05$, $T_{high} = 0.15$

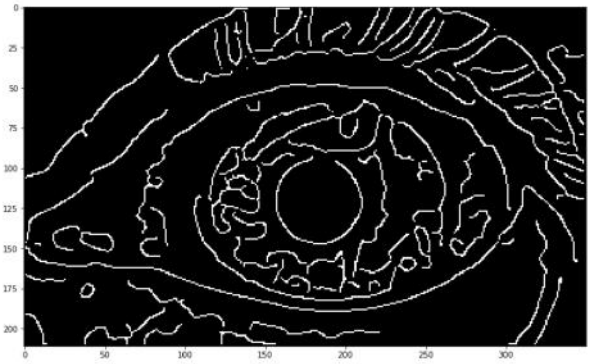

Figure 17: Trying Canny Edge Detector on edge3.png with $\sigma = 3$, $T_{low} = 0.05$, $T_{high} = 0.2$

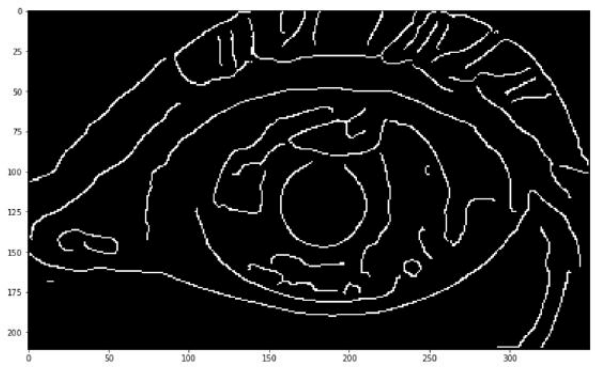

Figure 18: Trying Canny Edge Detector on edge3.png with $\sigma = 4$, $T_{low} = 0.03$, $T_{high} = 0.2$

1.2.3 Optimal Parameters

After trying out different alternatives for the images given for edge detection task, there are some facts that are realised. This section is dedicated on showing the best results and explaining the reasoning behind these results.

edge1.png: For the first image, the first fact realised was the presence of the small amount of noise. This presence of the noise is clearly identified when the σ value changed from 1 to 2. However, since the noise is small, when the σ value incremented to 3, most of the edge details are lost and detection became poor in terms of performance. The second thing observed is about the threshold values. Between the second and third attempts for this image, the small amount of noise left after applying Gaussian Filter is lost. The reason behind that is the increased value of T_{high}. By this way the interval that the weak pixels can be present is widened, where it is narrowed for the strong pixels. As the unwanted edges are separate from the main edges, this behavior was expected and done intentionally. The optimal hyper-parametrs for this image are given as follows:

- σ: 2
- T_{low}: 0.03
- T_{high}: 0.2

The best result for edge detection is as follows:

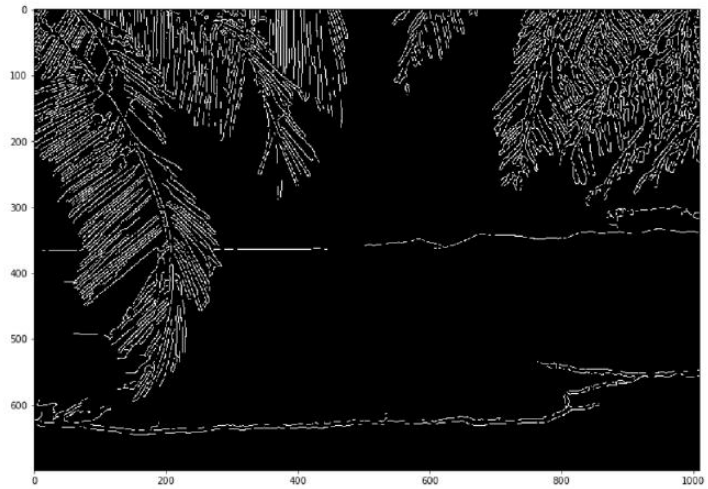

Figure 19: Best Edge Detection result obtained for edge1.png

For this image the disconnected links can easily be linked.

edge2.png: For this image, the noise was a crucial factor but it is not the only reasoning behind discussion. After attempting to detect the image with $\sigma = 1$ on the floor side, the presence of the noise is realised.

