Homework 3

CS484 - Introduction to Computer Vision

Efe Acer
21602217

Bilkent University, CS

Contents

1 Part 1: Edge Detection

Edges are one of the most useful features of an image. An edge is basically a rapid change in the intensities within an image. In other words, an edge introduces discontinuities in the image intensity function. Characterizing edges, and removing non-edge pixels oftentimes simplify the data and help better identify object boundaries.

In this part of the assignment, our task is to identify the edges in three different images using three edge detection methods. These methods include *Sobel* and *Prewitt* operators, and *Canny* edge detector. The images to work on are grayscale, they are provided below for reference:

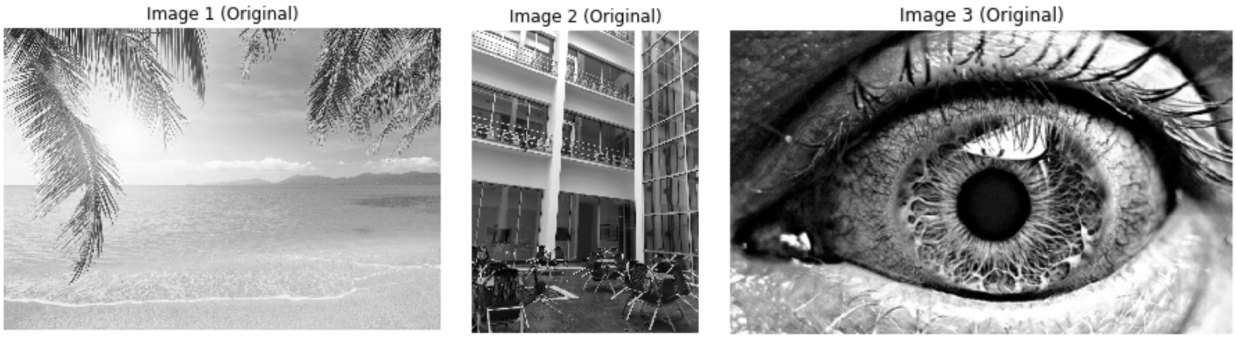

Figure 1: Original Images to be used in Edge Detection

1.1 Edge detection with Sobel and Prewitt operators

Image points of high contrast correspond to edges, such points can be detected by identifying the regions of the image where the rate of change of intensities are the highest. In other words, the derivative of an image, or at least an appropriate analogy, would give us the edges.

Suppose, for now, our intensity function $f(x)$ is 1-dimensional (1D) and continuous. Then the derivative is:

$$f(x) = lim_{\Delta x \to 0} \frac{f(x) - f(x + \Delta x)}{\Delta x} = f_x \tag{1}$$

However, intensity functions are discrete in digital systems. Thus, we use the discrete counterpart of a derivative and approximate f_x by difference equations, below $I[x]$ denotes the discretized intensity function:

$$f_x \approx I[x] - I[x-1] \quad \text{(backward difference)} \tag{2}$$

$$f_x \approx I[x] - I[x+1] \quad \text{(forward difference)} \tag{3}$$

$$f_x \approx (I[x-1] - I[x-1])/2 \quad \text{(central difference)} \tag{4}$$

These approximations are a result of the fact that the smallest value Δx can take in a discrete system is a single pixel unit, which is 1.

These difference equations can be realized by applying 1D cross-correlation to the discrete intensity function by using the filters with appropriate weights. The appropriate filters are $[-1, 1]$, $[1, -1]$, and $[-1, 0, 1]$ respectively. These filters are also known as the *derivative masks*.

Now, we should extend this idea to the 2D world to be able to work with images. In this case, a rapid change of intensity can occur along any vector in the 2D plane. Hence, we use the *gradient vector* to capture the direction in which the maximum change occurs.

$$\nabla f(x,\ y) = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} f_x \\ f_y \end{bmatrix} \tag{5}$$

To estimate f_x in 2D, we take the 8 neighborhood of the pixel located at $(x,\ y)$ and average the central difference results of the three rows. Mathematically:

$$\begin{aligned} f_x \approx \frac{1}{3} \Big(& (I[x+1,\ y+1] - I[x-1,\ y+1])/2 \\ & + (I[x+1,\ y] - I[x-1,\ y])/2 \\ & + (I[x+1,\ y-1] - I[x-1,\ y-1])/2 \Big) \end{aligned} \tag{6}$$

We estimate f_y in a similar manner but this time we apply central difference along the y direction, and average results of the three columns:

$$\begin{aligned} f_y \approx \frac{1}{3} \Big(& (I[x-1,\ y+1] - I[x-1,\ y-1])/2 \\ & + (I[x,\ y+1] - I[x,\ y-1])/2 \\ & + (I[x+1,\ y+1] - I[x+1,\ y-1])/2 \Big) \end{aligned} \tag{7}$$

Again, we can realize the same operations by applying 2D cross-correlation to the discrete intensity function using the appropriate derivative masks. $I_8[x,\ y]$ denotes the intensities of the 8 neighboring pixels located around $(x,\ y)$. The mask used to estimate f_x is denoted as M_x, and the other used to estimate f_y is denoted as M_y. These masks are used in the following way:

$$f_x \approx \frac{M_x * I_8[x, y]}{6} \tag{8}$$

$$f_y \approx \frac{M_y * I_8[x, y]}{6} \tag{9}$$

The operator '$*$' denotes 2D cross-correlation, which means that the mask M is overlaid on the neighborhood around the pixel at $(x,\ y)$, the overlapping pixel values are multiplied and the results are summed together.

After f_x and f_y are estimated according to equations (8) and (9), the magnitude and direction of the gradient vector located at $(x,\ y)$ can be found as follows:

$$|\nabla f(x,\ y)| = \sqrt{f_x^2 + f_y^2} \tag{10}$$

$$\theta = \angle \nabla f(x,\ y) = tan^{-1}\ \frac{f_y}{f_x} \tag{11}$$

The ***Prewitt Operator*** implements equations (8) and (9) by using the ***Prewitt Masks***, which are derived exactly from equations (6) and (7):

Prewitt Masks :

$$M_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad M_y = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Another approach to estimate f_x and f_y, is the ***Sobel Operator***. The only difference it has from the Prewitt Operator is the way ***Sobel Masks*** are defined. Sobel Masks assume that center estimations should be weighted twice as much as the side estimates:

Sobel Masks :

$$M_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad M_y = \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

Computing f_x for the entire image emphasizes the vertical edges, whereas computing f_y emphasizes the horizontal edges. The magnitude of the gradient vector $|\nabla f(x,\ y)|$ is typically used to combine the vertical and horizontal edges together, and the direction of the gradient vector θ captures the texture of the image nicely.

Applying these operators to an image means computing f_x and f_y for all pixels of the image, $\forall\ (x,\ y)$. Overlaying the derivative masks on the boundary pixels requires special care since the masks do not overlap completely with the image at the boundary pixels. The common solution to this problem is to apply *zero-padding* to the image, which means surrounding the image with 0 valued pixels until the masks overlap completely with the image at all boundary pixels.

To implement the Sobel and Prewitt Operators in `Python`, we first write a helper function that performs 2D cross-correlation:

```python
def cross_correlate2D(img, mask):
    """
    Given an image and a mask of arbitrary sizes, performs 2D
    cross-correlation and returns the result.
    Args:
        img: The given image
        mask: The mask to cross-correlate
    Returns:
        result: The result of the 2D cross-correlation
    """
    num_r, num_c = img.shape
    # Compute row and column offsets
    mask_r, mask_c = mask.shape
    r_offset = mask_r // 2 - (mask_r % 2 == 0)
    c_offset = mask_c // 2 - (mask_c % 2 == 0)
    # Apply zero-padding
    padded_img = np.pad(img, ((r_offset, r_offset + (mask_r % 2 == 0)),
                              (c_offset, c_offset + (mask_c % 2 == 0))))
    # Perform 2D cross-correlation
    result = np.zeros((num_r, num_c))
    for row in range(r_offset, r_offset + num_r):
        for col in range(c_offset, c_offset + num_c):
            result[row - r_offset, col - c_offset] = np.sum(
                mask *
                padded_img[row - r_offset: row + r_offset + (mask_r % 2 == 0) + 1,
                           col - c_offset: col + c_offset + (mask_c % 2 == 0) + 1]
            )
    return result
```

Now, we implement two functions that apply Prewitt and Sobel operators to images using the `cross_correlate2D` function:

```python
def prewitt(img):
    """
    Applies the Prewitt Operator to a given image, returns the vertical
    and horizontal edges, combined edges as gradient magnitudes,
    and gradient directions.
    Args:
        img The input image
    Returns:
        result_x: Vertical edges detected by the Prewitt Operator
        result_y: Horizontal edges detected by the Prewitt Operator
        gradient_magnitudes: The combined edges in terms of the gradient
            magnitudes
```

```
        gradient_directions: The gradient directions
    """
    mask_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    mask_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
    result_x = cross_correlate2D(img, mask_x) / 6
    result_y = cross_correlate2D(img, mask_y) / 6
    gradient_magnitudes = np.sqrt(result_x ** 2 + result_y ** 2)
    gradient_magnitudes *= 255 / np.max(gradient_magnitudes)
    gradient_directions = np.arctan2(result_y, result_x)
    return result_x, result_y, gradient_magnitudes, gradient_directions
```

```
def sobel(img):
    """
    Applies the Sobel Operator to a given image, returns the vertical
    and horizontal edges, combined edges as gradient magnitudes,
    and gradient directions.
    Args:
        img The input image
    Returns:
        result_x: Vertical edges detected by the Sobel Operator
        result_y: Horizontal edges detected by the Sobel Operator
        gradient_magnitudes: The combined edges in terms of the gradient
            magnitudes
        gradient_directions: The gradient directions
    """
    mask_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    mask_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    result_x = cross_correlate2D(img, mask_x) / 6
    result_y = cross_correlate2D(img, mask_y) / 6
    gradient_magnitudes = np.sqrt(result_x ** 2 + result_y ** 2)
    gradient_magnitudes *= 255 / np.max(gradient_magnitudes)
    gradient_directions = np.arctan2(result_y, result_x)
    return result_x, result_y, gradient_magnitudes, gradient_directions
```

Using these functions we apply Prewitt and Sobel operators on our original images given in figure 1. The outputs; vertical and horizontal edges detected by the operator, combined edges (gradient magnitudes), and gradient directions are given in the figures provided in the upcoming pages:

