Homework 3 Report

Ayhan Okuyan
ayhan.okuyan[at]bilkent.edu.tr

December 20, 2020

Contents

Part 1: Edge Detection

In this part of the homework assignment, we are asked to implement and run functions that enable edge detection in grayscale images, namely, Prewitt and Sobel filters, and the Canny Edge Detector. To test the algorithms, we are given three images with the names 'edge.png', 'edge2.png' and 'edge3.png', whose grayscaled versions are as given in Figure 1

(a) edge.png (b) edge2.png (c) edge3.png

Figure 1: Test Images in Grayscale

1.1 Edge Detection with Sobel and Prewitt Operators

Here, we use both horizontal and vertical filters to obtain the directional gradient, and then also show the magintude of the gradient as the filtered image.

1.1.1 3x3 Sobel Operators

3x3 Sobel operators in x and y directions are presented below.

$$s_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{1}$$

$$s_y = s_x^T = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

In order to find the outputs, we convolve the image with the filters separately, and the find the pixel-wise norm of the image as described.

$$g_x = I * s_x \tag{3}$$

$$g_y = I * s_y \tag{4}$$

$$||g|| = \sqrt{g_x^2 + g_y^2} \tag{5}$$

The results for each of the images with their vertical, horizontal edges and the magnitudes are given below in figures 2, 3 and 4.

Figure 2: Sobel Filtered 'edge.png'

Figure 3: Sobel Filtered 'edge2.png'

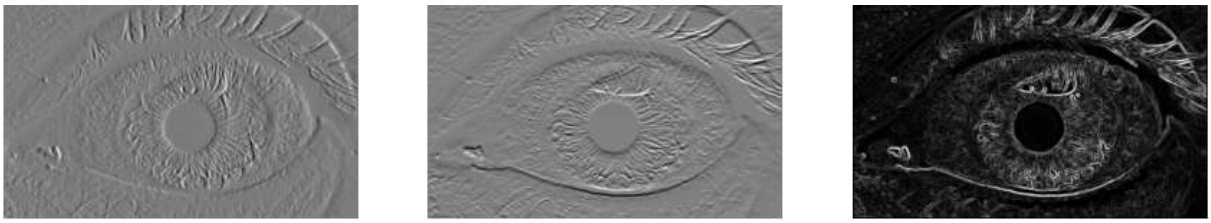

Figure 4: Sobel Filtered 'edge3.png'

The sobel function is as follows.

```
def sobel(img, return_grads=False):
    #define the function to convolve for the any square filter with zero padding.
    def convolve(image, kernel):
        result = np.zeros(image.shape)
        #set upper/lower size accroding to the filter size
        if kernel.shape[0] % 2 == 1:
            ksize = int((kernel.shape[0]-1)/2)
        else:
            ksize = int(kernel.shape[0]/2)

        for i in range(ksize,image.shape[0]-ksize):
            for j in range(ksize,image.shape[1]-ksize):
                result[i,j] = np.sum(image[i-ksize:i+ksize+1,j-ksize:j+ksize+1] * kernel)
        return result

    #Define operators for vertical and horizontal edge detection
    sobelx = np.array([[1,0,-1], [2,0,-2], [1,0,-1]])
    sobely = np.array([[1,2,1], [0,0,0], [-1,-2,-1]])

    #apply filtering for image and filters
```

```
    resx = convolve(image,sobelx)
    resy = convolve(image,sobely)
    #find magnitude
    magnitude = np.sqrt(resx**2+resy**2)

    if return_grads:
        return [resx,resy,magnitude]
    else:
        return magnitude
```

1.1.2 3x3 Prewitt Operators

3x3 Prewitt operators in x and y directions are presented below.

$$p_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{6}$$

$$p_y = p_x^T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{7}$$

In order to find the outputs, we convolve the image with the filters separately, and the find the pixel-wise norm of the image as described in the previous part.

$$g_x = I * p_x \tag{8}$$

$$g_y = I * p_y \tag{9}$$

$$||g|| = \sqrt{g_x^2 + g_y^2} \tag{10}$$

The results for each of the images with their vertical, horizontal edges and the magnitudes are given below in figures 5, 6 and 7.

Figure 5: Prewitt Filtered 'edge.png'

The code is given as follows.

```
def prewitt(img, return_grads=False):
    #define the function to convolve for the any square filter with zero padding.
    def convolve(image, kernel):
        result = np.zeros(image.shape)
        #set upper/lower size accroding to the filter size
        if kernel.shape[0] % 2 == 1:
            ksize = int((kernel.shape[0]-1)/2)
        else:
            ksize = int(kernel.shape[0]/2)

        for i in range(ksize,image.shape[0]-ksize):
            for j in range(ksize,image.shape[1]-ksize):
                result[i,j] = np.sum(image[i-ksize:i+ksize+1,j-ksize:j+ksize+1] * kernel)
        return result
```

Figure 6: Prewitt Filtered 'edge2.png'

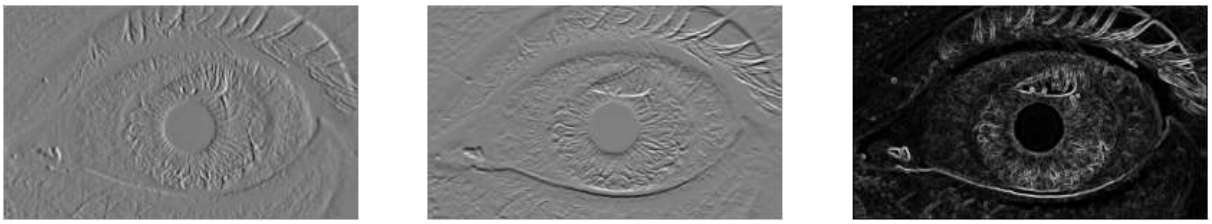

Figure 7: Prewitt Filtered 'edge3.png'

```
#Define operators for vertical and horizontal edge detection
prewittx = np.array([[1,0,-1], [1,0,-1], [1,0,-1]])
prewitty = np.array([[1,1,1], [0,0,0], [-1,-1,-1]])

#apply filtering for image and filters
resx = convolve(image,prewittx)
resy = convolve(image,prewitty)
#find magnitude
magnitude = np.sqrt(resx**2+resy**2)

if return_grads:
    return [resx,resy,magnitude]
else:
    return magnitude
```

