CV Lab 1

Youssef Hossam Aboelwafa	20012263
Ahmed Samir Sayed	20010102
Mohamed Raffeek Mohamed	20011574

Part I: Applying Image Processing Filters For Image Cartoonifying

1.1 Generating a black-and-white sketch

Convert the image to gray scale instead of BGR returned from **cv.imread()**

```
gray_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

1.1.1 Noise Reduction Using Median Filter

Blurring before edge detection is essential for the following reasons:

1. Noise Reduction: Blurring reduces noise, which can create false edges and interfere with the accuracy of edge detection.
2. Edge Preservation: Median blur, in particular, preserves edges while removing noise, making it ideal for preprocessing before edge detection.
3. Improved Accuracy: With reduced noise, edge detection algorithms can more accurately identify true edges, leading to better results.

```
median_filtered_img = cv.medianBlur(gray_img, 13)
```

1.1.2 Edge Detection Using Laplacian Filter

Laplacian Edge Detection:
Detects edges by calculating the second derivative of the image intensity. Highlights regions of rapid intensity change, corresponding to edges. cv.Laplacian(median_filtered_img, cv.CV_8U, ksize=5) computes the Laplacian of the median-filtered grayscale image with a kernel size of 5.

Dilation:
Enhances the detected edges by thickening them. cv.dilate(laplacian, kernel=(1, 1), iterations=1) applies dilation to the Laplacian edge-detected image, although with a (1, 1) kernel and 1 iteration, it has minimal effect.

```
laplacian = cv.Laplacian(median_filtered_img, cv.CV_8U, ksize=5)
laplacian = cv.dilate(laplacian, kernel=(1, 1), iterations=1)
```

1.1.3 Binary Thresholding and convert back to 3 channels

Thresholding:
Converts the Laplacian edge-detected image to a binary image.
Sets pixel values above 127 to 0 and below 127 to 255, creating a binary inverse image.

Color Conversion:
Converts the binary image from grayscale to BGR color space for a later overlay with the painting image.

```
_, thresholded_img = cv.threshold(laplacian, 127, 255,
                    cv.THRESH_BINARY_INV)
                    thresholded_img_BGR =
cv.cvtColor(thresholded_img,cv.COLOR_GRAY2BGR)
```

1.2 Generating a color painting and a cartoon

The loop applies bilateral filtering to the image 15 times, which helps in:

Smoothing the image while preserving edges.
Reducing noise and enhancing the cartoon-like effect.

```
for i in range(15):
    painting =
cv.bilateralFilter(painting,d=-1,sigmaColor=10,sigmaSpace=10)
```

Overlaying the edge mask with the painting image

The bitwise AND operation is used to combine the smoothed image with the edge-detected image, resulting in a cartoon-like effect. The edges are highlighted, and the regions are smoothed, creating a visually appealing cartoon image.

Edge Emphasis: The edges detected in the thresholded image are combined with the smoothed regions of the painting image.
Cartoon Effect: The resulting image has distinct edges and smooth regions, characteristic of a cartoon.

```
cartoon = cv.bitwise_and(painting, thresholded_img_BGR)
```

The Final Function Combining all steps:

```python
def cartoonize_image(img):
    gray_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    median_filtered_img = cv.medianBlur(gray_img, 13)
    laplacian = cv.Laplacian(median_filtered_img, cv.CV_8U, ksize=5)
    laplacian = cv.dilate(laplacian, kernel=(1, 1), iterations=1)
    _, thresholded_img = cv.threshold(laplacian, 127, 255, cv.THRESH_BINARY_INV)
    thresholded_img_BGR = cv.cvtColor(thresholded_img, cv.COLOR_GRAY2BGR)
    painting = img
    for _ in range(15):
        painting = cv.bilateralFilter(painting, d=-1, sigmaColor=10, sigmaSpace=10)
    cartoon = cv.bitwise_and(painting, thresholded_img_BGR)
    return cartoon
```

1.3 Examples:

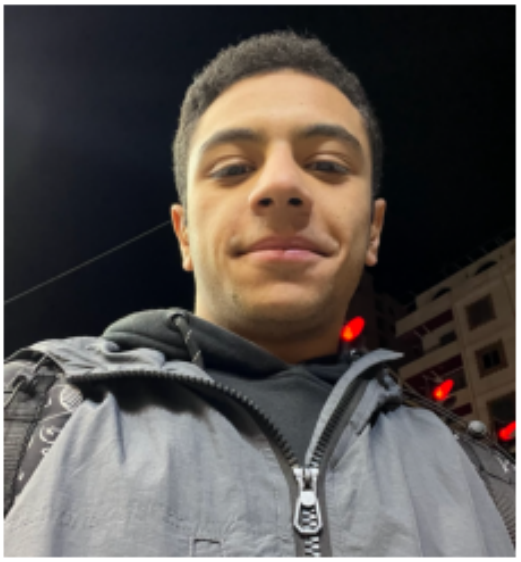
Original Image

Cartoon Image

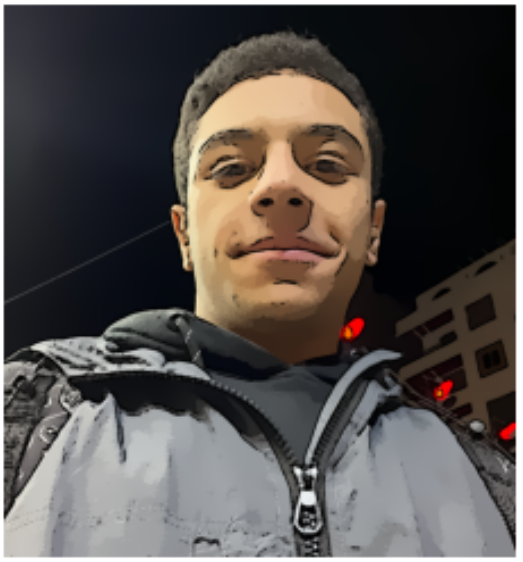
Original Image

Cartoon Image

Original Image



Cartoon Image



Original Image

Cartoon Image

Original Image

Cartoon Image

Original Image

Cartoon Image

Original Image

Cartoon Image

Part II:

Road Lane Detection Using Hough Transform

Original Image

2.2.1 Smoothing the image

First, the median_smoothing function applies a median blur a 2-dimensional median smoothing filter to the image to reduce noise, which is a common preprocessing step in image processing.

```
def median_smoothing(img, kernel_size):
    return cv.medianBlur(img, kernel_size)
img = median_smoothing(img, 5)
show_image(img, 'Median Smoothing')
```

✓ 0.0s

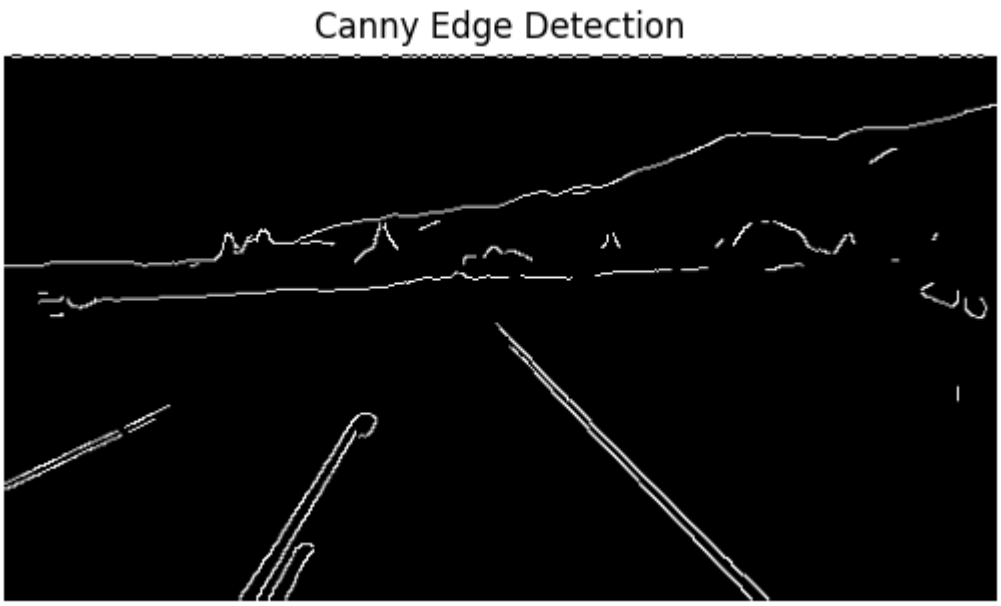
Median Smoothing

2.2.2 Edge Detection

After smoothing the image, edge detection is performed using the canny_edge_detection function, which applies the Canny edge detection algorithm to the image. We use relatively high values for thresholding to remove most of the noise.