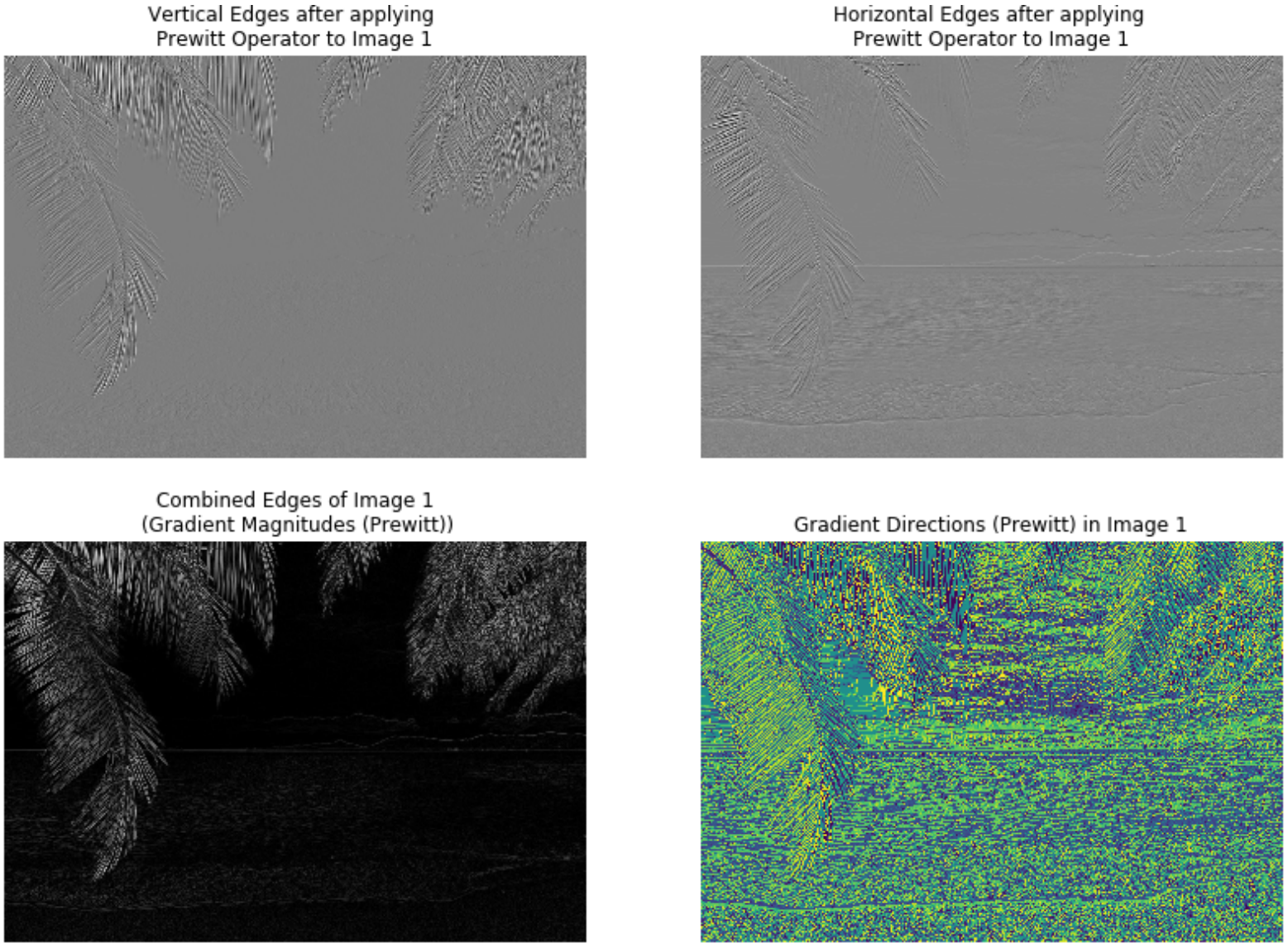

Figure 2: Results after applying Prewitt Operator to Image 1

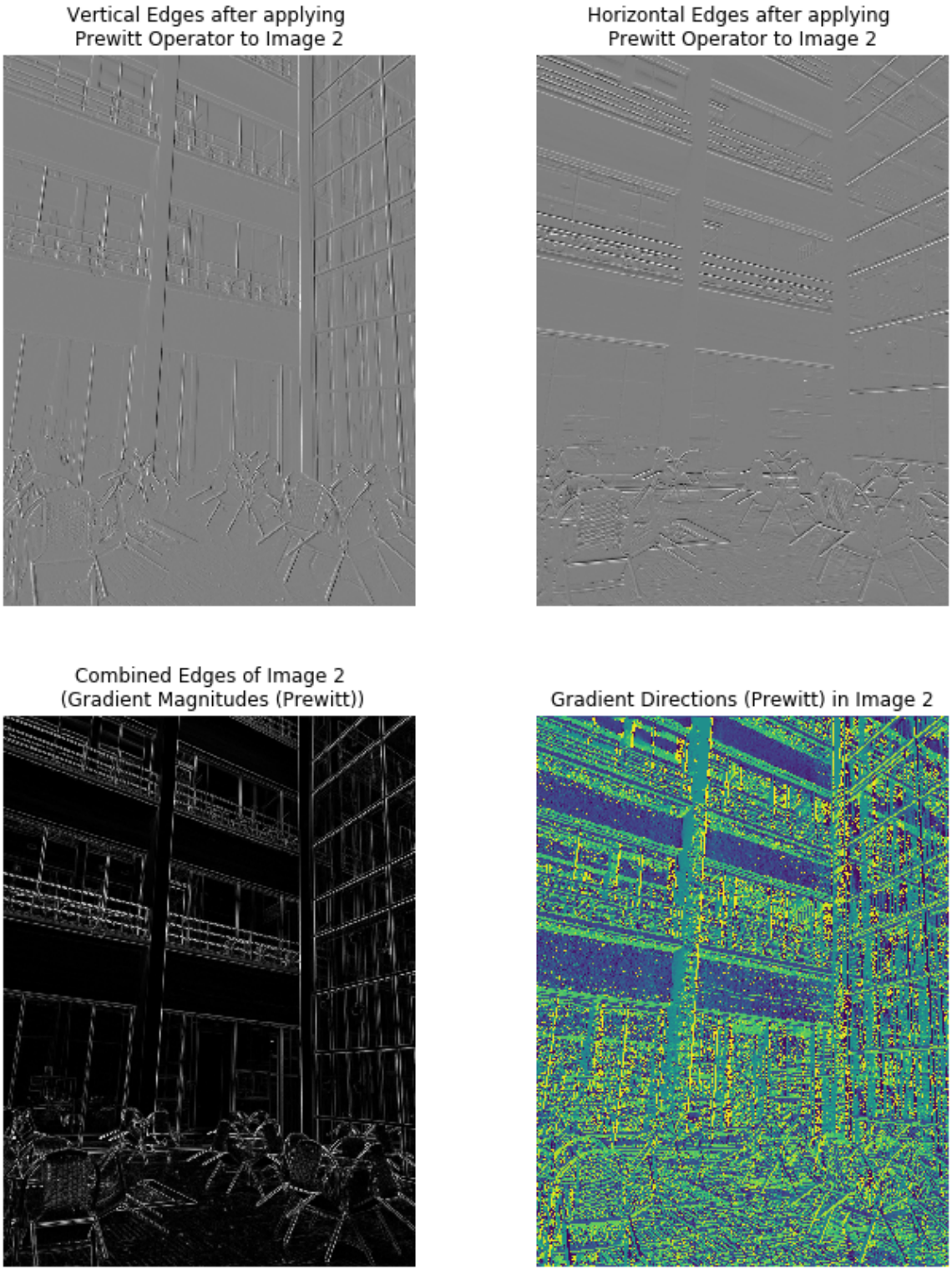

Figure 3: Results after applying Prewitt Operator to Image 2

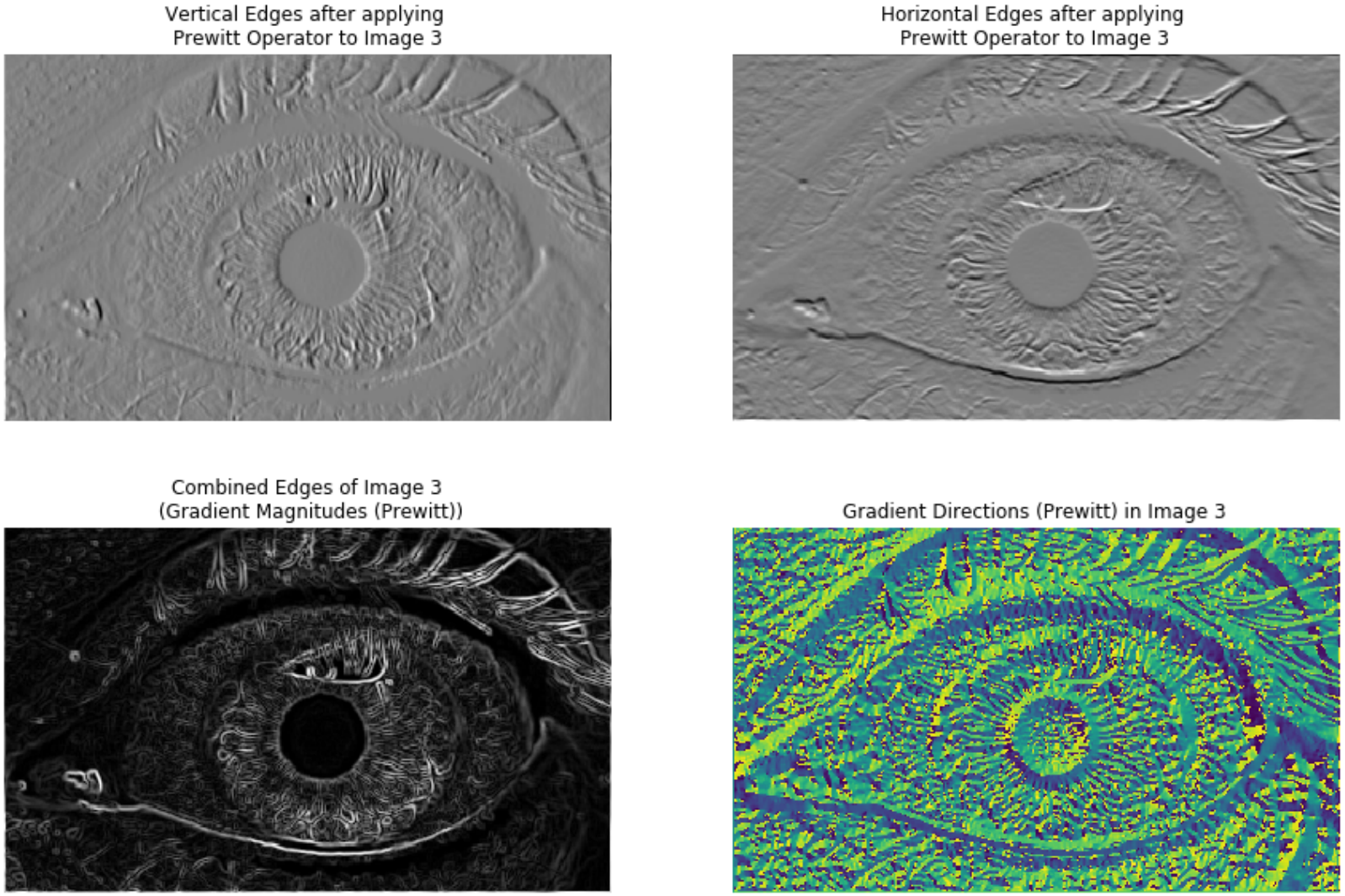

Figure 4: Results after applying Prewitt Operator to Image 3

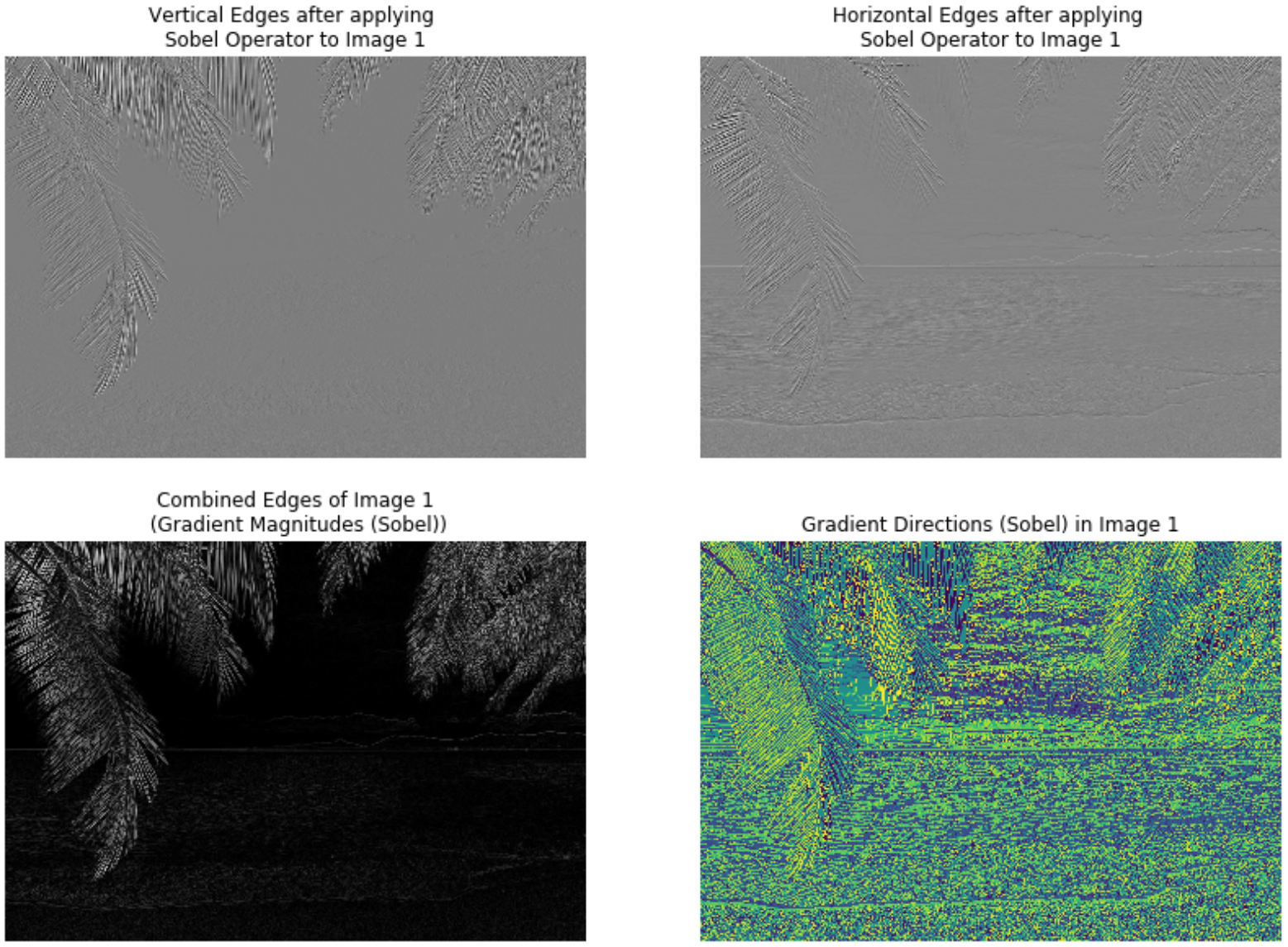

Figure 5: Results after applying Sobel Operator to Image 1

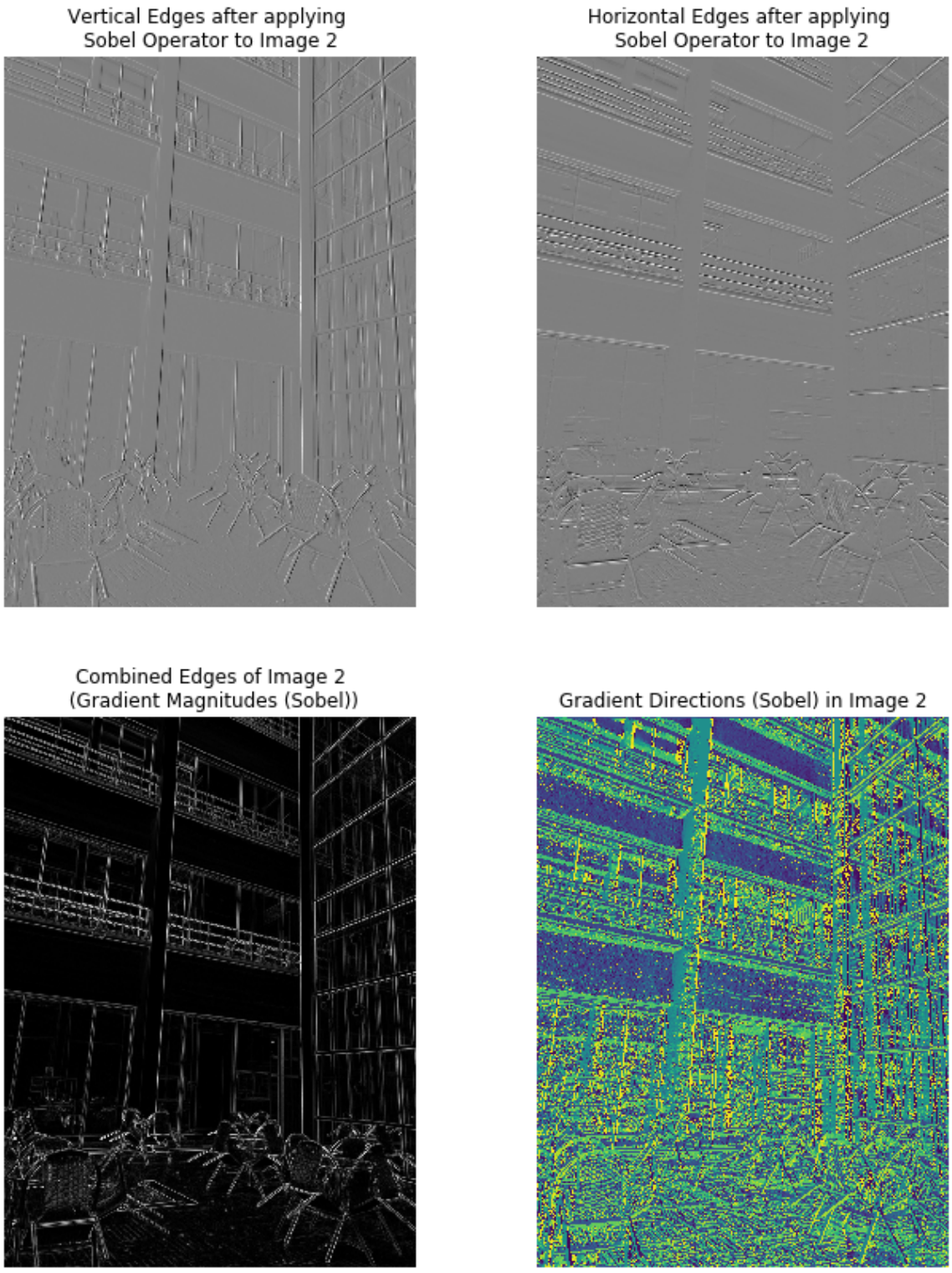

Figure 6: Results after applying Sobel Operator to Image 2

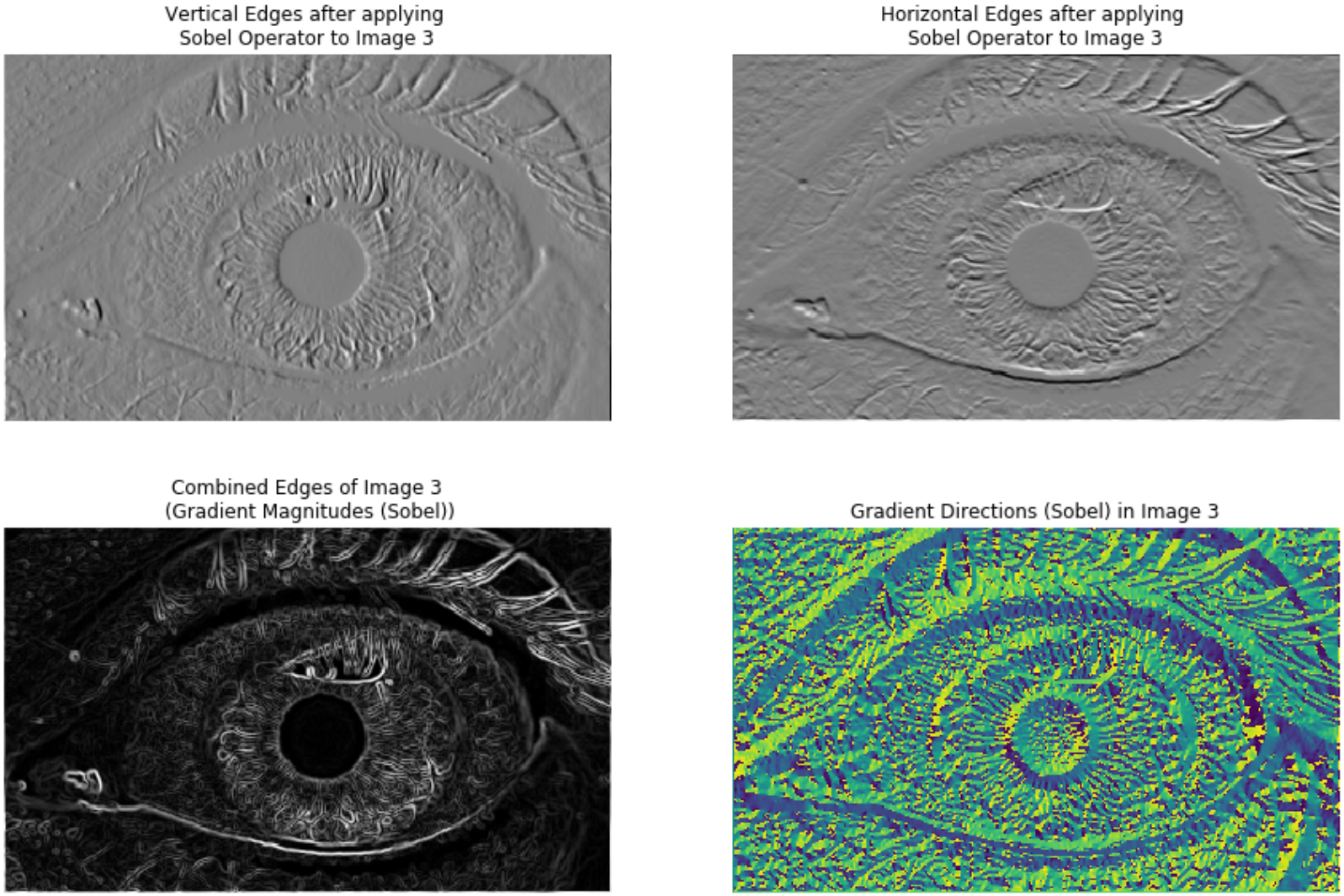

Figure 7: Results after applying Sobel Operator to Image 3

Looking at the results, we can indicate that not all edges are perfectly detected. However, we can clearly say that the results are promising, meaning that we captured many of the high contrast points in all images by both methods. Combined edge results (gradient magnitudes) show a nice overview of the edges for each of the images. Vertical and horizontal edge separation is also done quite successfully. Gradient directions are also useful in the sense that they capture the texture and some of the perspective information present in the original images.

We cannot really see a clear performance difference between the Prewitt and Sobel Operators, in terms of the amount of edges they detect. This is not surprising since the Sobel Operator differs from the Prewitt Operator only in the sense that it values the central pixels more. After careful inspection, one can say that Sobel Operator performs slightly better because it uncovers some of the smaller details.

1.2 Edge detection with Canny edge detector

One of the most widespread methods to detect edges is the *Canny Edge Detector* algorithm. The algorithm consists of five sub-algorithms, which are listed below:

1. Noise Reduction (using the Gaussian Blur)
2. Gradient Calculation (using the Sobel Operator)
3. Non-maximum Suppression
4. Double Threshold
5. Edge Tracking by Hysteresis

1.2.1 Noise Reduction

This stage of the algorithm can be thought as a pre-processing for the Gradient Calculation step. We do not want the noise of the original image to distort the edge detection results by altering the derivatives. Hence, we apply a Gaussian Blur Filter to the original image to make it smoother which helps get rid of the noise.