1.1.3 Comments

As the filters are presented in previous sections, they are not identical in terms of value although they are close. They are close in the sense that they are both trying to estimate the first derivatives of the image. We can see the differences when we observe the x and y direction gradients. With the high resolution of the images displayed, we can see that there is no significant differences in the way the edges are shaped. By assigning a higher weight to the closer pixels of the filter mask, the edges that are thinner can be detected more easily. Although it is unobservable from the images, we expect to see sharper edges with the Sobel operation.

1.2 Edge detection with Canny edge detector

In this part, we are asked to use the Canny edge detector to detect the edges of the test images. For that, we have implemented our own Canny from scratch to understand the underlying mechanics better and

then compared it with scikit image's Canny method and moved forwards with the one that gave better results. Canny edge detector consists of five steps given as:

- Gaussian Smoothing
- Edge Filtering
- Norm and Direction Calculation of Gradients
- Non-Maximum Suppression
- Hysteresis Thresholding

We will be explaining each of them separately.

1.2.1 Gaussian Smoothing

The first step we have done is to smooth the image with a 5-by-5 Gaussian filter, whose standard deviation is a hyperparameter. We construct the filter as follows.

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left\{ \frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2} \right\}, \quad 1 \leq i,j \leq 2k+1 \tag{11}$$

```
def gaussianFilter(std, size=5):
    filter = np.zeros((size,size))
    var = std**2
    shift = (size-1)/2

    for i in range(size):
        for j in range(size):
            filter[i,j] = np.exp(- ((i-shift)**2 + (j-shift)**2) / (2*var) )
    filter /= (2*np.pi*var)
    return filter
```

Then, we convolve the image with the filter using the same convolution function defined on the previous parts. This way, the algorithm becomes more robust to noise.

1.2.2 Edge Filtering

Then, we use either Sobel or Prewitt algorithm to derive the directional gradients from the blurred image. The same operations are applied as in Sections 1.1.1 and 1.1.2. For this implementation, while we present the choice to the user, we mostly preferred the Sobel filter over Prewitt.

1.2.3 Norm and Direction Calculation of Gradients

After finding G_x, G_y we find the magnitude and direction of the image intensities as follows.

$$||G||_2 = \sqrt{g_x^2 + g_y^2} \tag{12}$$

$$\angle G = tan^{-1}\left(\frac{g_y}{g_x}\right) \tag{13}$$

The code is as follows.

```
def magdir(image, filters):
        # assumes vertical filter is given first and horizontal second
        xgrad = convolve(image,filters[0])
        ygrad = convolve(image,filters[1])

        # find gradient
        grad = np.sqrt(xgrad**2 + ygrad**2)
```

```
        grad = grad/np.max(grad) * 255
        # find direction
        direction = np.arctan2(ygrad, xgrad)

        return grad,direction
```

1.2.4 Non-Maximum Suppression

It is noticeable that the intensity images obtained from filtering have wide edges on them. To make these edges thinner, non-maximum suppression (an edge thinning technique) is applied. the algorithm states that if a pixel's intensity is bigger than the two interpolated pixels in its gradient direction, then they are considered edge pixels, and else is not. Also, we set the range of the angles to $[-\pi, \pi)$ and converted the negative angles to postitive by adding π to the angle, which essentially converges to the same calculations. For our implementation, we have used a simplification of this method where we quantized the angles to the bins of $0, 45, 90, 135, 180°$. Which enabled us to approximate the interpolated pixels. The function is as follows.

```
def suppress(grad, direction):
    res_img = np.zeros(grad.shape)
    for i in range(1,grad.shape[0]-1):
        for j in range(1,grad.shape[1]-1):

            #convert the negative angles to positive scale
            if(direction[i,j] < 0):
                direction[i,j] += np.pi

            p1 = 255
            p2 = 255

            if (0 <= direction[i,j] < np.pi/8) or (7*np.pi/8 <= direction[i,j] <= np.pi):
                p1 = grad[i, j+1]
                p2 = grad[i, j-1]
            elif (np.pi/8 <= direction[i,j] < 3*np.pi/8):
                p1 = grad[i+1, j-1]
                p2 = grad[i-1, j+1]
            elif (3*np.pi/8 <= direction[i,j] < 5*np.pi/8):
                p1 = grad[i+1, j]
                p2 = grad[i-1, j]
            elif (5*np.pi/8 <= direction[i,j] < 7*np.pi/8):
                p1 = grad[i-1, j-1]
                p2 = grad[i+1, j+1]

            if (grad[i,j] >= p1) and (grad[i,j] >= p2):
                res_img[i,j] = grad[i,j]
    return res_img
```