Then the σ value increased to 2 and the noise was eliminated partially. To reduce the remaining unwanted pixels the threshold window shifted from (0.07, 0.09) to (0.09, 0.11). By this was the fact identified was that the noisy parts had lower pixel values. As a final attempt the σ value increased to 4 but then the links between most of the edges was lost significantly. The best attempt is selected as the third one here. There are disconnections in the image, but considering the elimination of noise and losing details this was considered as the optimal solution. The corresponding hyper-parameters are as follows:

- σ: 2
- T_{low}: 0.09
- T_{high}: 0.11

The resulting image for these hyper-parameters are as follows:

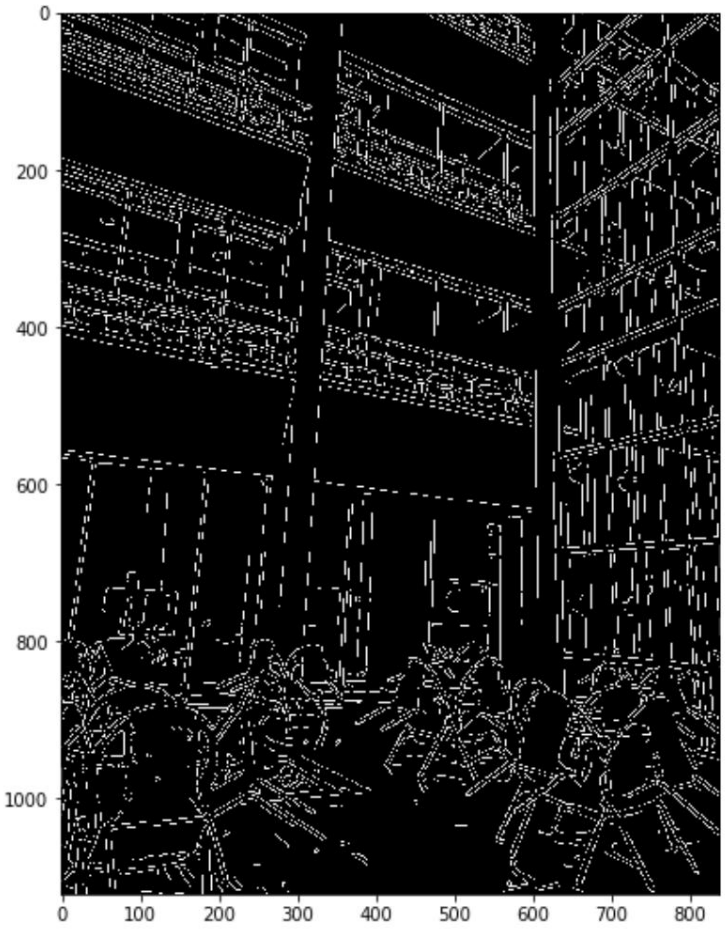

Figure 20: Best Edge Detection result obtained for edge2.png

edge3.png: Different than the other two images, the eye image involved significant amount of noise. Due to this fact, in these trials the center of the attempts was the variable σ. As shown in the attempts, the decreasing amount of noise always resulted with better edge detection results. Due to that the sigma value selected for the best attempt was 4. As another point, with the experiments at sigma = 3, it can be seen that the pixel values between 0.15 and 0.2 resulted with a lot of undesired edges. By setting the high threshold to 0.2, these edges are eliminated. Due to that, the high threshold value in the last attempt is set as 0.2. Resulting, the fourth trial was the best in the sense that the structure of the eye is clear mostly. The hyper-parameters for the best trial is as follows:

- σ: 4
- T_{low}: 0.03

- T_{high}: 0.2

The resulting image for the hyper-parameters is as follows:

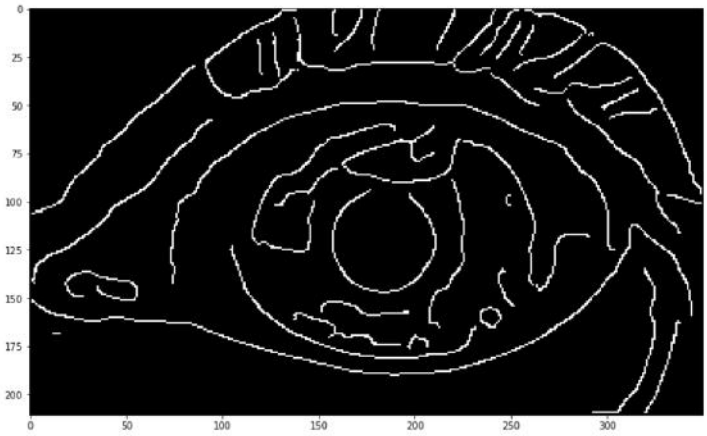

Figure 21: Best Edge Detection result obtained for edge3.png

1.2.4 Comparing Canny Edge Detector with Sobel & Prewitt Operators

As it is explained in the discussion for the Canny Edge Detector, it is mentioned that the basic edge detection with Sobel filter is a stage of this algorithm. For finding the gradients, the Sobel filter is being used in Canny Edge Detector. The main difference that Canny Edge Detector creates is examining the edges detected and applying Gaussian filter for noise removal for better detection for edges. Canny Edge Detector adds modifications the edges found like manipulating its thickness and adding selectivity for the edges (marking as strong or weak). As the result of that the edge detection task has been done in a better way. The edges can be examined. In these regards, Canny Edge Detector outperforms the Sobel and Prewitt filters. But the main issue is that Sobel and Prewitt operators are subroutines that are being used for Gradient Calculation stage of Canny Edge Detection algorithm. For the sake of comparison, applying Noise Reduction and Double Thresholding resulted with improvement in Edge Detection task.

2 Edge Linking with Hough Transform

2.1 Implementation of Hough Transform

For the implementation for the Hough Transform the main idea is using the line equation

$$y = mx + b \tag{6}$$

As mentioned in the lecture slides and the book from Shapiro and Stockman [3], using this line equation for Hough Transform is not feasible due to memory constraints of a computer. For a regular line equation, there would be infinitely many slopes to be tried manually. With the approximation by using angles, the following equation for a typical line is obtained:

$$y = -\frac{\cos\theta}{\sin\theta} + \frac{\rho}{\sin\theta} \tag{7}$$

$$\rho = x\cos\theta + y\sin\theta \tag{8}$$

Considering the possible values of θ, for an image by covering 180 values as angles whole line possibilities could be tried accordingly. Here the only approximation done was thinking that the angles are integers (in degrees), which creates a small error in terms of accuracy of the lines drawn. As the initial implementation, a grid like object (2D matrix to be precise) has been created. Here in order to obey the implementation limits, a rounding to integer operation has been done for the ρ values found. Even though this issue was not likely to create a huge difference, with the results obtained it is observed that this approximation introduced error for the constant factor in the line equation ($b = \frac{\rho}{\sin\theta}$). Even though some error has been introduced, this is a negligible difference. Here note that the variable ρ is the distance of the pixel to the origin in Hough space and θ is the respective angle. The initial(rounding) implementation for the Hough Transform with the grid logic is as follows:

```
def perform_hough(image):
    # Theta values in range (-90, 90)
    theta_values = np.arange(0, 180)
    image_diagonal = int(np.ceil(np.sqrt(image.shape[0] ** 2 + image.shape[1] ** 2)))
    vote_grid = np.zeros((2 * image_diagonal, len(theta_values)))
    for x_idx in range(image.shape[0]):
        for y_idx in range(image.shape[1]):
            for theta in theta_values:
                if image[x_idx, y_idx] != 0:
                    d = x_idx * math.cos(math.radians(theta)) + y_idx * math.sin(math.radians(theta))
                    if d % 1 < 0.5:
                        d = int(d)
                    else:
                        d = int(d) + 1
                    vote_grid[int(d) + image_diagonal, theta] += 1
    return vote_grid
```