The Gaussian Blur Filter of shape $(2k+1)$x$(2k+1)$ is defined by the following equation:

$$G_{ij} = \frac{1}{2\pi\sigma^2}\ exp\Big(-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}\Big) \quad \text{for} \ \ 1 \leq i,j \leq (2k+1) \tag{12}$$

A `Python` function that generates a Gaussian Blur Filter of any given size is implemented as follows:

```python
def gaussian_blur(sigma, k=2):
    """
    Generates a Gaussian Blur Filter with shape (2k + 1, 2k + 1), with
    the specified standard deviation.
    Args:
        sigma: The standard deviation of the Gaussian Blur
        k: The parameter that determines the shape of the filter (2 by default)
    Returns:
        G: The resulting Gaussian Blur Filter
    """
    x, y = np.mgrid[-k: k + 1, -k: k + 1]
    G = np.exp(-(x ** 2 + y ** 2) / (2 * sigma ** 2)) / 2 * np.pi * sigma ** 2
    return G
```

Below is a 3D plot that shows a Gaussian Blur Filter of shape 11x11 and standard deviation $\sigma = 1$ for illustration purposes:

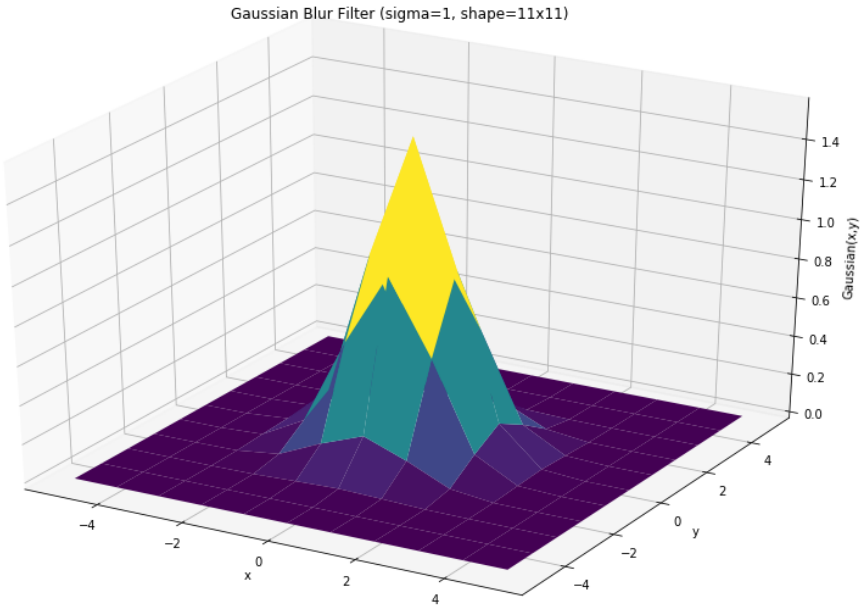

Figure 8: Gaussian Blur Filter of shape 11x11 and standard deviation $\sigma = 1$

As it can be seen above, the Gaussian Blur Filter coefficient diminish as the distance from the filter's center increases, which indeed causes the blur. Central pixels are given the highest weights. The width of the peak is determined by the filter's standard deviation, as σ decreases the peak becomes narrower; this is simply because of the fact that σ is a measure of how probable it is to observe values that are distant from the center.

1.2.2 Gradient Calculation

The Gradient Calculation step is the backbone of the algorithm. In this step, we compute and store the gradient magnitudes and directions, which correspond to edge intensities and orientations. We use Sobel Masks as our derivative masks and calculate the gradient using the Sobel Operator, as described earlier. Edge intensities and orientations are already returned by the `sobel` function as `gradient_magnitudes` and `gradient_directions`, the implementation is given in the previous section. Thus, we do not implement a new function for this step.

1.2.3 Non-Maximum Suppression

The third step is to thin out the detected edges using *Non-Maximum Suppression* algorithm, this helps us obtain clearer boundaries.

The Non-Maximum Suppression algorithm simply goes over all items in the gradient magnitude matrix and removes the items that have a higher intensity neighbor along the gradient direction. The main steps of the algorithm are the following:

- Convert gradient directions (angles) to degrees.
- Identify the edge direction as a horizontal line, diagonal line with positive slope, vertical line, or diagonal line with negative slope; based on the gradient's degree. To be clearer select the line that the gradient angle is closest to.
- Check if the adjacent pixels in the edge direction have a higher intensity than the pixel of interest, if so remove the pixel of interest by setting it to 0.

The figure below helps visualize the steps described above:

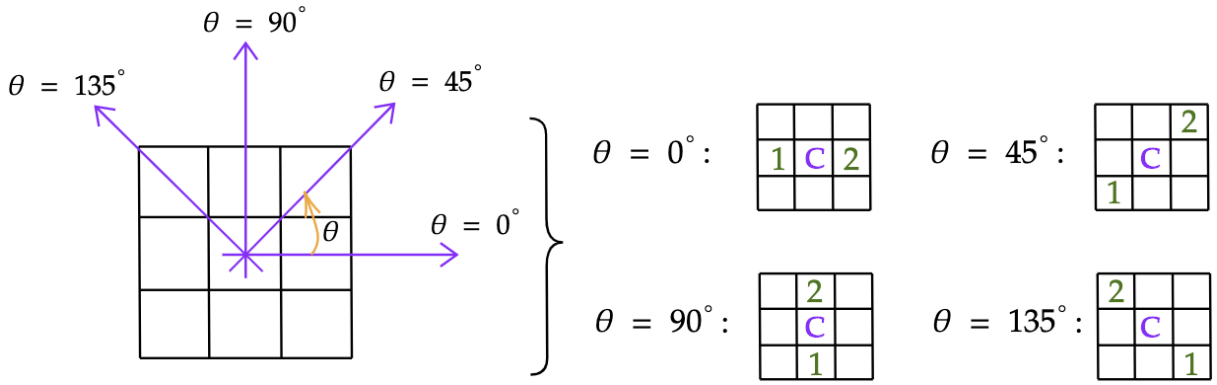

Figure 9: Non-Maximum Suppression Diagram

The implementation of the algorithm follows:

```python
def non_max_suppression(gradient_magnitudes, gradient_directions):
    """
    Implements the Non-Maximum Suppression algorithm to thin out the
    edges described as gradient magnitudes.
    Args:
        gradient_magnitudes: Edge intensities
        gradient_directions: Edge directions (angles in Radians)
    Returns:
        result: The processed edge intensities
    """
    num_rows, num_cols = gradient_magnitudes.shape
    result = np.zeros((num_rows, num_cols), dtype=np.int32)

    # Convert radians to angles
    gradient_angles = gradient_directions * 180 / np.pi
    # To only deal with angles between 0-180
    gradient_angles[gradient_angles < 0] += 180

    for i in range(1, num_rows - 1):
        for j in range(1, num_cols - 1):

            intensity = gradient_magnitudes[i, j] # center's intensity
            angle = gradient_angles[i, j] # edge angle
            intensity1 = 255
            intensity2 = 255

            # angle 0 -> vertical line
            if 0 <= angle < 22.5 or 157.5 <= angle <= 180:
                intensity1 = gradient_magnitudes[i, j + 1]
                intensity2 = gradient_magnitudes[i, j - 1]

            # angle 45 -> diagonal line with positive slope
            elif 22.5 <= angle < 67.5:
                intensity1 = gradient_magnitudes[i + 1, j - 1]
                intensity2 = gradient_magnitudes[i - 1, j + 1]

            # angle 90 -> horizontal line
            elif 67.5 <= angle < 112.5:
                intensity1 = gradient_magnitudes[i + 1, j]
                intensity2 = gradient_magnitudes[i - 1, j]

            # angle 135 -> diagonal line with negative slope
            elif 112.5 <= angle < 157.5:
                intensity1 = gradient_magnitudes[i - 1, j - 1]
```

```
                intensity2 = gradient_magnitudes[i + 1, j + 1]

            if intensity >= intensity1 and intensity >= intensity2:
                result[i, j] = intensity
            else:
                result[i, j] = 0

    return result
```

1.2.4 Double Threshold

The fourth step, *Double Threshold*, aims to categorize each pixel as strong, weak, or non-relevant. The Double Threshold algorithm receives a high threshold ratio, and a low threshold ratio as inputs, then the categories are defined in the following way:

- **Strong**: The pixels whose intensity are greater than or equal to (maximum intensity value) $\times$ (high threshold ratio), we are almost sure that such pixels will contribute to the final result as edges.
- **Weak**: The pixels that are not Strong but have an intensity value greater than or equal to (maximum intensity value) $\times$ (low threshold ratio), such pixels have some potential to contribute to the final result as edges.
- **Non-relevant**: The pixels that are neither Strong nor Weak, such pixels are assumed to have no effect on the final result.

The `Python` implementation of Double Threshold is as follows:

```
def double_threshold(img, ratio_low, ratio_high):
    """
    Categorizes the image pixels as strong, weak or non-relevant
    using double thresholding, returns the thresholded image.
    Args:
        img: The input image
        ratio_low: The low threshold ratio (default is 0.2)
        ratio_high: The high threshold ratio (default is 0.9)
    Returns:
        thresholded_img: The thresholded output
        weak: The intensity level of weak pixels
        strong: The intensity level of strong pixels
    """
    high = ratio_high * np.max(img)
    low = ratio_low * np.max(img)
    weak = 25
```

```
    strong = 255
    thresholded_img = np.zeros(img.shape)
    thresholded_img[np.where(img >= high)] = strong
    thresholded_img[np.where((img >= low) & (img < high))] = weak
    return thresholded_img, weak, strong
```

1.2.5 Edge Tracking by Hysteresis

The last step, *Hysteresis*, combines the Strong and Weak pixels and produces the final result. The rule of combining the pixel types is that a Weak pixel is turned into a Strong one if and only if there exists at least one Strong pixel in its 8-neighborhood, else it is considered Non-relevant. The figure below illustrates the update rule of a Hysteresis step:

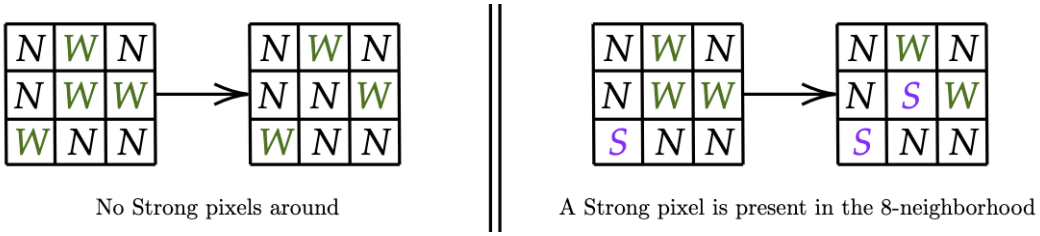

Figure 10: Update Rule of a Hysteresis step

This update rule is applied iteratively to all pixels. The implementation of the algorithm is provided below:

```
def hysteresis(img, weak, strong):
    """
    Combines the Strong and Weak pixels, the rule of combining the pixel types
    is that a Weak pixel is turned into a Strong one if there is at least one
    Strong pixel in its 8-neighborhood, else it is considered Non-relevant.
    Args:
        img: The input image
        weak: The weak intensity level
        strong: The strong intensity level
    Returns:
        result: The processed output image
    """
    num_rows, num_cols = img.shape
    weak_i, weak_j = np.where(img == weak)
    result = img.copy()
    for i, j in zip(weak_i, weak_j):
        if np.any(result[i - 1: i + 2, j - 1: j + 2] == strong):
            result[i, j] = strong
```

```
19          else:
20              result[i, j] = 0
21      return result
```

1.2.6 Combining the Steps

Now, it is time for us to combine the five steps in a single algorithm. The following diagram shows how the methods above are combined to implement the `canny_edge_detector` function:

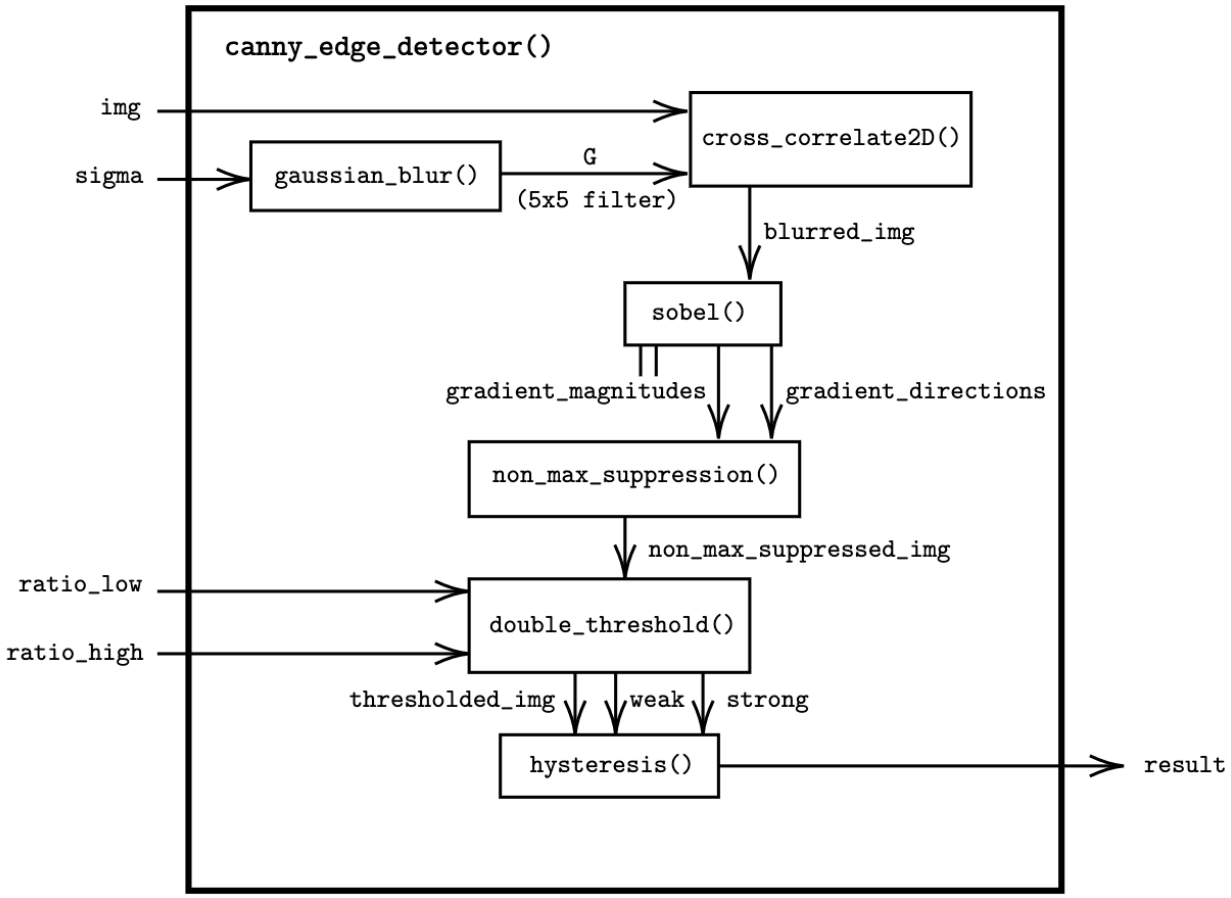

Figure 11: Function Diagram for Canny Edge Detector

The implementation itself follows:

```
def canny_edge_detector(img, sigma, ratio_low, ratio_high, k=2):
    """
    Applys Canny Edge Detection algorithm to a given image.
    Args:
        img: The input image
        sigma: The standard deviation of the 5x5 Gaussian Blur Filter
            used in the first step of the algorithm (default is 1)
        ratio_low: The low threshold ratio used in the Double Thresholding step
            (default is 0.02)
        ratio_high: The high threshold ratio used in the Double Thresholding step
            (default is 0.09)
        k: Parameter to adjust the shape of the (2k + 1, 2k + 1) gaussian blur
    """
    G = gaussian_blur(sigma, k)
    blurred_img = cross_correlate2D(img, G)
    _, _, gradient_magnitudes, gradient_directions = sobel(blurred_img)
    non_max_suppressed_img = non_max_suppression(gradient_magnitudes,
        gradient_directions)
    thresholded_img, weak, strong = double_threshold(non_max_suppressed_img,
        ratio_low, ratio_high)
    result = hysteresis(thresholded_img, weak, strong)
    return result
```

1.2.7 Results

Using our own implementation `canny_edge_detector`, we process the images given in figure 1. For each image we use four different combinations of function parameters (`sigma`, `ratio_low`, and `ratio_high`). Note that the parameters `ratio_low` and `ratio_high` are not the thresholds itself but they serve as ratios that define the thresholds, this design choice makes the parameters independent of the image of interest. The results are provided in the upcoming pages:

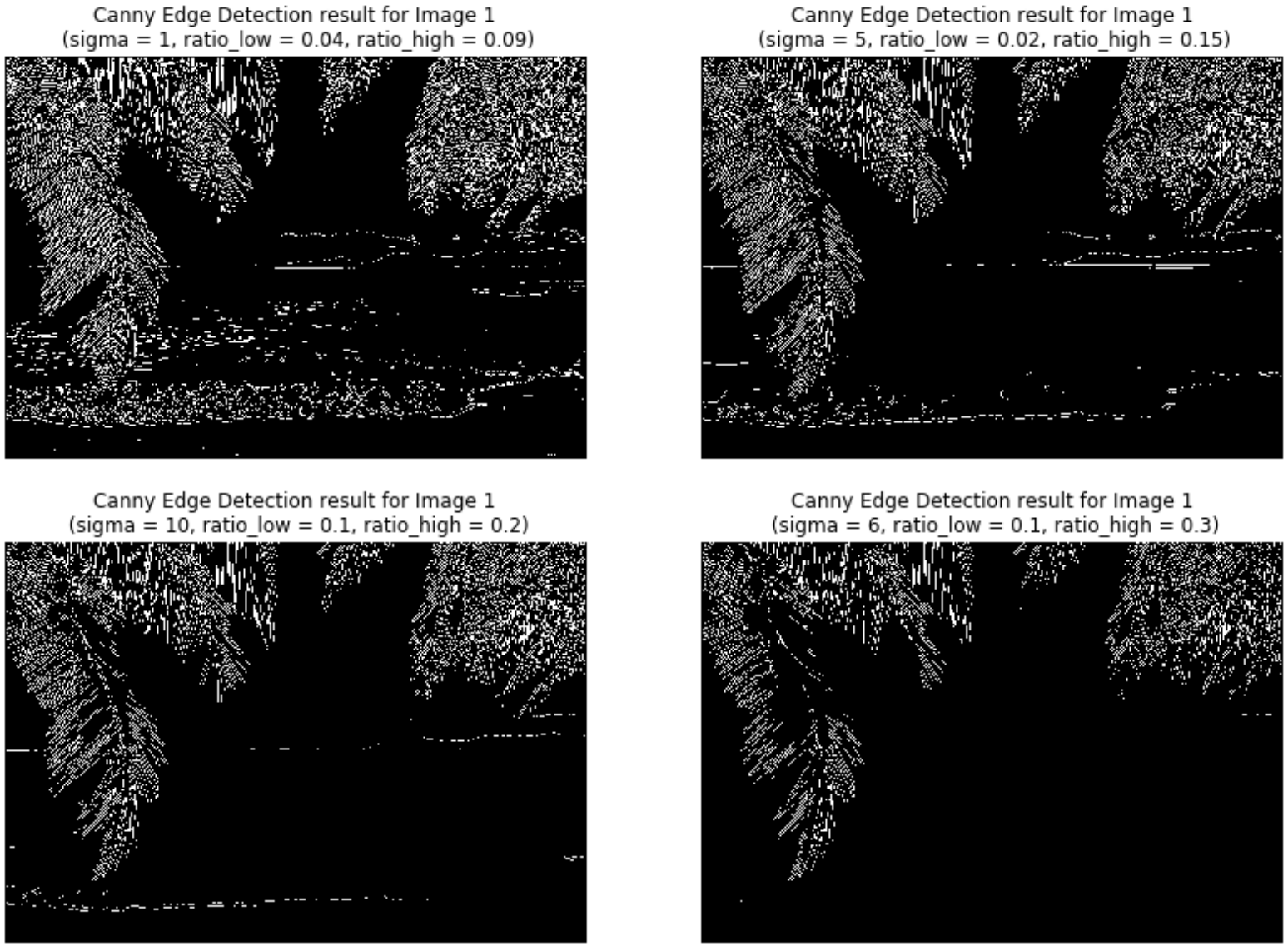

Figure 12: Results after applying Canny Edge Detector to Image 1

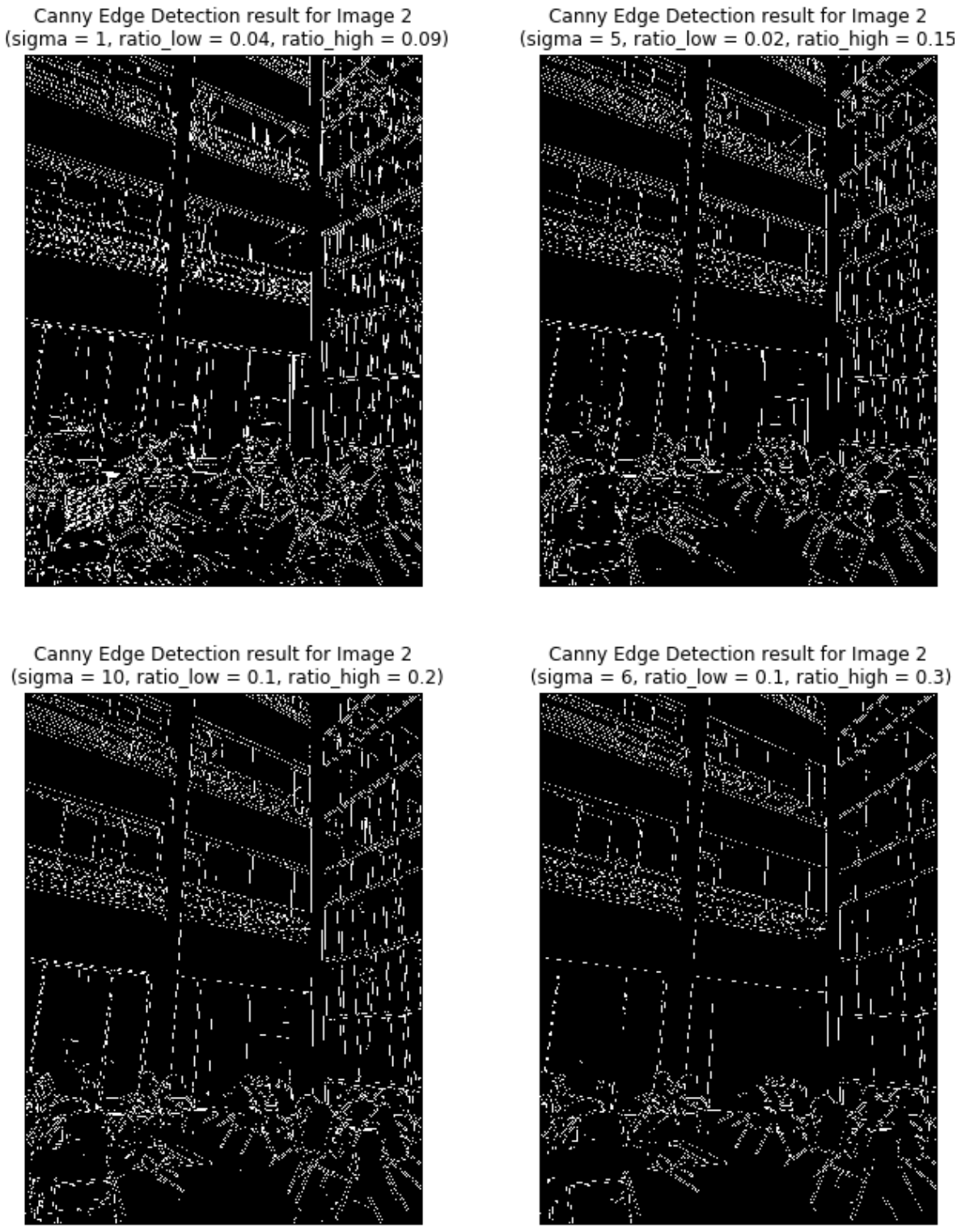

Figure 13: Results after applying Canny Edge Detector to Image 2

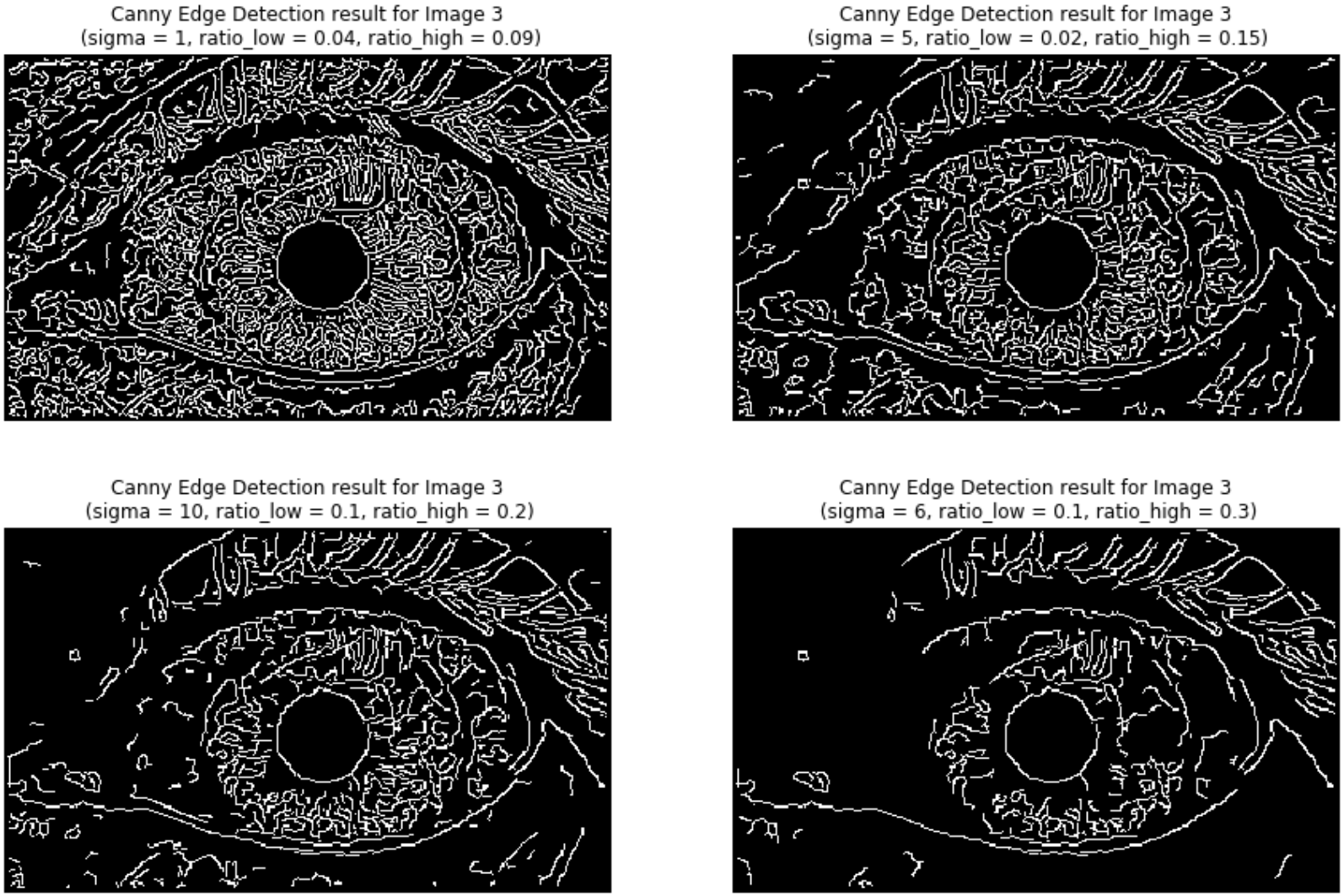

Figure 14: Results after applying Canny Edge Detector to Image 3

The optimal function parameters are listed in the table below. Note that optimality is determined by inspection, in other words the binary image that looks as if it captures most of the edges is said to be optimal.

Image Number	`sigma` (σ)	`ratio_low`	`ratio_high`
1	**5**	**0.02**	**0.15**
2	**10**	**0.1**	**0.2**
3	**10**	**0.1**	**0.2**

- The best parameters for the first image are chosen to be `sigma` = 5, `ratio_low` = 0.02, `ratio_high` = 0.15, since these parameters detect most of the coastline (intersection of the sea and the land), horizon and leaves.
- For the second image the best parameters are chosen to be `sigma` = 10, `ratio_low` = 0.1, `ratio_high` = 0.2, because this set of parameters do not detect the unnecessary details but capture the overall geometry of the space including the shapes of the chairs.
- For the third image, the best parameters are also `sigma` = 10, `ratio_low` = 0.1, `ratio_high` = 0.2, these parameters detect the boundary of the eye, the pupil, and the eyelashes quite well. However iris contents are misdetected as edges, this is an inevitable result since iris contents are full of sudden color changes.

Note that a large σ value leads to more blur in the Noise Reduction stage. Increasing σ too much may lead to the removal of image features in addition to the removal of noise. A small `ratio_low` increases the number of Weak pixels and thus increases the number of pixels processed in the Hysteresis stage, and a large `ratio_high` decreases the number of Strong pixels which in turn leads to the detection of less edge points.

Considering that the outputs of the Canny Edge Detector are binary images, we can say that this method outperforms Sobel and Prewitt Operators. Outputs of those operators are grayscale images that require thresholding (single) to be compared with the outputs of the Canny Edge Detector. However, thresholding will definitely remove most of the weak pixels without even considering them as potential contributions to the final edges, when applied to Sobel and Prewitt outputs.

In conclusion, Canny Edge Detector adds many pre-processing and post-processing steps to the Gradient Calculation, whereas Sobel and Prewitt Operators only perform Gradient Calculation. Thus, it is not a surprise that Canny Edge Detector works better than Sobel and Prewitt Operators.

2 Part 2: Edge Linking with Hough Transform

Hough Transform is a technique to extract particular features of an image. The technique requires the features to be representable in a certain parametric form, hence Hough Transform is typically used to detect geometric shapes such as lines and circles. In this part, we will focus on detecting lines with Hough Transform.

The starting point of Hough Transform is the following line representation:

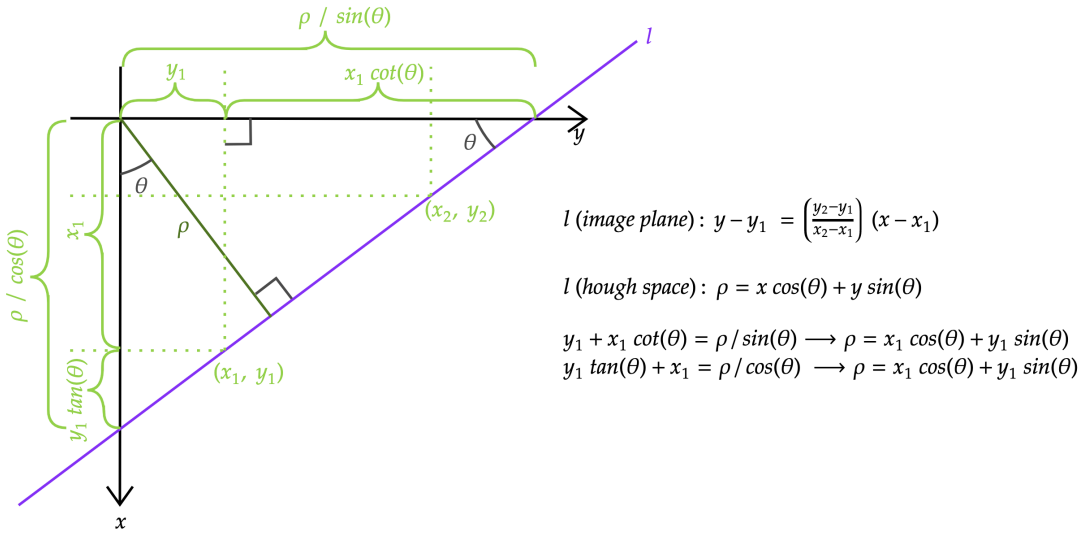

Figure 15: Line Representation in Hough Space

Any line l in the image plane can be represented in Hough Space by the parameters ρ and θ as described in figure 15.

Given a binary image where nonzero valued pixels represent the edge points, the Hough Line Detection algorithm traverses each edge point and votes for each $(p,\ \theta)$ combination that defines a line passing from the processed edge point. Here, "voting" means incrementing the value of the $(p,\ \theta)$ point in the Hough Space in order to emphasize that the line defined by this point intersects with a certain number of edge points.

We know that an infinite amount of lines pass from a single edge point, but this is not a problem since we discretize the ρ and θ axes of the Hough Space when programming the algorithm. θ axis of the Hough Space is limited to $(0°,\ 180°)$ and ρ axis is limitied to $\pm$(longest distance from the origin to an edge point); this way all edge points become representable in the discretized Hough Space. The discretized Hough Space is also called the *accumulator*, and we will refer to it by this word when programming.

After each edge point is processed, a vote threshold can be applied to the Hough Space to extract the "best" lines. Here, "best" means the lines that received more votes than the threshold which indicates that those lines intersect with a significant number of edge points.

It is oftentimes hard to imagine what the algorithm is doing after reading a description. Thus, we present a diagram representing an instance of the voting:

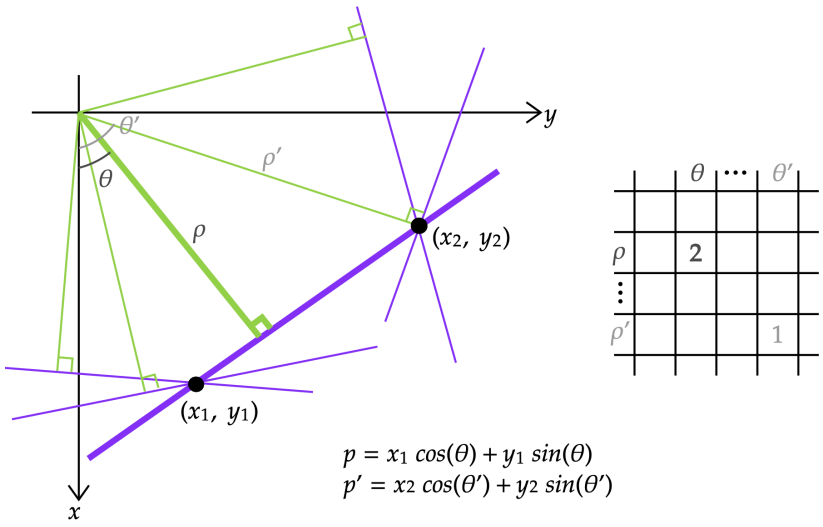

Figure 16: Voting Diagram

The left side of figure 16 represents the image plane, whereas the right side represents the discretized Hough Space or the accumulator. We see that a line receives more votes as it intersects with more edge points.

The `Python` implementation of the Hough Line Detection algorithm is given below:

```
1  def hough_line_detection(binary_img, step_rho, step_theta):
2      """
3      Performs line detection on a binary edge image using Hough transform.
4      Args:
5          binary_img: The given binary edge image
6          step_rho: The distance between discretized rho samples
7          step_theta: The distance between discretized theta samples
8      Returns:
9          accumulator: The image in the discretized Hough Space
10         rhos: The discretized rho values
11         thetas: The discretized theta values
12     """
13     # Get the nonzero pixel indices
14     y_idxs, x_idxs = np.nonzero(binary_img)
15
16     # Find maximum possible value for rho
17     hypots = np.sqrt(x_idxs ** 2 + y_idxs ** 2)
18     rho_max = np.ceil(np.max(hypots))
19
```

```
    # Discretize rhos and thetas
    rhos = np.arange(-rho_max, rho_max + step_rho, step_rho)
    thetas = np.deg2rad(np.arange(0, 180, step_theta))

    # Cache results for performance
    cos_vals = np.cos(thetas)
    sin_vals = np.sin(thetas)
    num_thetas = len(thetas)
    theta_idxs = np.arange(0, num_thetas)

    # Perform voting
    accumulator = np.zeros((len(rhos), num_thetas))
    for x, y in zip(x_idxs, y_idxs):
        rho_idxs = ((x * cos_vals + y * sin_vals + rho_max) // step_rho).astype(
            int)
        accumulator[rho_idxs, theta_idxs] += 1

    return accumulator, rhos, thetas
```

The following method typically runs after `hough_line_detection` to find the image plane parameters $(x_1,\ y_1), (x_2,\ y_2)$ of the lines that are voted above a threshold. The method implements basic trigonometry to map the lines from Hough Space back to the image plane, the logic can be understood by inspecting figure 15.

```
def get_best_lines(accumulator, rhos, thetas, threshold):
    """
    Given an image in hough space, and the rho and theta axes finds the
    image plane parameters for the lines that are voted above a threshold.
    Args:
        accumulator: The image in the discretized Hough Space
        rhos: The discretized rho values
        thetas: The discretized theta values
    Returns:
        best_lines: The image plane parameters of the most-voted lines
    """
    rho_idx, theta_idx = np.where(accumulator >= threshold)
    rho_best = rhos[rho_idx]
    theta_best = thetas[theta_idx]
    best_lines = []
    for rho, theta in zip(rho_best, theta_best):
        cos_theta = np.cos(theta)
        sin_theta = np.sin(theta)
        x0 = rho * cos_theta
        y0 = rho * sin_theta
        x1 = int(x0 - 1000 * sin_theta)
```

```
        y1 = int(y0 + 1000 * cos_theta)
        x2 = int(x0 + 1000 * sin_theta)
        y2 = int(y0 - 1000 * cos_theta)
        best_lines.append((x1, y1, x2, y2))
    return best_lines
```