1.2.5 Hysteresis Thresholding

After thinning the edges, we perform a double-thresholding operation on the obtained image to eliminate the noise obtained suppression and to connect the edges that are separated. For implementation, two separate thresholds are used given as t_{low} and t_{high}. t_{high} is a hard-threshold defined in the range $[0, 1]$. the algorithm first selects the pixels whose values are greater than t_{high}, and calls it the **strong** pixels. Then, a Connected Component Analysis (CCA, blob analysis) is applied, meaning for each pixel that is in range $[t_{low}, t_{high}]$, it is considered as a weak pixel if at least one out of eight of its neighboring pixels are strong. The ones that are left are discarded. This procedure gives the end image as the result. The code is given as follows.

```
def hysteresisThresholding(img, t_low, t_high):
    # scale the thresholds for the img. Expecting both inputs to be in between (0,1)
    # high threshold is scaled for the image
```

```
    t_high = np.max(img) * t_high
    # low threshold is scaled for the image.
    t_low = np.max(img) * t_low
    # raise error if low threshold is bigger than high threshold
    if t_low > t_high:
        raise ValueError('t_low must be smaller than or equal to t_high')

    #apply hysteresis thresholding
    res = np.zeros(img.shape)
    # set strong edges first
    res[img >= t_high] = 255

    #set weak edges with strong connections
    for i in range(1,img.shape[0]-1):
        for j in range(1,img.shape[1]-1):
            if(img[i,j] < t_high and img[i,j] >= t_low and
               (res[i+1,j] == 255 or res[i+1,j+1] == 255 or res[i,j+1] == 255 or res[i-1,
                                                          j+1] == 255 or
                res[i-1,j] == 255 or res[i-1,j-1] == 255 or res[i,j-1] == 255 or res[i+1,
                                                           j-1] == 255)):
                res[i,j] = 255

    return res
```

For better understanding, we have shown these steps on one of the test images as presented in Figure 8.

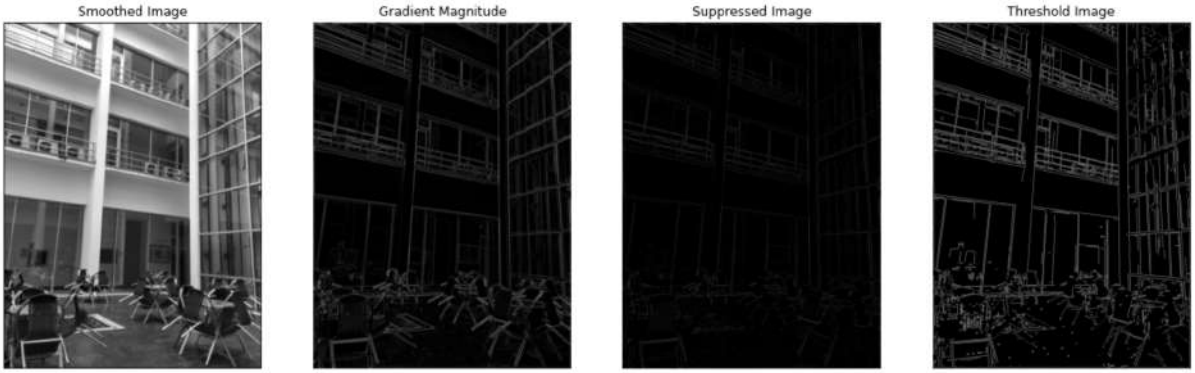

Figure 8: Individual Steps of the Canny Edge Detection

After completing the implementation, we have observed the images we obtained with the scikit-image implementation using the same parameters for standard deviation, and low and high thresholds. The images are provided in Figure 9

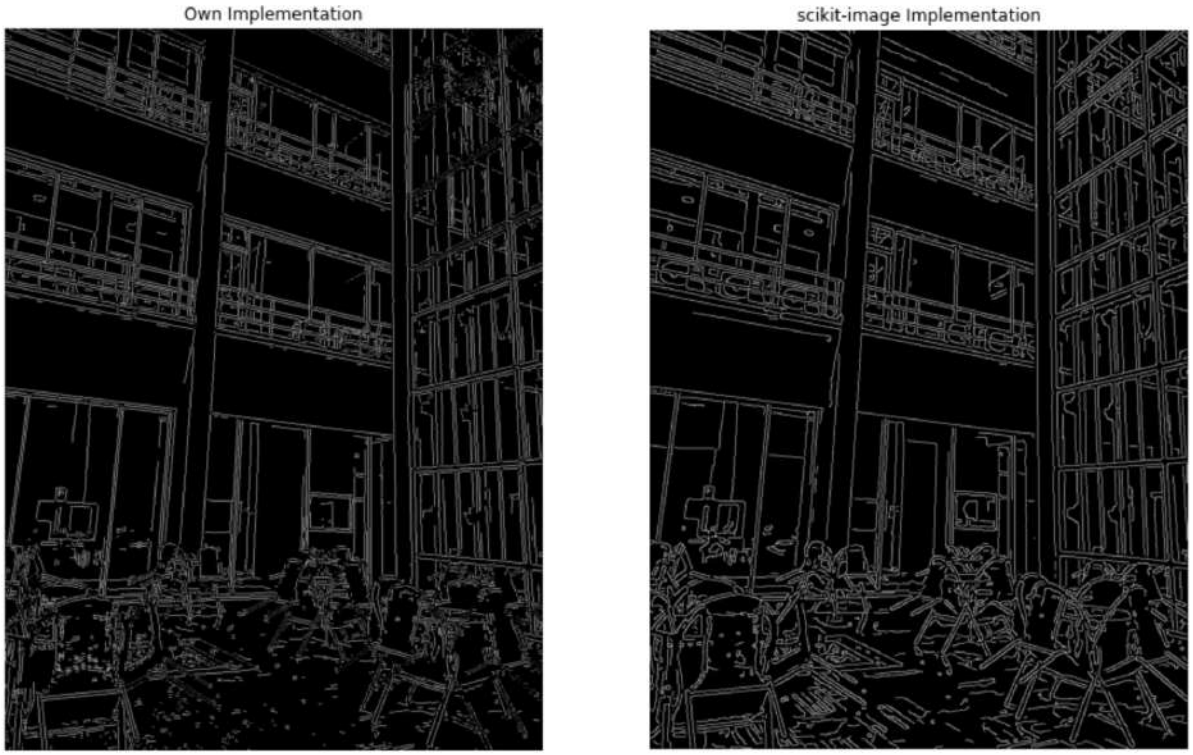

Figure 9: From Scratch vs Sci-Kit Image Canny on 'edge2.png'