```
def canny_edge_detection(img, low_threshold, high_threshold):
    return cv.Canny(img, low_threshold, high_threshold)
edges = canny_edge_detection(img, 150, 200)
show_image(edges, 'Canny Edge Detection')
```

✓ 0.8s



2.3 Region Of Interest

The output of the edge detection contains unnecessary edges that belongs to objects outside the road. Therefore, to eliminate this noise, define a polygon (region) of the image to mask the noise edges producing only an edge image, as shown in the figure, that contains the region of interest that focus on the road. To focus on the region of interest (ROI) where lanes are likely to be, a mask is created using the region_of_interest function. This function applies a polygonal mask to the image, keeping only the region defined by the vertices and discarding the rest.

The region of interest function takes the lane image and the polygon mask that determine the ROI where the lane exist and mask all the channels for the input image using `bitwise_and` function from opencv.

```
def region_of_interest(img, vertices):
    mask = np.zeros_like(img)

    if len(img.shape) > 2:
        channel_count = img.shape[2]
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255
    cv.fillPoly(mask, vertices, ignore_mask_color)
    masked_image = cv.bitwise_and(img, mask)
    return masked_image
```

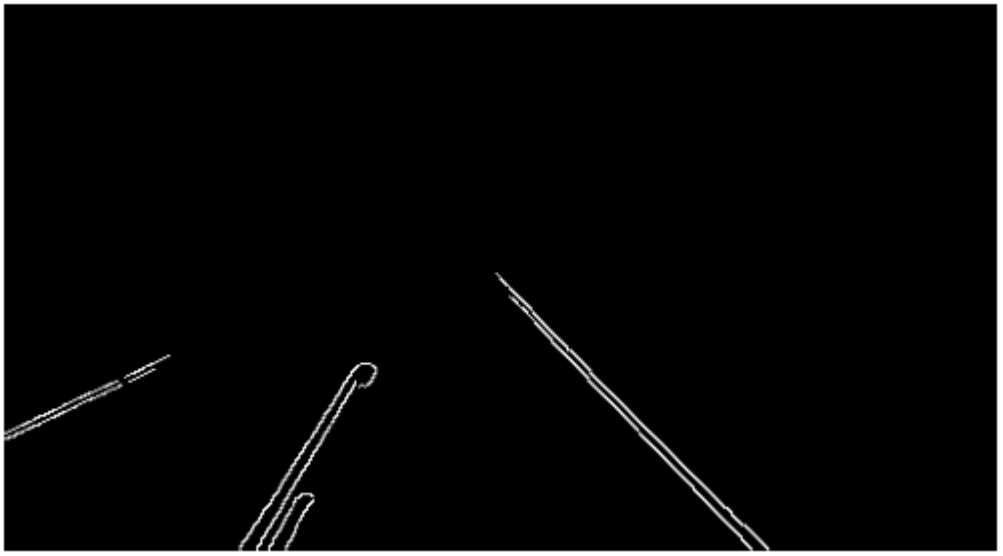
```
    height, width = edges.shape
    print('Height:', height, 'Width:', width)
    vertices = np.array([[
            (0, height),
            (int(0.0 * width), int(0.5* height)),
            (int(0.6 * width), int(0.42 * height)),
            (int(0.8 * width), height)
        ]], dtype=np.int32)
    mask = np.zeros_like(edges)
    cv.fillPoly(mask, vertices, 255)
    show_image(mask, 'Mask')
```

Mask