The above implementations are first tested using the binary image of a noisy parallelogram. The original image is provided below:

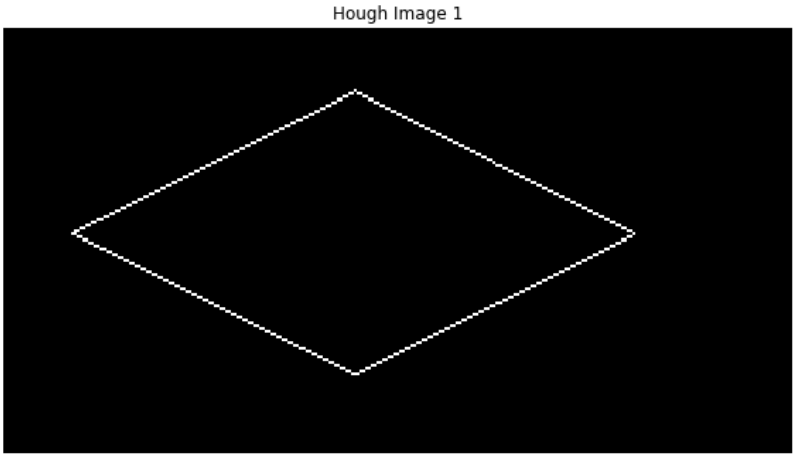

Figure 17: The Binary Image of a Noisy Parallelogram

After this image is fed into the `hough_line_detection` as an input (with parameters `step_rho = 0.5` and `step_theta = 0.125`) and image plane parameters of the most-voted lines are found using `get_best_lines`, we plot the obtained lines on top of the original image:

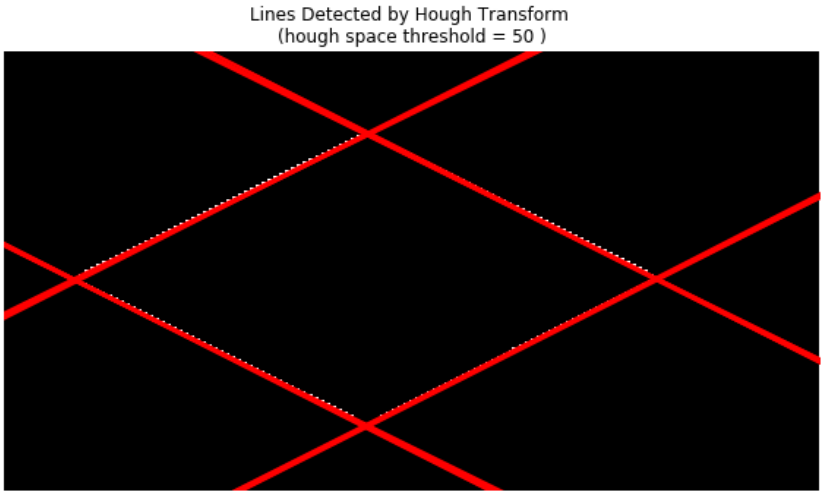

Figure 18: Lines detected by Hough Transform (Parallelogram)

We also plot the accumulator corresponding to the parallelogram:

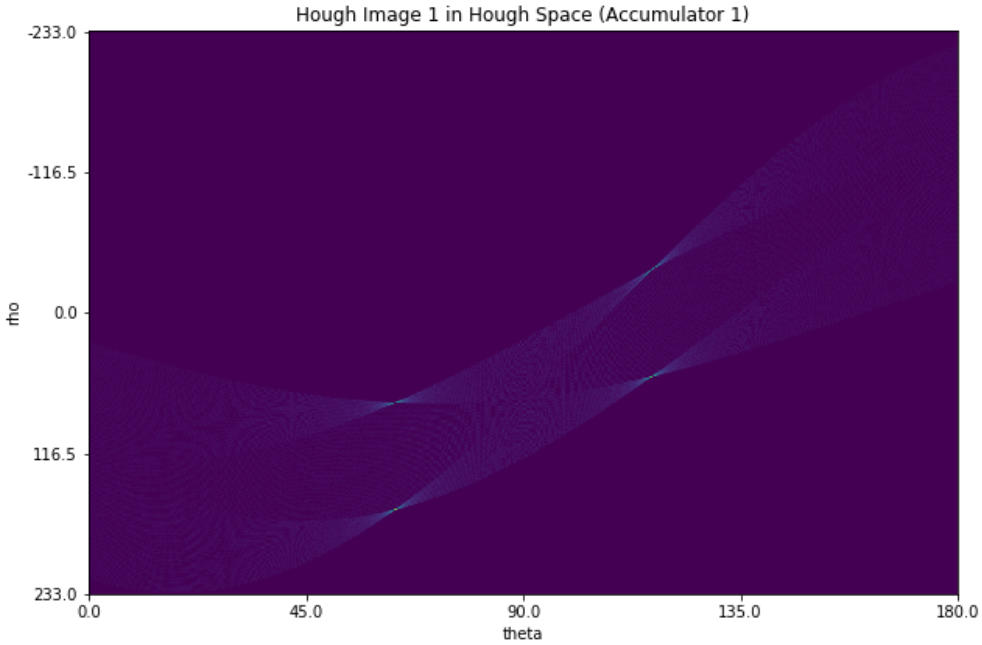

Figure 19: Accumulator Corresponding to the Parallelogram

Note that the votes coming from each edge point form sinusoids in the accumulator and these sinusoids cumulate at the Hough Space points corresponding to the best lines. We can say that there are four main lines in the image plane by only seeing that we have four places in the Hough Space where the sinusoids cumulate.

Now, we use two new images of the Turkish desert Baklava, and repeat the same procedure for both. The first input Baklava and its edges detected by the Canny Edge Detector (`sigma = 5`, `ratio_low = 0.15`, `ratio_high = 0.5`) are given below:

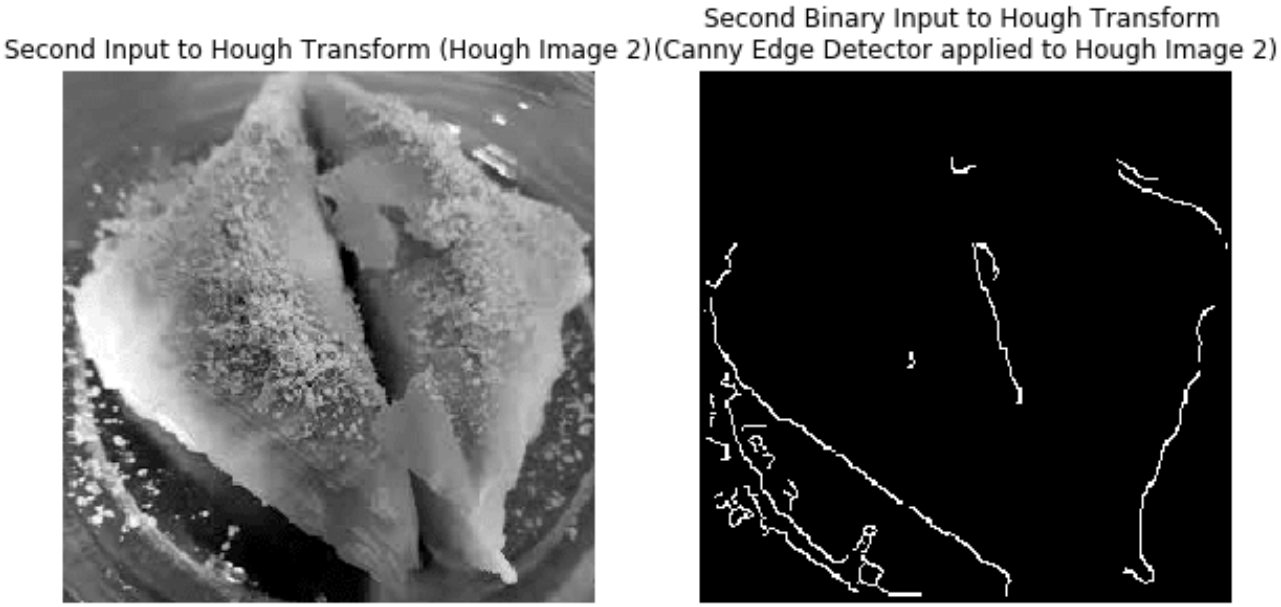

Figure 20: The first Baklava Input and its Edges

We call `hough_line_detection` with the edges above and the parameters `step_rho = 0.5` and `step_theta = 0.125`. The result follows:

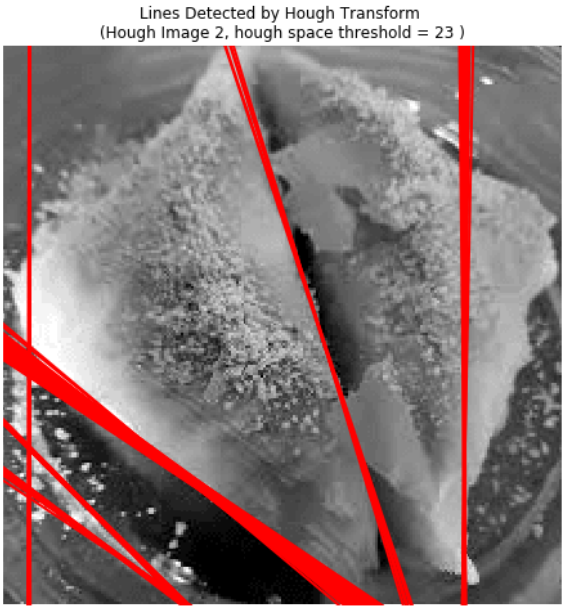

Figure 21: Lines detected by Hough Transform (First Baklava)

The output is not as expected. This is primarily because of the pistachio particles on top of the Baklava. Those particles deceive the Canny Edge Detector and act as if they are edges.

We tune the parameters to get rid of the particle but then the real edge information gets corrupted.