We can see that with a high resolution image with mostly linear components, our implementation worked if not better, as good as the sk-image implementation. Now, we try it on 'edge3.png', which is a more low resolution image with natural curves (see Figure 10).

Figure 10: From Scratch vs Sci-Kit Image Canny on 'edge3.png'

From Figure 10, it is observable that the sk-image implementation has shown much more satisfying results. Our algorithm performed poorly in capturing the natural curves since we have used an approximation technique in non-maximum suppression. For further experiments, we have decided to move on with the sk-image implementation.

1.2.6 Reporting Best Values

In this part, we report the best values we have encountered for each image. For 'edge.png', the best values are $\sigma = 2, t_{low} = 0.05, t_{high} = 0.2$. The result is as follows.

Figure 11: Best Canny Image for 'edge.png'

For 'edge2.png', the best values are $\sigma = 2, t_{low} = 0.2, t_{high} = 0.3$. The result is as follows.

Figure 12: Best Canny Image for 'edge2.png'

For 'edge3.png', the best values are $\sigma = 2, t_{low} = 0.06, t_{high} = 0.1$. The result is as follows.

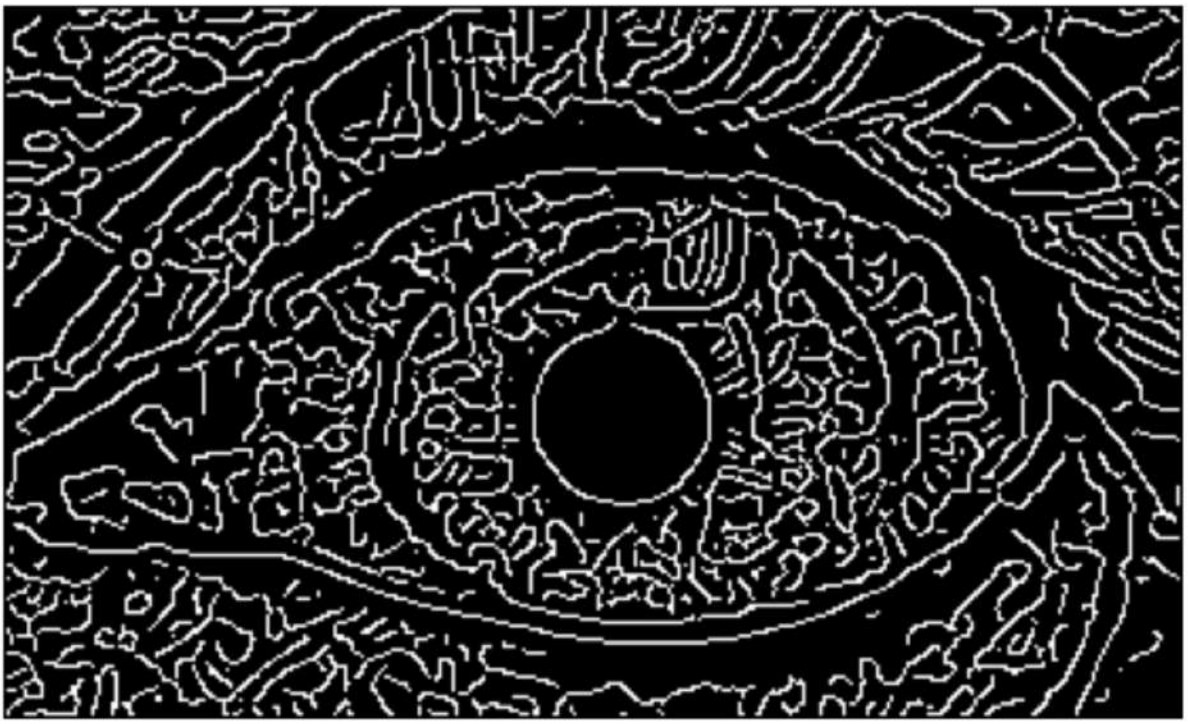

Figure 13: Best Canny Image for 'edge3.png'

We believe these are the best parameters since the resulting images are not overly crowded and there is ample space between edges and they are easily distinguishable.

1.2.7 Comparisons

In this part, we set four different sets of parameters for each of the images, and discuss what different parameters do in this system. The parameter set for 'edge.png' is given as:

$$(\sigma, t_{low}, t_{high}) = \{(0.5, 0.03, 0.1), (1, 0.05, 0.1), (2, 0.05, 0.2), (2, 0.2, 0.5)\} \tag{14}$$

The results are given in Figure 14. We can easily observe from here that the $\sigma < 0$ results in a noisy image and a too high high threshold results in loss of the edges. The parameter set for 'edge2.png' is given as:

$$(\sigma, t_{low}, t_{high}) = \{(0.5, 0.05, 0.1), (1, 0.1, 0.2), (2, 0.2, 0.3), (2, 0.3, 0.5)\} \tag{15}$$

The results are given in Figure 15. The parameter set for 'edge3.png' is given as:

$$(\sigma, t_{low}, t_{high}) = \{(0.5, 0.05, 0.2), (1, 0.1, 0.2), (2, 0.06, 0.1), (3, 0.03, 0.1)\} \tag{16}$$

The results are given in Figure 16.

