Indiana University Indianapolis
Department of Computer Science

Computer Vision

Project 2

Author
Shishir Yadav

Contents

1 Problem 1: Gradient Based Edge Detector Subproblem 1: Gradient Magnitude and Orientation

1.1 Problem

The goal is to implement a gradient-based edge detector that first smooths the image using a Gaussian filter and then computes horizontal and vertical derivatives using a Sobel filter. The gradient magnitude and orientation are then visualized.

1.2 Implementation

Steps involved in the Process:

1. Load the grayscale image.
2. Generate a Gaussian kernel for smoothing, based on the specified sigma.
3. Convolve the smoothed image with Sobel filters to calculate horizontal and vertical gradients.
4. Compute the gradient magnitude and orientation using the gradients.
5. Display the original image, gradient magnitude, and gradient orientation (quiver plot).

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

def generate_gaussian_kernel(kernel_size, sigma):

    kernel = np.zeros((kernel_size, kernel_size))
    half_size = kernel_size // 2
    sum_val = 0


    for i in range(kernel_size):
        for j in range(kernel_size):
            x = i - half_size
            y = j - half_size
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * sigma**2))
            sum_val += kernel[i, j]

    # Normalize the kernel so that the sum is 1
    kernel /= sum_val
    return kernel

def custom_convolution(image, kernel):

    kernel_height, kernel_width = kernel.shape
    image_height, image_width = image.shape

    # output array for the convolved result
    convolved_image = np.zeros((image_height - kernel_height + 1, image_width - kernel_width + 1)
    )

    # convolution
    for i in range(convolved_image.shape[0]):
        for j in range(convolved_image.shape[1]):
            region = image[i:i + kernel_height, j:j + kernel_width]
            convolved_image[i, j] = np.sum(region * kernel)

```

```
    return convolved_image

def gradient_edge_detection(image, sigma):

    # Generate Gaussian kernel
    kernel_size = int(6 * sigma) + 1  # Ensure odd kernel size
    gaussian_kernel = generate_gaussian_kernel(kernel_size, sigma)

    # Smooth the image
    smoothed_image = custom_convolution(image, gaussian_kernel)

    # Create gradient filters for x and y
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])  # Sobel filter for x-derivative
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])  # Sobel filter for y-derivative

    # Apply convolution with derivative filters
    grad_x = custom_convolution(smoothed_image, sobel_x)
    grad_y = custom_convolution(smoothed_image, sobel_y)

    # Compute gradient magnitude and orientation
    magnitude = np.sqrt(grad_x**2 + grad_y**2)
    orientation = np.arctan2(grad_y, grad_x)

    return magnitude, orientation

def display_results(image, magnitude, orientation):

    plt.figure(figsize=(15, 5))

    # Original Image
    plt.subplot(1, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')

    # Gradient Magnitude
    plt.subplot(1, 3, 2)
    plt.imshow(magnitude, cmap='gray')
    plt.title('Gradient Magnitude')
    plt.axis('off')

    #Orientation
    plt.subplot(1, 3, 3)
    step = 10
    Y, X = np.mgrid[step//2:magnitude.shape[0]:step, step//2:magnitude.shape[1]:step]
    U = np.cos(orientation[Y, X])
    V = np.sin(orientation[Y, X])
    plt.quiver(X, Y, U, V, color='red')
    plt.title('Gradient Orientation')
    plt.gca().invert_yaxis()
    plt.axis('off')

    plt.tight_layout()
    plt.show()


image_path = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\1.png'
image = np.array(Image.open(image_path).convert('L'), dtype=np.float32) / 255.0


```