```
roi_edges = region_of_interest(edges, vertices)
show_image(roi_edges, 'Region of Interest')
```

✓ 0.1s

Region of Interest



2.3.1 Accumulation into (ρ, θ)-space using Hough transform

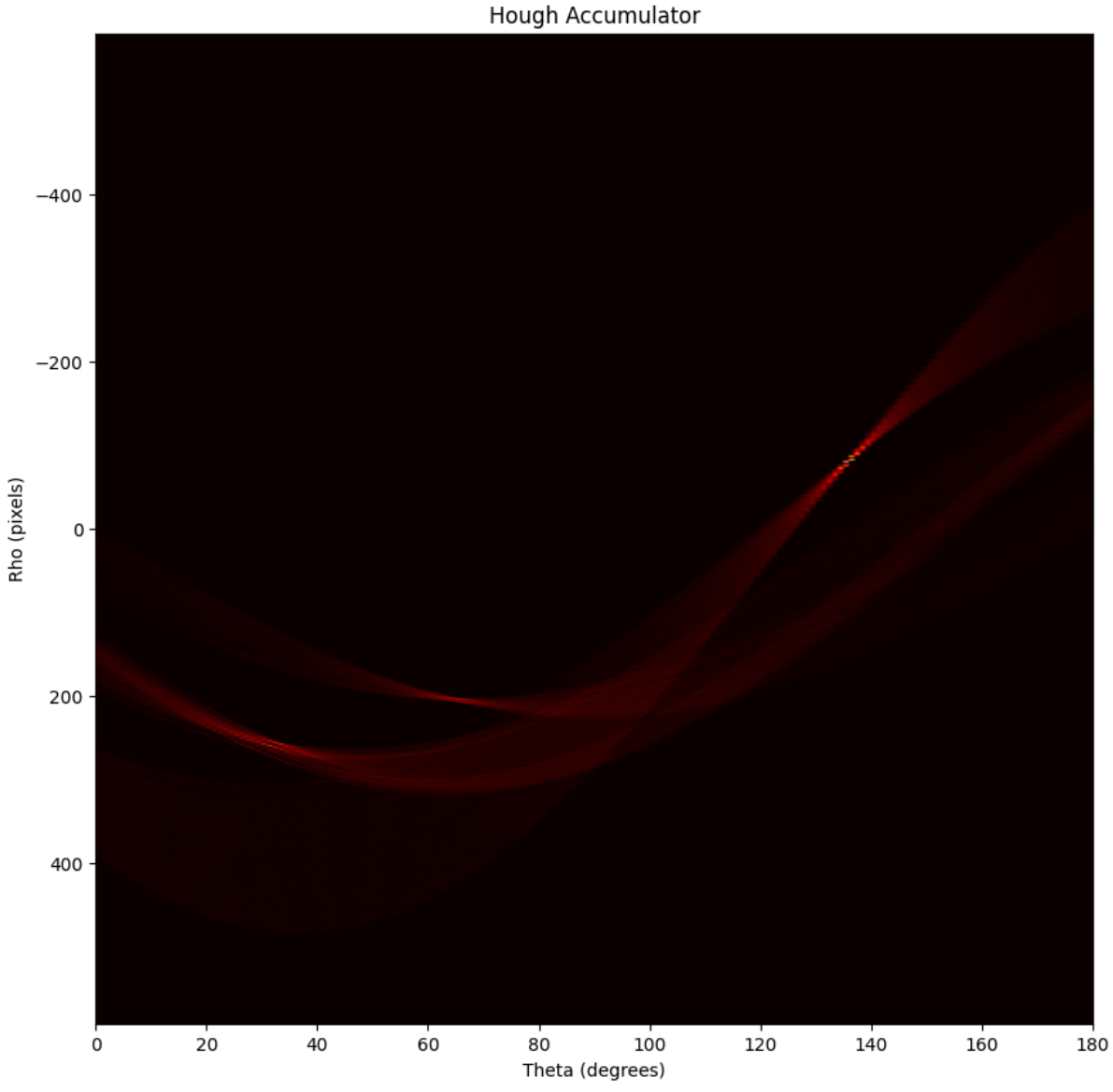
The Hough Transform is implemented in the `hough_transform` function to detect lines in the edge-detected image.

The function takes the ROI edge image and pins for the ρ and θ in the accumulator matrix then iterate over each edge point in the image to calculate the sinsudial wave of it, defined using ρ and θ.