1.2.8 Improvements

The most important difference between this algorithm and the previous operators is that it is a tuneable system. Also, we were able to see a major difference in the resulting images when compared. The potential reason for such improvement comes from the Non-Maximum Suppression and the Hysteresis Thresholding Algorithms. It is clear that the thinned lines connected back together with hysteresis results in a much more satisfying outcome.

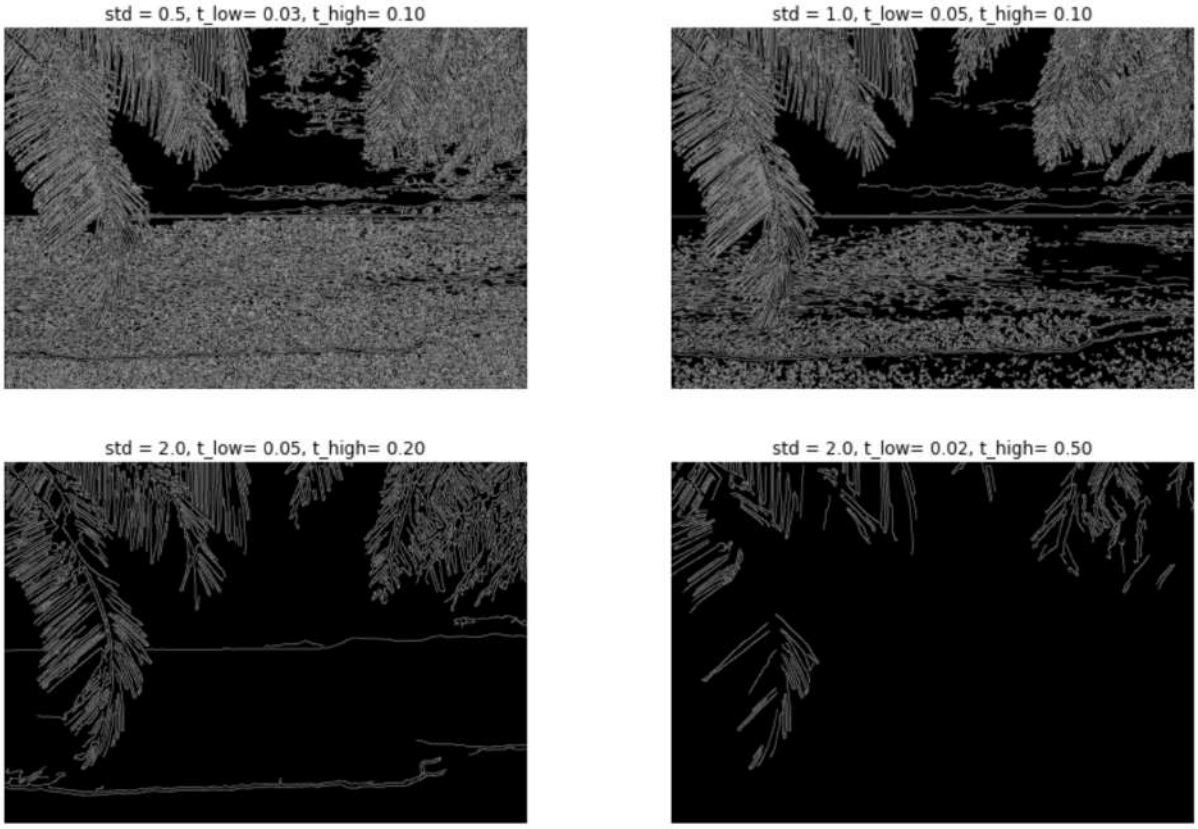

Figure 14: Sets of Parameters for 'edge.png'