As explained above, the rounding done created some error for the constant in the line equation. This error was not that significant, the results of the line fitting operation is given as follows:

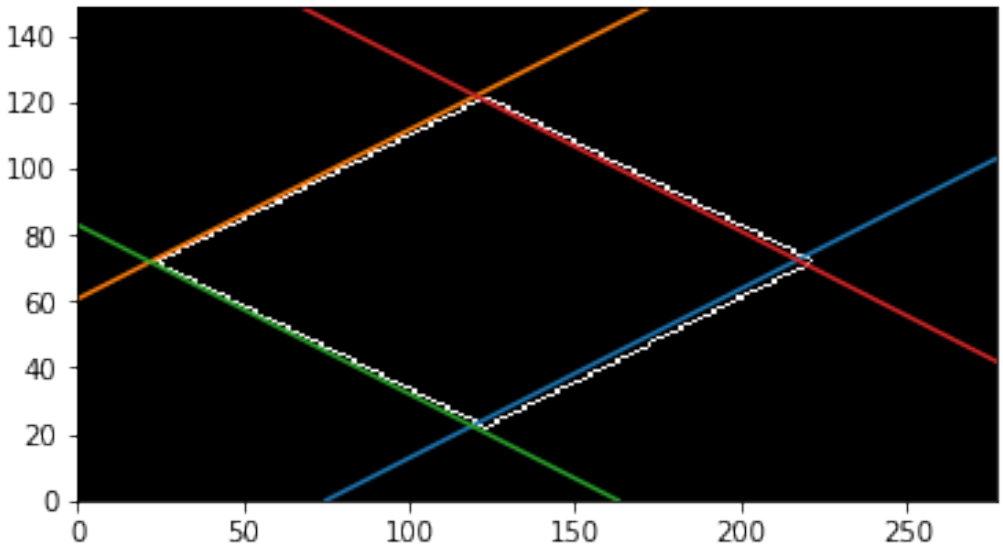

Figure 22: Result of the rounding implementation of Hough Transform

The main idea behind the rounding approach was being able to access to the votes in linear time. However, for accurate results a different implementation sacrificing time efficiency has been implemented. In the following implementation, sets are being used for efficient access and the lines found are stored in a set as (ρ, θ) pairs. Then the corresponding votes are being stored as dictionary elements which use these pairs as keys. Again, since the access to dictionaries and sets are the most efficient, these data structures are being used. The dictionary stores the votes is named as **hough_votes** and the set storing the lines discovered is named as **found_lines**. The check if the line was found is done by using set membership operation.

```
def perform_hough(image, desired_fp):
    # Theta values in range (-90, 90)
    theta_values = np.arange(-90, 90)
    hough_votes = {}
    found_lines = set([])
    for x_idx in range(image.shape[0]):
        for y_idx in range(image.shape[1]):
            for theta in np.radians(theta_values):
                if image[x_idx, y_idx] != 0:
                    d = round(x_idx * np.cos(theta) + y_idx * np.sin(theta), desired_fp)
```

```
                if (d, math.degrees(theta)) in found_lines:
                    hough_votes[(d, np.degrees(theta))] += 1
                else:
                    hough_votes[(d,math.degrees(theta))] = 1
                    found_lines.add((d, np.degrees(theta)))
    return hough_votes
```

This implementation is the one that considers floating points. As no rounding is done, the results obtained was perfectly accurate. However, since the floating points are preserved until a point in ρ values, multiple lines differing in very little amount has been found. In the visualization these lines can be seen as overlapping lines. To explain the concept in detail, the (ρ, θ) pairs found for hough.png is given:

```
(75.4, 27.0), (75.3, 27.0), (75.2, 27.0), (164.3, 27.0), (164.2, 27.0)
(-36.3, -27.0), (-35.9, -27.0),  (-36.4, -27.0), (-36.0, -27.0),  (-36.5, -27.0), (-36.1, -27.0),
(52.6, -27.0),  (52.5, -27.0)
```