```
sigma_value = 1.0


magnitude, orientation = gradient_edge_detection(image, sigma_value)


display_results(image, magnitude, orientation)
```

1.3 Observation

- Gradient Magnitude: The edges of the objects in the image were highlighted. The sharper the gradient, the more intense the edges appeared.
- The quiver plot shows the direction of the edges in the image, where arrows point in the direction of the edge gradient.

  The algorithm successfully detected the edges in the image. The gradient magnitude clearly identifies sharp transitions in pixel intensity, making it useful for detecting the boundaries of objects. The orientation plot, which uses a quiver plot, gives insight into the direction of these edges, which can be helpful in further edge-tracing tasks or understanding object shapes in the image.

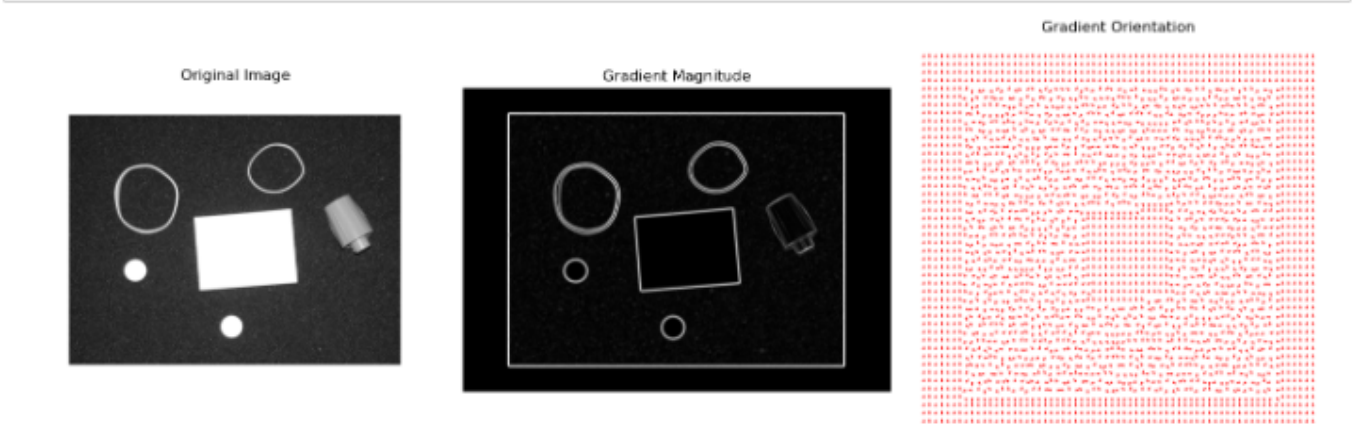


Figure 1. Results of Gradient Magnitude and Orientation

1.4 Conclusion

Gradient-based edge detection is effective in identifying strong edges in an image. The combination of Gaussian smoothing and Sobel filters provides a robust method for detecting and visualizing both the magnitude and direction of image gradients.
While the gradient magnitude effectively captures edge locations, the orientation plot provides additional information on the edge direction. Gaussian smoothing helps reduce noise, ensuring that the detected edges are prominent and not influenced by minor noise artifacts.

2 Problem 1: Subproblem 2 - Manual Boundary Tracing

2.1 Problem

The goal of this subproblem is to extract the outer boundary of objects in an image. First, edge detection is performed using the Canny edge detector. Then, the user manually selects seed points to

trace the boundaries of the objects.

2.2 Implementation

The following steps outline the implementation:

1. Edge Detection: The Canny edge detection algorithm is applied to identify the edges in the image.
2. Seed Point Selection: The user manually selects seed points on the boundary of each object. These points act as starting points for tracing the object boundaries.
3. Boundary Tracing: From each seed point, the algorithm traces the boundary of the object by checking neighboring pixels for edge continuity, ensuring that the boundary is traced as accurately as possible.
4. Visualization: The traced boundaries are plotted on the original image.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def perform_edge_detection(image):

    edges = cv2.Canny(image, 50, 150)
    kernel = np.ones((3, 3), np.uint8)
    edges = cv2.dilate(edges, kernel, iterations=1)
    return edges

def get_seed_points(edge_image):
    seed_points = []

    def on_mouse_click(event, x, y, flags, param):
        if event == cv2.EVENT_LBUTTONDOWN:
            seed_points.append((x, y))
            print(f"Seed point selected at: x={x}, y={y}")
            cv2.circle(edge_image_display, (x, y), 3, (255, 0, 0), -1)
            cv2.imshow('Edge Image', edge_image_display)

    edge_image_display = edge_image.copy()  # Copy for displaying selection feedback
    cv2.namedWindow('Edge Image')
    cv2.setMouseCallback('Edge Image', on_mouse_click)

    print("Click to select seed points. Press 'q' to finish selection.")

    while True:
        cv2.imshow('Edge Image', edge_image_display)
        key = cv2.waitKey(1) & 0xFF
        if key == ord('q'):  # Exit on 'q' key press
            break

    cv2.destroyAllWindows()

    if seed_points:
        return seed_points
    else:
        print("No seed points selected. Exiting.")
        exit()

def manual_boundary_tracing(edge_image, start_point):
```

```
    height, width = edge_image.shape
    traced_boundary = []
    visited = np.zeros_like(edge_image, dtype=bool)

    directions = [(-1,  0),  # N
                  (-1,  1),  # NE
                  ( 0,  1),  # E
                  ( 1,  1),  # SE
                  ( 1,  0),  # S
                  ( 1, -1),  # SW
                  ( 0, -1),  # W
                  (-1, -1)]  # NW

    current_point = start_point
    traced_boundary.append(current_point)
    visited[current_point[1], current_point[0]] = True

    backtrack_direction = 0

    while True:
        found_next = False

        for i in range(8):
            idx = (backtrack_direction + i) % 8
            dx, dy = directions[idx]
            x, y = current_point[0] + dx, current_point[1] + dy

            if 0 <= x < width and 0 <= y < height:
                if edge_image[y, x] != 0 and not visited[y, x]:
                    current_point = (x, y)
                    traced_boundary.append(current_point)
                    visited[y, x] = True
                    backtrack_direction = (idx + 5) % 8
                    found_next = True
                    break

        if not found_next or current_point == start_point or len(traced_boundary) > 10000:
            break

    return traced_boundary

def plot_traced_boundaries(original_image, all_boundaries):

    plt.figure(figsize=(10, 8))
    plt.imshow(original_image, cmap='gray')

    for boundary_points in all_boundaries:
        if boundary_points:
            x_coords, y_coords = zip(*boundary_points)
            plt.plot(x_coords, y_coords, 'r-', linewidth=2)
        else:
            print("No boundary points to display.")

    plt.title('Final Traced Boundaries')
    plt.axis('off')
    plt.show()

def main():