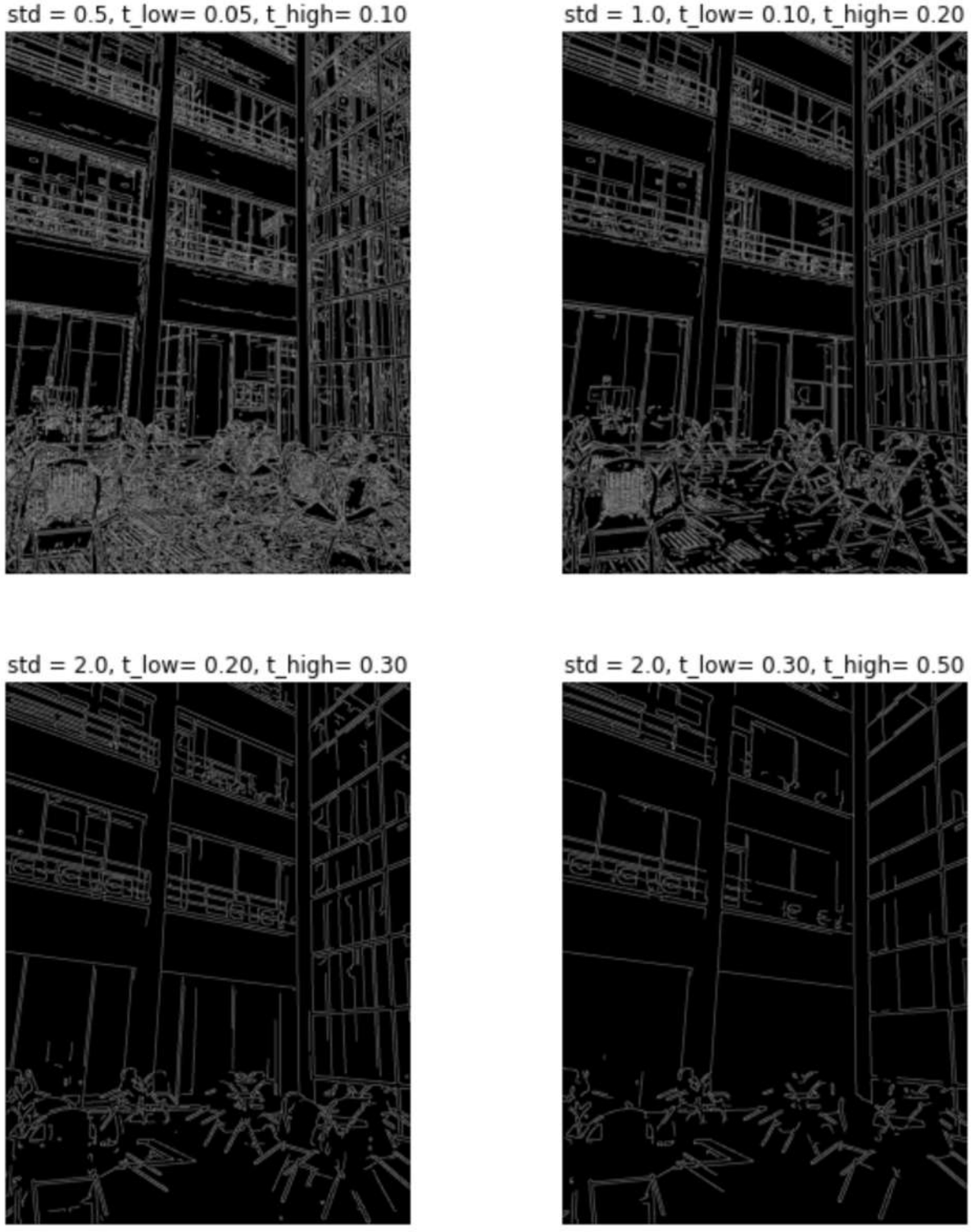

Figure 15: Sets of Parameters for 'edge2.png'

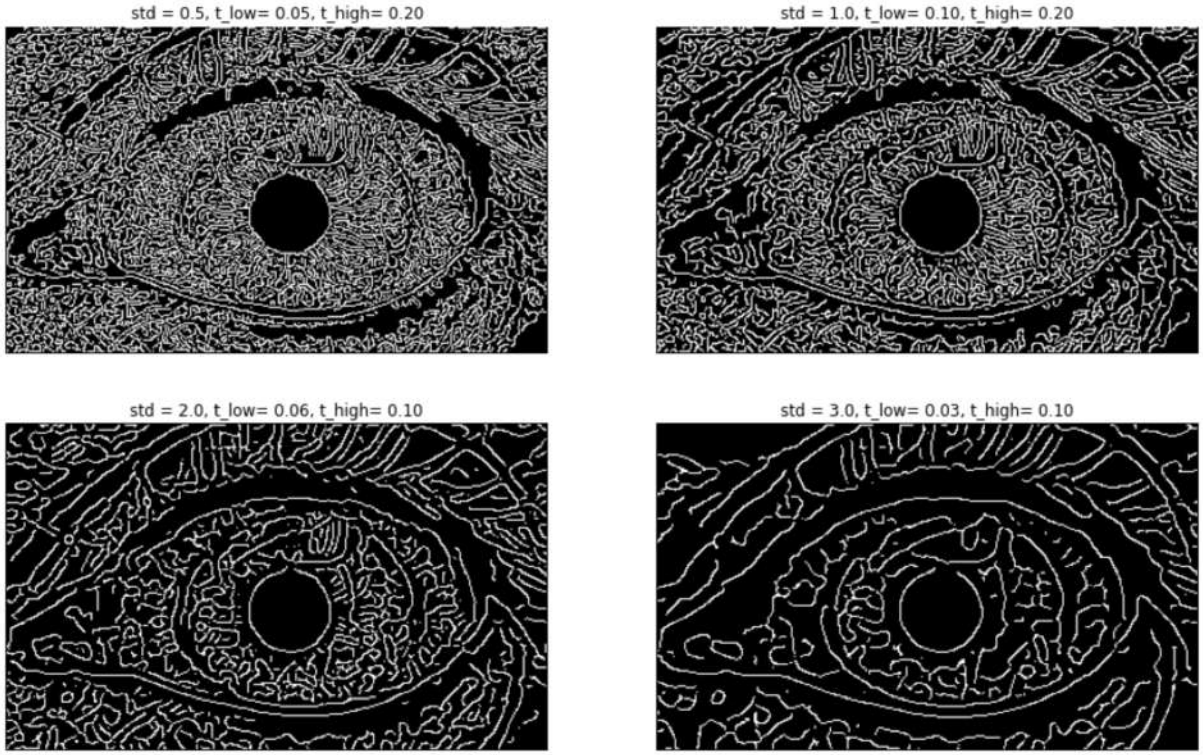

Figure 16: Sets of Parameters for 'edge3.png'

Part 2: Edge Linking with Hough Transform

In this part of the homework, we are asked to implement a Hough Transform algorithm that converts the input binary image to the Hough space given as $H(r, \theta)$ in polar coordinates. For implementation of this algorithm, we have applied the most realistic and precise form of Hough Transform. The basis of Hough transform depends on the ability to show lines with single parameters in polar domain as.

$$r - x\cos\theta - y\sin\theta = 0 \tag{17}$$

In the beginning of the algorithm, we prepare the angle and distance bins, which represent the axes of our Hough space. Then, for each pairs of points i.e. $p_1 = (x_1, y_1), p_2(x_2, y_2) | p_1 \neq p_2$, we define r and θ which are:

- r: The perpendicular distance from the origin to the line that passes through p_1 and p_2.
- θ: Angle between $y = 0$ and r.