As it can be seen here most of these lines are nearly identical, but to preserve this accuracy, no preventive action is taken. The floating point accuracy can be changed in the implementation with the attribute desired_fp. The results obtained after visualizing these lines found is given as follows: Results of floating point approach:

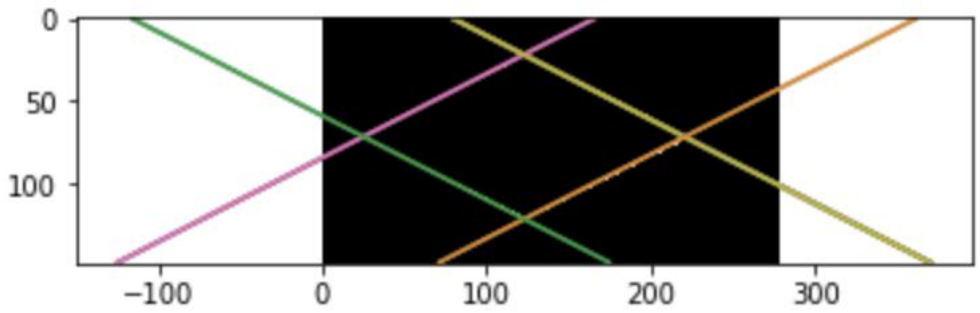

Figure 23: Result of the floating point implementation of Hough Transform

As it can be seen here, the problem in the accuracy of the lines found has been solved clearly. As another step for the implementation, these close lines could have been averaged but for the purpose of this assignment, this is not attempted.

Considering the trade-off between time efficiency and the gain in accuracy, the rounding approach is preferred. In the experiments made during implementation, set based approach introduced significant delay due to search in set. Considering that, the rounding implementation is preferred for this implementation. The rounding approach is the implementation given in the submission.

2.2 Applying Hough Transform

For both making use of thresholding and visualizing the lines with the image, the following function has been used:

```
def find_best_lines(votes, threshold):
    best_lines = []
    sorted_votes = np.sort(np.unique(votes))
    if threshold > sorted_votes.size:
        best_vote = sorted_votes[0]
    else:
        best_vote = sorted_votes[-threshold]
    diagonal = int(votes.shape[0] / 2)
    for x_idx in range(votes.shape[0]):
        for y_idx in range(votes.shape[1]):
            if votes[x_idx, y_idx] >= best_vote:
                theta = y_idx
```

```
                best_lines.append((x_idx - diagonal, theta))
    return best_lines
```

Here the unique vote scores are sorted and the first n values are selected, which is specified by the variable *threshold*. Here the selected lines are the ones having the vote greater or equal to $n'th$ biggest vote. At the final stage of line selection, the theta value was not converted in a way that obeys the rule $|\theta| \leq 90$. This was not an issue for matplotlib in plotting. In order to use negative ρ values, the addition and subtraction of the maximum possible rho value has been performed, which is the diagonal of the image.

2.2.1 Finding lines for image hough3.png

For this part of the assignment, the image selected was *hough3.png*. The original form of the image is shown as follows:

Figure 24: The image hough3.png as given in the assignment

For finding the lines in the image, the following method has been used:

```
def find_lines_part_2_1():
    hough_2 = dataset.get_hough_images()[2]
    edge_hough = feature.canny(hough_2, sigma = 3)
    accumulator = perform_hough(edge_hough)
    best_lines = find_best_lines(accumulator, 30)
    print(best_lines)
    fig, ax = plt.subplots()
    ax.set_xlim(0, hough_2.shape[1])
    ax.set_ylim(0, hough_2.shape[0])
    ax.imshow(hough_2, cmap = 'Greys_r', extent =
        [0, hough_2.shape[1], 0, hough_2.shape[0]])
    x = np.array(range(hough_2.shape[0]))
    for line_idx in range(len(best_lines)):
        theta = math.radians(best_lines[line_idx][1])
        if best_lines[line_idx][1] != 0:
            y = -math.cos(theta) / math.sin(theta) * x +
                (best_lines[line_idx][0] / math.sin(theta))
            ax.plot(y,x)
        else:
            y = best_lines[line_idx][0] * np.ones(hough_2.shape[1])
            ax.plot(np.array(range(hough_2.shape[1])),y)
```