```python
def hough_transform(edge_img, p_bin=1, theta_bin=1):

    height, width = edge_img.shape
    thetas = np.deg2rad(np.arange(0, 180, theta_bin))
    diag_len = int(np.ceil(np.hypot(height, width)))
    ps = np.arange(-diag_len, diag_len, p_bin)
    accumulator = np.zeros((len(ps), len(thetas)), dtype=np.uint64)
    y_idxs, x_idxs = np.nonzero(edge_img)
    for i in range(len(x_idxs)):
        x = x_idxs[i]
        y = y_idxs[i]
        for t_idx, theta in enumerate(thetas):
            p = int(round(x * np.cos(theta) + y * np.sin(theta))) + diag_len
            accumulator[p, t_idx] += 1
    return accumulator, thetas, ps



accumulator, thetas, ps = hough_transform(roi_edges, p_bin=1, theta_bin=1)
```

The accumulator array, which stores the results of the Hough Transform, is visualized using a heatmap.

```
plt.figure(figsize=(10, 10))
plt.title("Hough Accumulator")
plt.xlabel("Theta (degrees)")
plt.ylabel("Rho (pixels)")
plt.imshow(accumulator, cmap="hot", aspect='auto',
           extent=[0, 180, ps[-1], ps[0]])
plt.show()
```



2.3.2 Refining Coordinates and HT Post-Processing

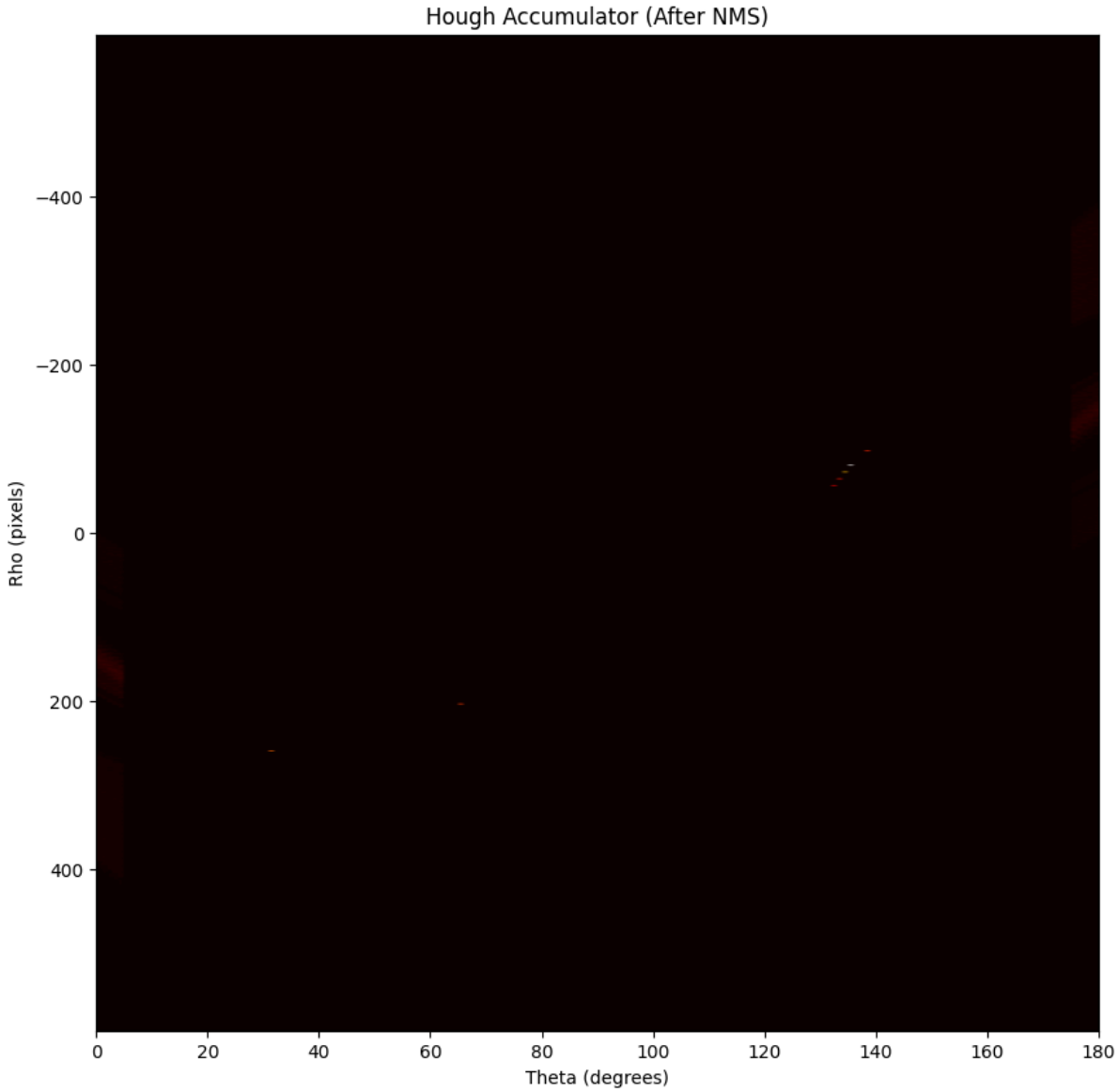
During the whole process of finding the parameters, some inaccuracies could occur. This could be due to choosing a large bin size for HT or due to noise in the detected edges. Therefore, after finding the parameters, a search in the (ρ, θ)-space is executed. We look for the highest peaks of the accumulator function and perform non-maximum suppression for lower values.

Non-maximum suppression is applied to the accumulator array using the `non_maximum_suppression` function to filter out weaker lines and retain the strongest ones.

It search within a window of neighborhood and certain threshold to maintain the real peeks lines and suppress the noise.

```python
def non_maximum_suppression(acc, neighborhood_size=3, threshold=10):
    suppressed = np.copy(acc)
    half_size = neighborhood_size // 2
    for i in range(half_size, acc.shape[0] - half_size):
        for j in range(half_size, acc.shape[1] - half_size):
            window = acc[i - half_size : i + half_size + 1,
                         j - half_size : j + half_size + 1]
            if acc[i, j] != np.max(window) or acc[i, j] < threshold:
                suppressed[i, j] = 0
    return suppressed

acc_nms = non_maximum_suppression(accumulator, neighborhood_size=10, threshold=50)
```