We can find r using Cartesian Algebra easily as

$$r = \frac{|y_1 * \Delta x - x_1 * \Delta y|}{\sqrt{\Delta x^2 + \Delta y^2}}, \qquad \Delta x = x_2 - x_1, \Delta y = y_2 - y_1 \tag{18}$$

and then we can find θ as,

$$\theta = tan^{-1}\left(\frac{\Delta y}{\Delta x}\right) \tag{19}$$

for each (r, θ) pair, we save the value by incrementing the closest bin on the Hough space. We use the interval of angles $[-\pi/2, \pi/2)$ and distance $[-r_{max}, r_{max}]$ where r_{max} is the length of the diagonal of the image in pixels. However, as we have used pi angles, in order to prevent confusion, we have used the signed distance of the line from the image, which can be described as follows.

$$r = \frac{y_1 * \Delta x - x_1 * \Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}, \qquad \Delta x = x_2 - x_1, \Delta y = y_2 - y_1 \tag{20}$$

The code is as follows for the Hough Transform.

```
def houghTransform(image, angle_num=180, r_num=500):
    if len(np.unique(image)) > 2:
        image = feature.canny(image, sigma=1, high_threshold=0.5, low_threshold=0.3)
    x, y = image.shape
    max_r = int(np.ceil(np.sqrt(x**2+y**2)))

    theta_bins = np.linspace(-np.pi/2,np.pi/2, angle_num)
    r_bins = np.linspace(-max_r,max_r, r_num)

    hough_plane = np.zeros((len(r_bins),len(theta_bins)))
    valid_points = np.argwhere(image > 0)
    for i,pt in enumerate(valid_points):
        for j,pt2 in enumerate(valid_points):
            if j>i:
                xdif = pt2[0]- pt[0]
                ydif = pt2[1]- pt[1]
                r = (xdif*pt[1] - ydif*pt[0]) / np.sqrt(xdif**2 + ydif**2)
                theta = np.arctan2(ydif,xdif) - 1e-9

                r_ind = np.searchsorted(r_bins, r, side='left')
                theta_ind = np.searchsorted(theta_bins,theta,side='left')

                if(theta_ind == len(theta_bins)):
                    theta_ind -= 1
                if(r_ind == len(r_bins)):
                    r_ind -= 1

                hough_plane[r_ind, theta_ind] += 1

    return hough_plane, theta_bins, r_bins
```

2.1 Finding Parameters of 'hough.png'

The 'hough.png' is given below (see Figure 17).

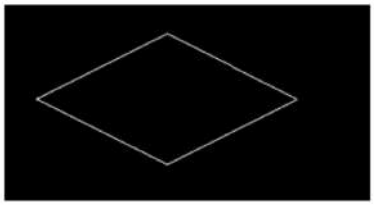

Figure 17: 'hough.png'

Then we apply the algorithm described and select four values (correspond to four straight lines in the image) (r, θ) that gives the highest values. Below (see Figure 18) is presented the Hough space, with the x coordinate representing the θ values, and y coordinates rs, together with the edges displayed on the image. The Hough space clearly shows four points, representing the lines.

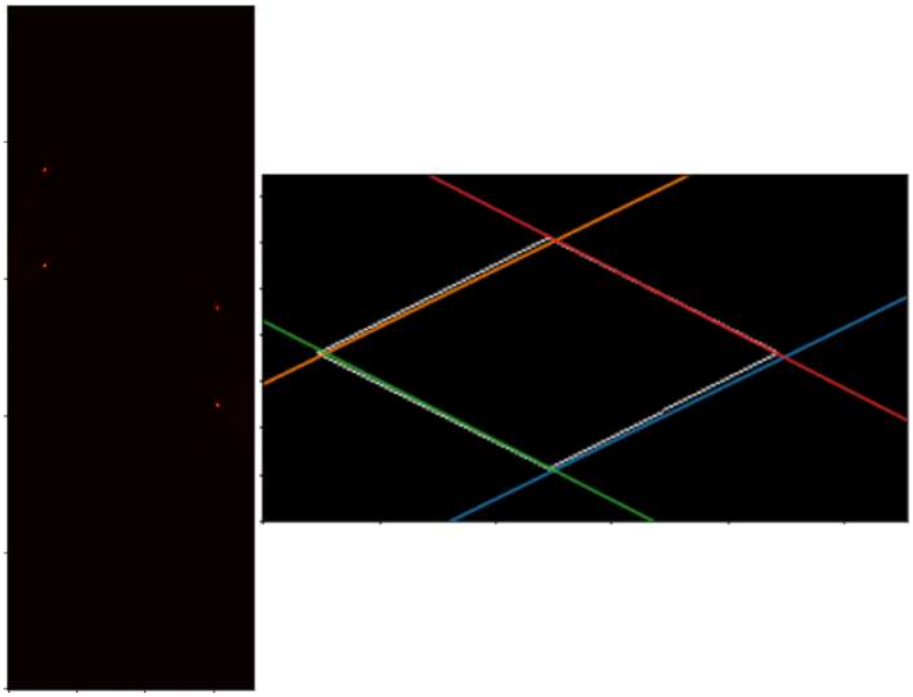

Figure 18: The Hough Space (left) and the corresponding output'

2.2 Finding Parameters of 'hough2.png'

In this section, we have applied the same algorithm to 'hough2.png', which is given in Figure 19

Figure 19: 'hough2.png'

Since this, is a grayscale image, we have applied the Canny Edge Detection algorithm to it in order to lower the computational cost and show the edges that are relevant to the Hough transform. The Canny parameters we used are given as, $(\sigma, t_{low}, t_{high}) = (1, 0.3, 0.5)$. Here, we see that we have used a high value for t_{high} and a lower standard deviation. This is because, we wanted to oversimplify the edge information. Our implementation uses two nested for loops and binary search algorithm to digitize θ and r values, which makes the time complexity,

$$O(\frac{N^2}{2} N \log N) = O(\frac{N^3 \log N}{2}) \tag{21}$$

As this is a slow algorithm, we tried to oversimplify the image as much as we could. The detected edges and its Hough space are presented in Figure 20. The Hough space clearly shows four clusters.