In this plotting approach, an approximation for the angle zero has been done. If the respective theta is found as 0, then the line is simulated as the line is $x = \rho$ at every point. then the plot has been drawn as a straight line accıdingly. The resultant image is shown below:

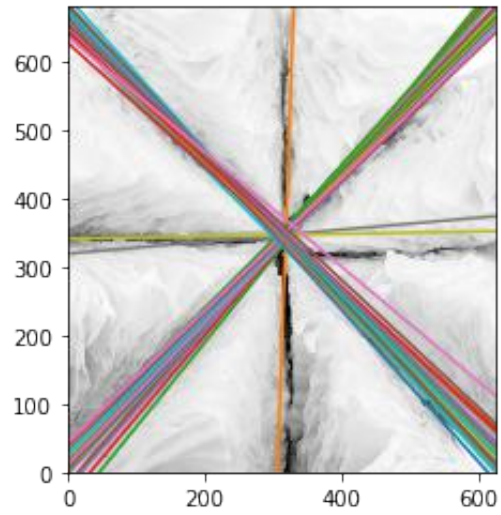

Figure 25: Applying Hough Transform on image *hough_ 3.png*

As it can be seen from the resulting image, for the vertical space and horizontal spaces between the slices, the lines were easily be able to fitted and most of the line predictions has been discovered in these regions. Considering the fact that the rounding approach introduces uncertainty, lines that are close to each other could e considered as expected. Here since the horizontal line between the slices is interrupted due to the alignment of the slices, it is expected that the votes for that line would be lower. This result can be observed from the figure above. Other than that the issues that the rounding approach generates are also significant. However, considering the extraction of the features, the algorithm successfully found the most significant lines, which are the diagonal lines and the vertical line. This performance can possibly be increased by setting better parameters for the Canny Edge Detector, which is the edge detection algorithm selected for this task.

Specific for this image, the result obtained with threshold value 30. This means that the lines that has the vote that is equal or higher from the 30^{th} highest vote are selected. This is set by using the **threshold** parameter in the function *find_ best_ lines*. For a clear view of the points where the votes are heavily populated the hough space has been visualized. The visualization of the Hough Space is as follows:

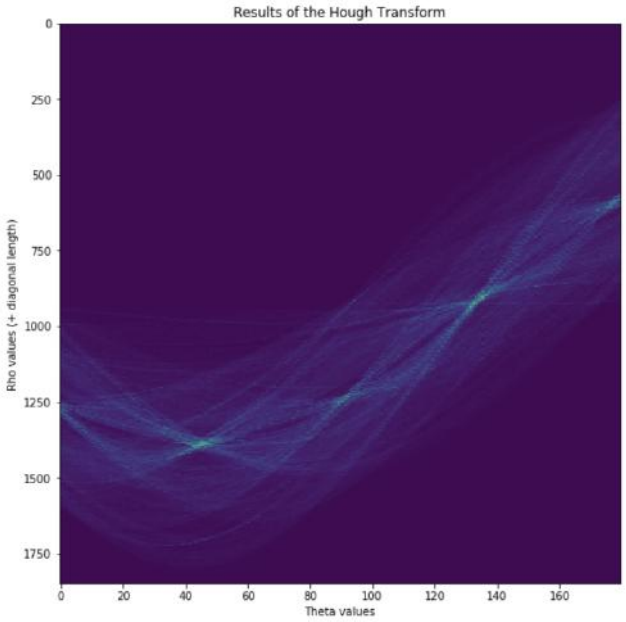

Figure 26: Hough Space after applying Hough Transform on image *hough_ 3.png*

Here it is obvious that the votes given are focused on 4 main points (last one is at angle 180 (0)). These

lines correspond to the 4 main lines detected in the previous figure. Here note that the values corresponding to rho values are multiplied with the diagonal value of the image for proper indexing. Here diagonal length refers to the following equation:

$$Diagonal = \sqrt{(Number\,of\,Rows)^2 + (Number\,of\,Columns)^2} \tag{9}$$

For the focus points that are approximately at theta value = 45 and 135, these are some fluctuations. As it can be seen from the Hough Space visualization, the significant area where the votes are high is not a point but an area. This shows the fact that there are many lines that are close to each other, which shows the presence as the diagonal lines in figure 25. This is the main reason why the threshold value was 30.