```
plt.figure(figsize=(10, 10))
plt.title("Hough Accumulator (After NMS)")
plt.xlabel("Theta (degrees)")
plt.ylabel("Rho (pixels)")
plt.imshow(acc_nms, cmap="hot", aspect='auto',
           extent=[0, 180, ps[-1], ps[0]])
plt.show()
```



2.3.3 Find the peeks in the NMS accumulator

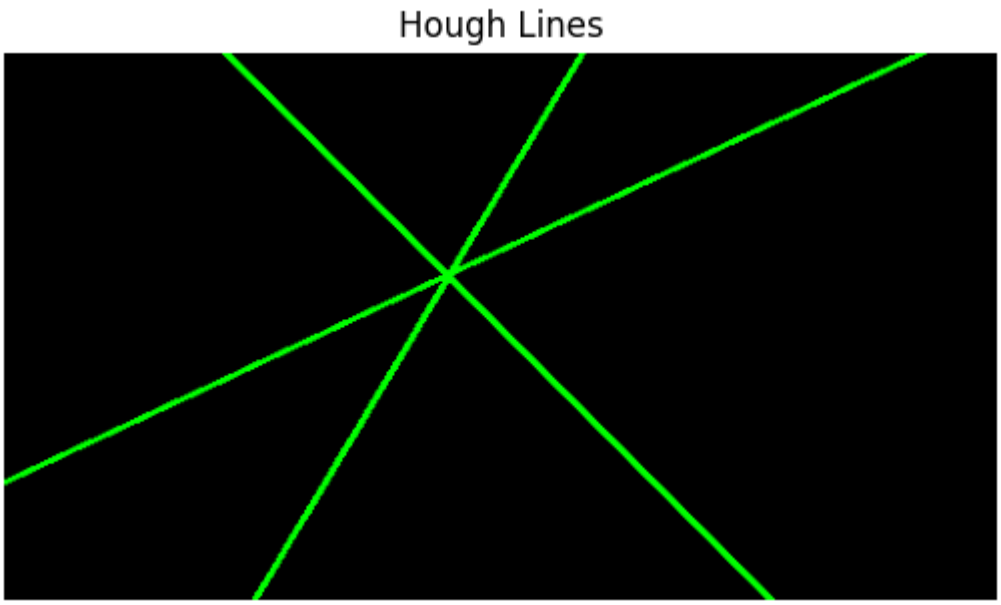
The `hough_peaks` function identifies the peaks in the NMS Hough accumulator array, which correspond to the most prominent lines in the image.

The function search for certain number of peeks to be displayed and remove all other peeks around them to identify only needs lines.

```
def hough_peaks(hough_acc, num_peaks, threshold=50, nhood_size=3):
    peaks = []
    acc = hough_acc.copy()
    for i in range(num_peaks):
        r, t = np.unravel_index(acc.argmax(), acc.shape)
        if acc[r, t] < threshold:
            break
        peaks.append((r, t))
        r1 = max(0, r - nhood_size)
        r2 = min(acc.shape[0], r + nhood_size + 1)
        t1 = max(0, t - nhood_size)
        t2 = min(acc.shape[1], t + nhood_size + 1)
        acc[r1:r2, t1:t2] = 0
    return peaks
peaks = hough_peaks(acc_nms, num_peaks=10, threshold=20, nhood_size=50)
```

These peaks are used to draw the detected lines on the image using the `draw_hough_lines` function.