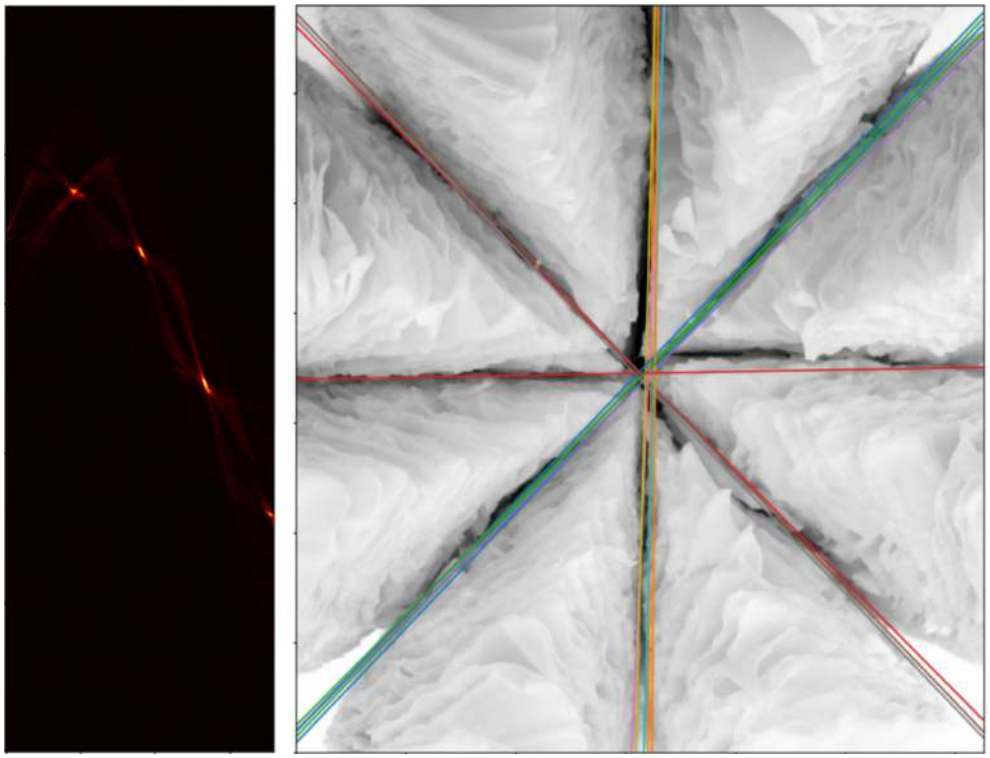

Figure 20: The Hough Space (left) and the corresponding output'

2.3 Finding Parameters of 'edge.png'

This part asked us if we could detect the horizon that is visible in 'edge.png'. Since 'edge.png' is a larger image with many edges, we have altered our algorithm to run faster with a θ approximation. Instead of finding the pairs of points, we have only used one point in each iteration and added its full θ spectrum to the Hough space. Here, we then found r for each θ value as follows, which is essentially identical with Equation 17.

$$r = x\cos\theta + y\sin\theta \tag{22}$$

Code for the altered algorithm)is given below.

```
def houghTransformApprox(image, angle_num=180, r_num=500):
    if len(np.unique(image)) > 2:
        image = feature.canny(image, sigma=1, high_threshold=0.5, low_threshold=0.3)
    x, y = image.shape
    max_r = int(np.ceil(np.sqrt(x**2+y**2)))

    theta_bins = np.linspace(-np.pi/2,np.pi/2, angle_num)
    r_bins = np.linspace(-max_r,max_r, r_num)

    hough_plane = np.zeros((len(r_bins),len(theta_bins)))
    valid_points = np.argwhere(image > 0)
    for pt in valid_points:
        for theta in theta_bins:
            x = pt[0]
            y = pt[1]

            r = x*np.cos(theta) + y*np.sin(theta)

            r_ind = np.searchsorted(r_bins, r, side='left')
            theta_ind = np.searchsorted(theta_bins,theta,side='left')

            if(theta_ind == len(theta_bins)):
                theta_ind -= 1
            if(r_ind == len(r_bins)):
                r_ind -= 1

            hough_plane[r_ind, theta_ind] += 1

    return hough_plane, theta_bins, r_bins
```

Also, to be able to find a more precise line, we have increase the number of angles used to 360 and also applied Canny algorithm on the image with $(\sigma, t_{low}, t_{high}) = (2.8, 0.1, 0.2)$. This time, we used a high standard deviation value to blur the image, such that the edges of the palm tree are negated as much as possible. The edges detected are as follows together with the Hough space in Figure 21. Here, since we used a more speculative approach, the Hough space is more noisy. However, we were still able to find the horizon line which is the center of the bright cluster in the middle of the Hough space. Overall, we were successful with the application of Hough transform on images to find the straight lines.

Figure 21: The Hough Space (left) and the corresponding output'