2.3 Finding the Horizon Line and Evaluating the Implementation

Just like it was applied to image hough_3.png, the hough transform has been applied to image edge_1.png also. For clarity, the original image and the result of edge detection task is given below. Note that the edge detection task has been done as a component of the first part of the assignment.

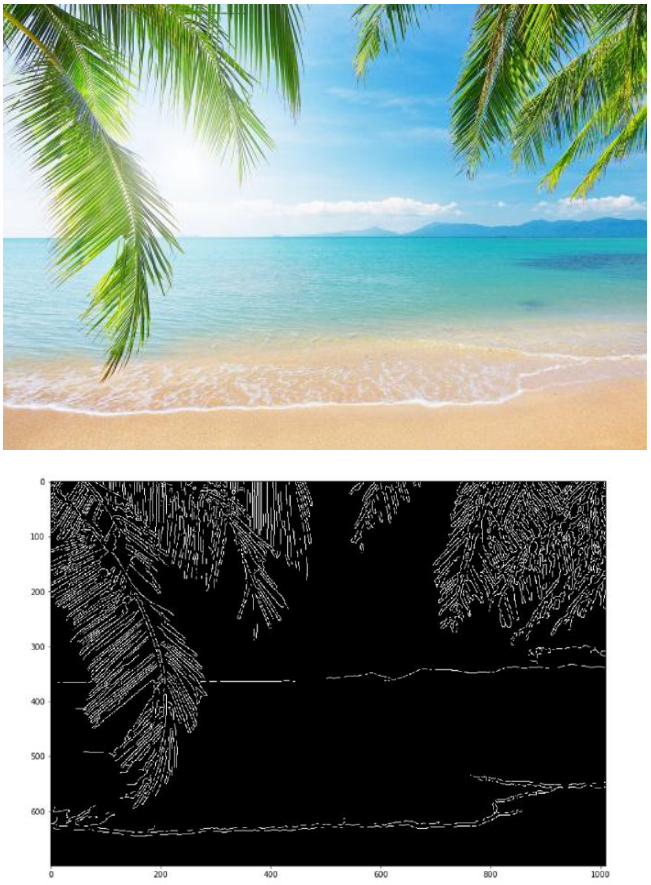

Figure 27: Original form and the Edge Detection result for the image edge_1.png

As it can be observed from the second image, the line parts showing the horizon line are most likely the most significant line present in the image, if the edges were going to be linked. After that the attempts will focus on fitting a line to tree that is in foreground. To question the truth of this observation and examining the performance of the Hough Transform implemented, the Hough Space has been visualized. The hough space for image edge_1.png is as follows:

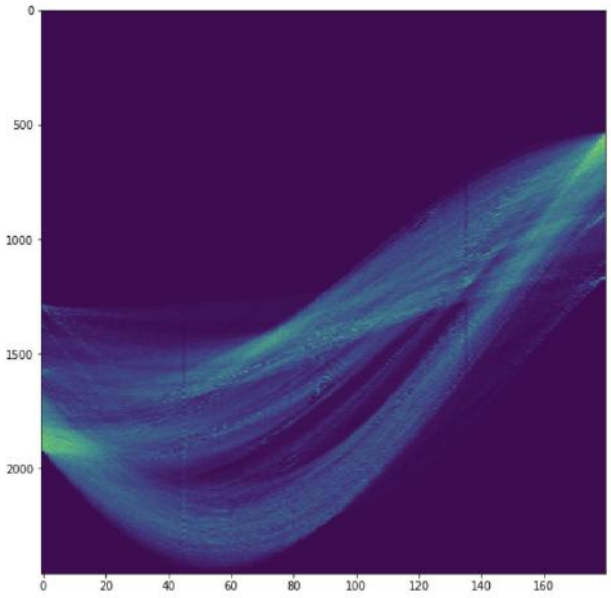

Figure 28: Original form and the Edge Detection result for the image edge_1.png

Here the algorithm shows its success in terms of the truth of the implementation according to the observation. The success of the algorithm is also shown with the image hough.png in the previous parts.