```
def draw_hough_lines(img, lines, diag_len):
    img_out = np.zeros_like(img)
    for p_idx, theta in lines:
        p = p_idx - diag_len
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * p
        y0 = b * p
        x1 = int(x0 - 1700 * (b))
        x2 = int(x0 + 1700 * (b))
        y1 = int(y0 + 1700 * (a))
        y2 = int(y0 - 1700 * (a))
        cv.line(img_out, (x1, y1), (x2, y2), (0, 255, 0), 2)
    return img_out
lines = [(p, np.deg2rad(t)) for p, t in peaks]
diag_len = int(np.ceil(np.hypot(img.shape[0], img.shape[1])))
hough_lines = draw_hough_lines(img, lines, diag_len)
show_image(hough_lines, 'Hough Lines')
```

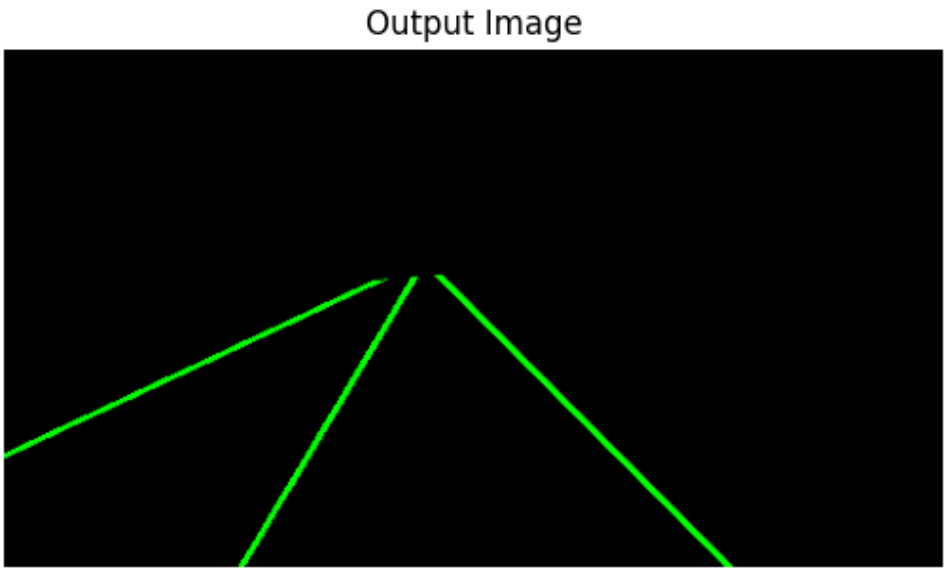


2.3.4 Draw the lines on the original image

Refine the output peaks lines to be shown only within the ROI. Mask the hough lines with the same mask used to mask the edge image.

```
output_img = cv.bitwise_and(hough_lines, cv.cvtColor(mask, cv.COLOR_GRAY2BGR))
show_image(output_img, 'Output Image')
```

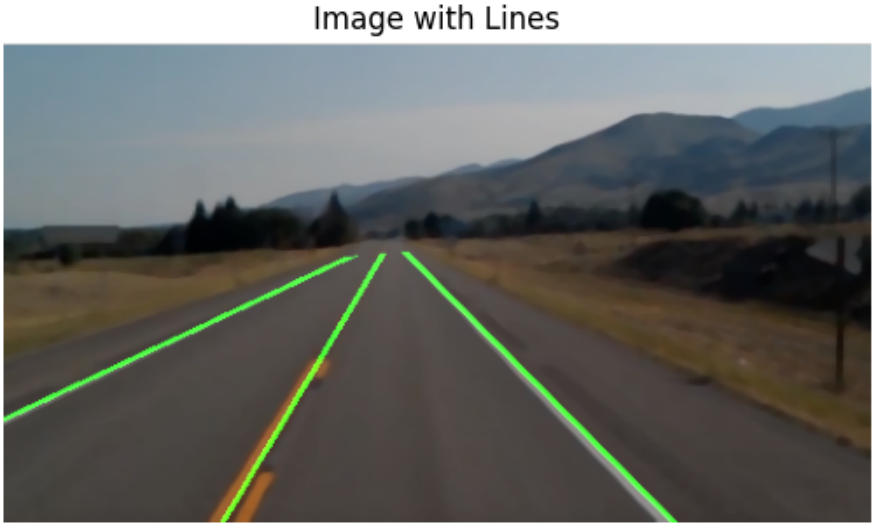
✓ 0.1s



Finally, the detected lines are combined with the original image using bitwise operations and weighted addition to highlight the lanes.

```
image_with_lines = cv.addWeighted(img, 0.8, output_img, 1, 0)
show_image(image_with_lines, 'Image with Lines')
```

✓ 0.1s



2.3.5 Examples

Original Image

Image with Lines

Original Image

Image with Lines