The accumulator of the first Baklava image is the following:

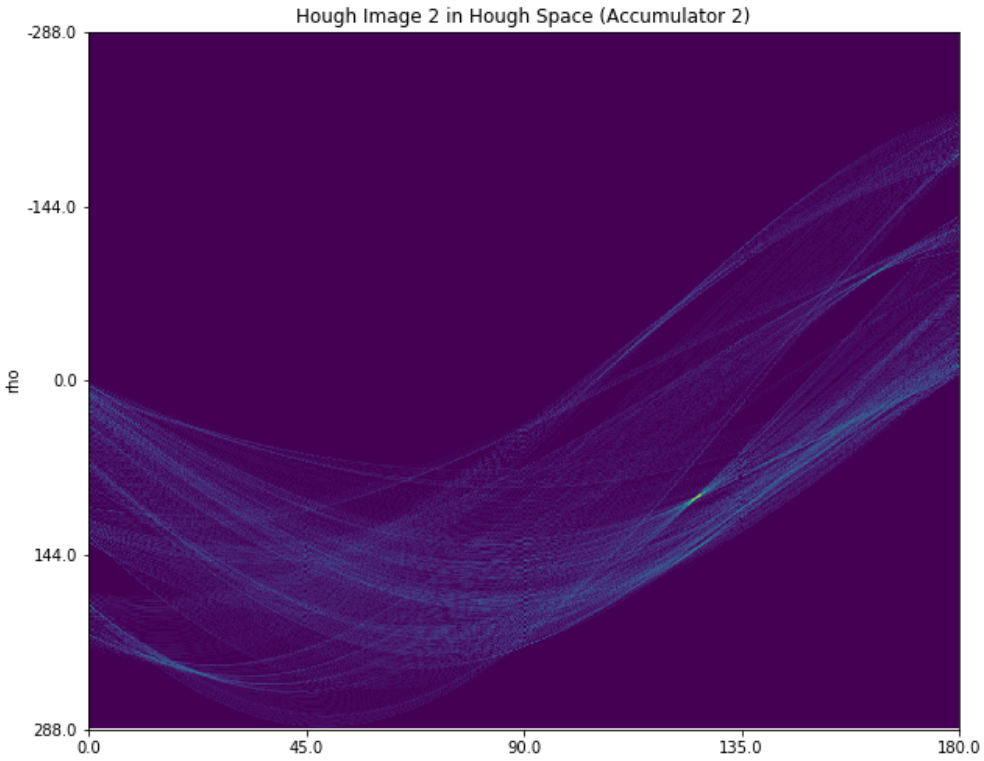

Figure 22: Accumulator Corresponding to the First Baklava

We proceed to the second Baklava image. The second input Baklava and its edges detected by the Canny Edge Detector (`sigma = 10`, `k = 10` `ratio_low = 0.05`, `ratio_high = 0.15`) are as follows:

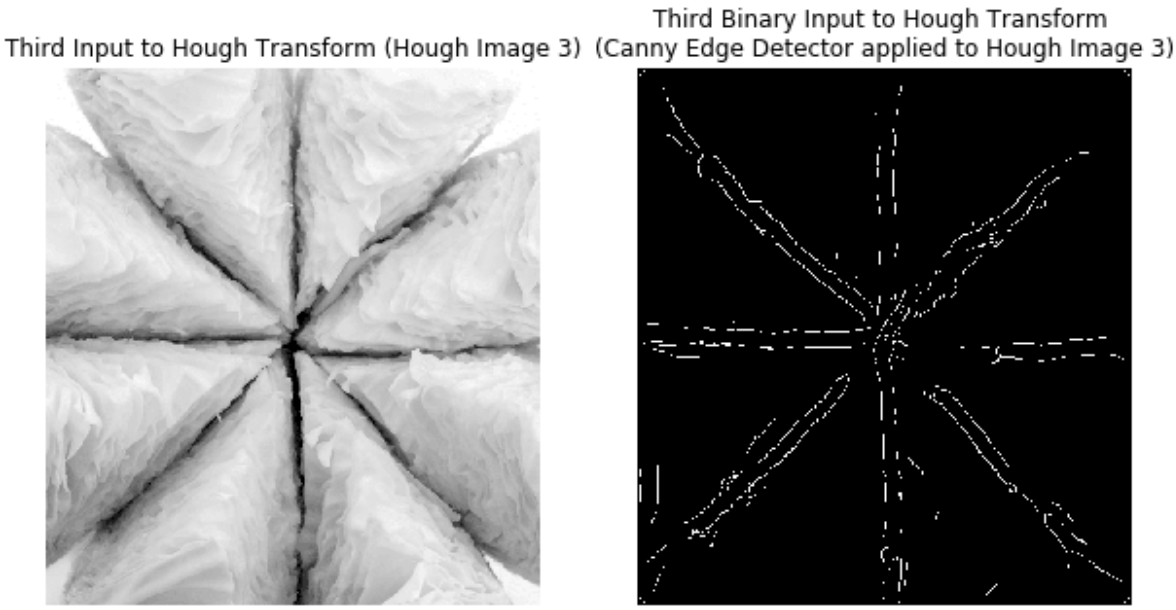

Figure 23: The second Baklava Input and its Edges

We call `hough_line_detection` with the edges above and the parameters `step_rho = 1` and `step_theta = 0.25`. The result follows:

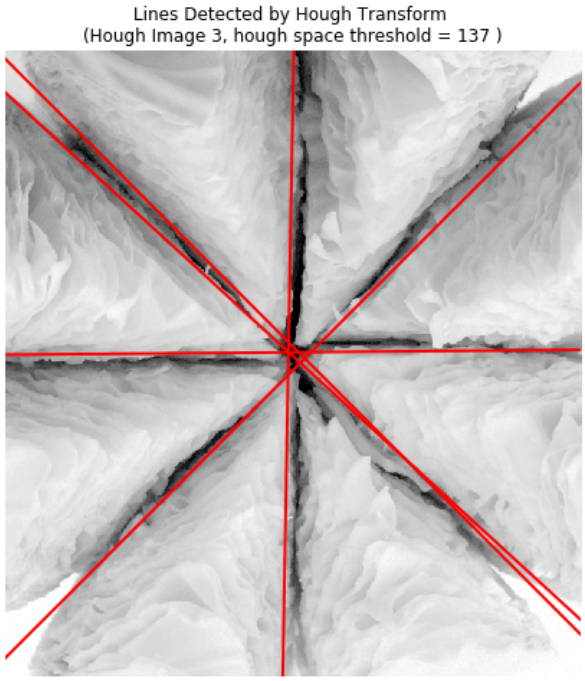

Figure 24: Lines detected by Hough Transform (Second Baklava)

As it can be seen, the edge lines are properly detected in this image, this is particularly because the second Baklava image is orderly in terms of its gradient and has a well defined geometric pattern.

The accumulator for the second Baklava is:

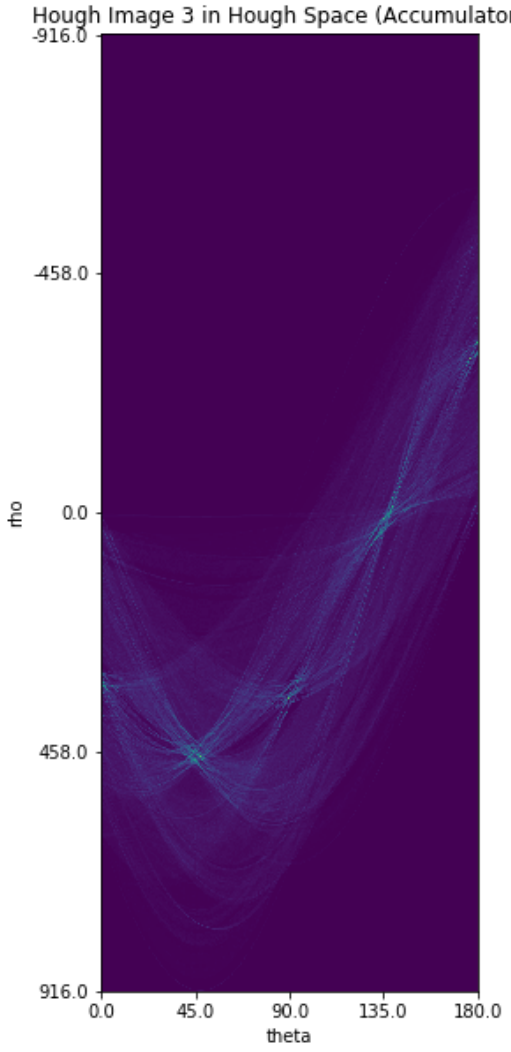

Figure 25: Accumulator Corresponding to the Second Baklava

Now we use the top-right image in figure 12 and apply Hough Line Detection to the edges of the coast image with the aim of detecting the horizon line. We call `hough_line_detection` with parameters `step_rho = 1` and `step_theta = 0.1`. The result follows.

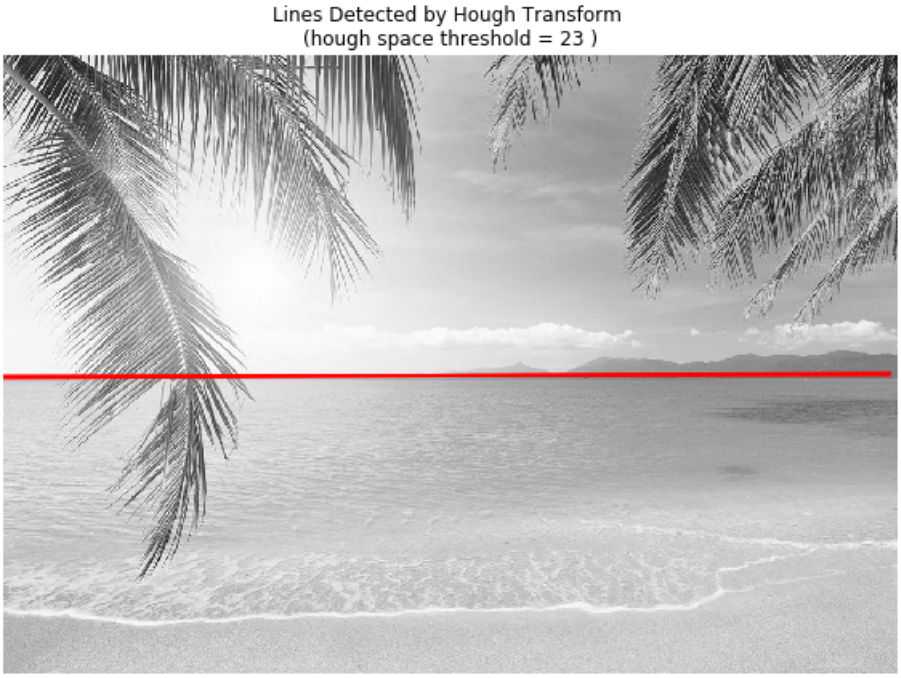

Figure 26: Lines detected by Hough Transform (Coast Image)

Getting the horizon line was not straightforward, it required some time to tune the parameters. However, given the correct edge output and the correct step sizes; our implementation succeeded to find the horizon line. The accumulator of the coast image is given in the next page:

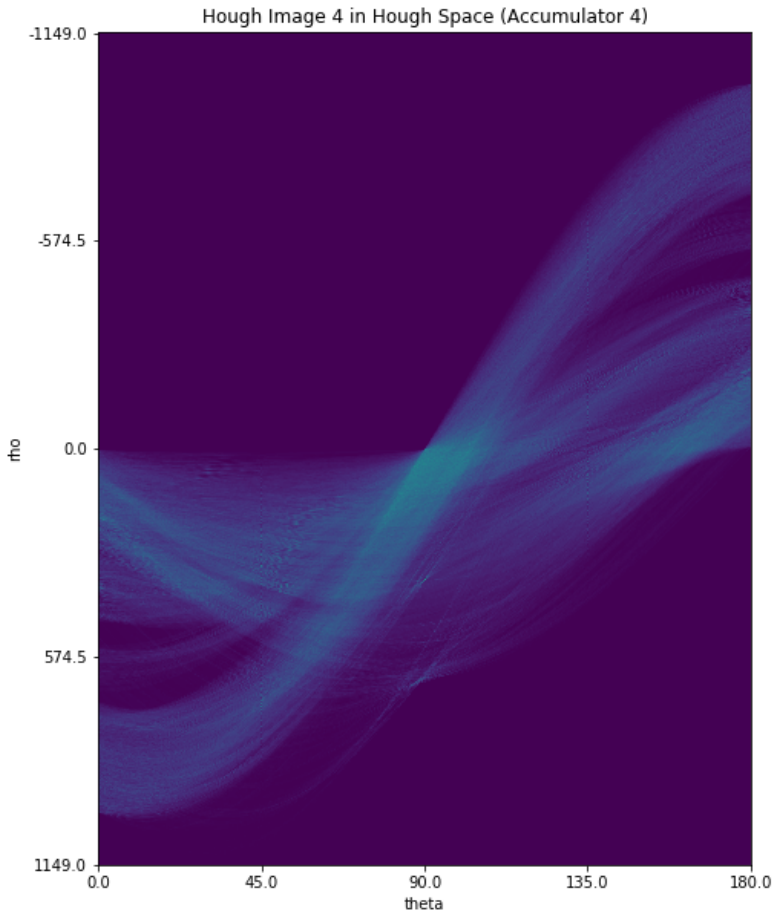

Figure 27: Accumulator Corresponding to the Coast Image

To sum up, Hough Transform comes in handy in detecting geometric features when a good binary edge image is given. However, computing a good binary edge image requires nontrivial works such as tuning edge detector parameters, applying pre-processing and post-processing steps, etc.. Provided that the quality of the binary edge image is not so good, we should not trust the results of Hough Line Detection since it tends to generate a significant amount of false positives in that case.