As the next step, the horizon line is searched. Here the threshold value is set to 1, as it is clearly the most significant line due to results of edge detection. After foundations, the result is visualized using the following method where there is an approximation for $\theta = 0$. The method used is as follows:

```
def find_lines_part_2_2():
    edge_1 = dataset.get_edge_images()[0]
    edge_1 = np.flipud(edge_1)
    edge_img = feature.canny(edge_1, sigma = 2)
    fig, (ax1, ax2) = plt.subplots(1,2, figsize = (20,20))
    accumulator = perform_hough(edge_img)
    ax1.imshow(accumulator)
    ax1.set_aspect(accumulator.shape[1] / accumulator.shape[0])
    best_lines = find_best_lines(accumulator, 1)
    ax2.imshow(edge_1, cmap = 'Greys_r', origin = 'upper')
    ax2.set_xlim(0, edge_1.shape[1])
    ax2.set_ylim(0, edge_1.shape[0])
    x = np.array(range(edge_1.shape[0]))
    for line_idx in range(len(best_lines)):
        theta = math.radians(best_lines[line_idx][1])
        if math.sin(theta) != 0:
            y = -math.cos(theta) / math.sin(theta) * x +
                (best_lines[line_idx][0] / math.sin(theta))
            ax2.plot(y,x)
        else:
            # Simulating infinite slope
            y = best_lines[line_idx][0] * np.ones(edge_1.shape[1])
            ax2.plot(np.array(range(edge_1.shape[1])),y)
find_lines_part_2_2()
```

As the result of this calculation, the image and the corresponding best line fitted is as follows:

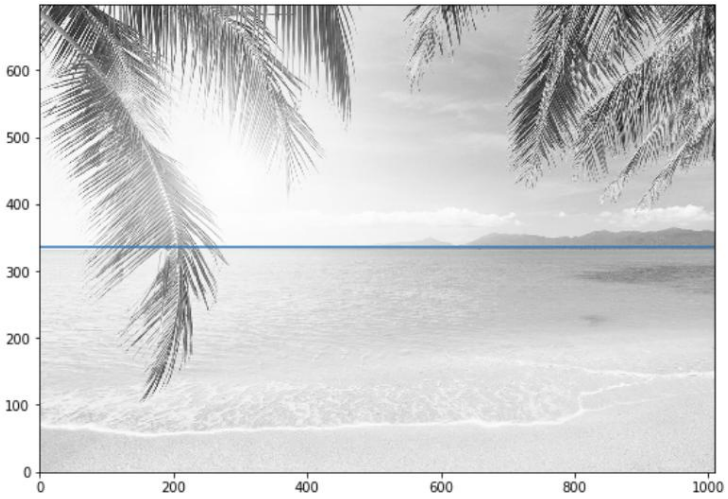

Figure 29: Line of the Horizon found on image edge1.png after applying Hough Transform

As the results prove both in the visualization of the line found and the visualization of the Hough Space, the algorithm works correctly. However note that the success of the implementation is mainly dependant on the result of the edge detection value, which is why the value $\sigma = 2$ is used for this case. For the threshold value, the threshold value 1 is used due to its significance, with higher threshold values it is expected to have lines at th top side of the image. In general this implementation can be considered as a success.

References

Deepika Adlakha, D. A., & Tanwar, R. (2016). Analytical comparison between sobel and prewitt edge detection techniques. *International Journal of Scientific and Engineering Research*, 1482–1485.

Sahir, S. (2019). *Canny edge detection step by step in python — computer vision.* Retrieved 2020-04-16, from `https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123`

Shapiro, L., & Stockman, G. (2001). *Computer vision.* Pearson.

Tsankashvili, N. (2018). *Comparing edge detection methods.* Retrieved 2020-04-16, from `https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e`