```

```
    image_path = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\1.png'
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print("Error loading image.")
        return


    _, binary_image = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)


    edges = perform_edge_detection(binary_image)


    seed_points = get_seed_points(edges)


    all_boundaries = []
    for seed_point in seed_points:
        print(f"Tracing boundary starting from seed point: {seed_point}")
        boundary = manual_boundary_tracing(edges, seed_point)
        all_boundaries.append(boundary)


    plot_traced_boundaries(image, all_boundaries)

if __name__ == "__main__":
    main()
```

2.3 Observation

- After selecting the seed points on the edges of the objects, the algorithm successfully traced the outer boundaries of each object in the image.
- Each seed point acted as a starting point for boundary tracing, and the algorithm followed edge continuity to outline the objects accurately.

The manual seed point selection ensures that the algorithm starts from the correct edge points. The traced boundaries, marked in red, closely follow the actual edges of the objects. The method can be improved by automating the seed point selection or applying edge refinement techniques to handle noisy or incomplete edges.

```
Click to select seed points. Press 'q' to finish selection.
Seed point selected at: x=493, y=170
Seed point selected at: x=400, y=88
Seed point selected at: x=195, y=118
Seed point selected at: x=335, y=194
Seed point selected at: x=329, y=349
Seed point selected at: x=172, y=264
Tracing boundary starting from seed point: (493, 170)
Tracing boundary starting from seed point: (400, 88)
Tracing boundary starting from seed point: (195, 118)
Tracing boundary starting from seed point: (335, 194)
Tracing boundary starting from seed point: (329, 349)
Tracing boundary starting from seed point: (172, 264)
```

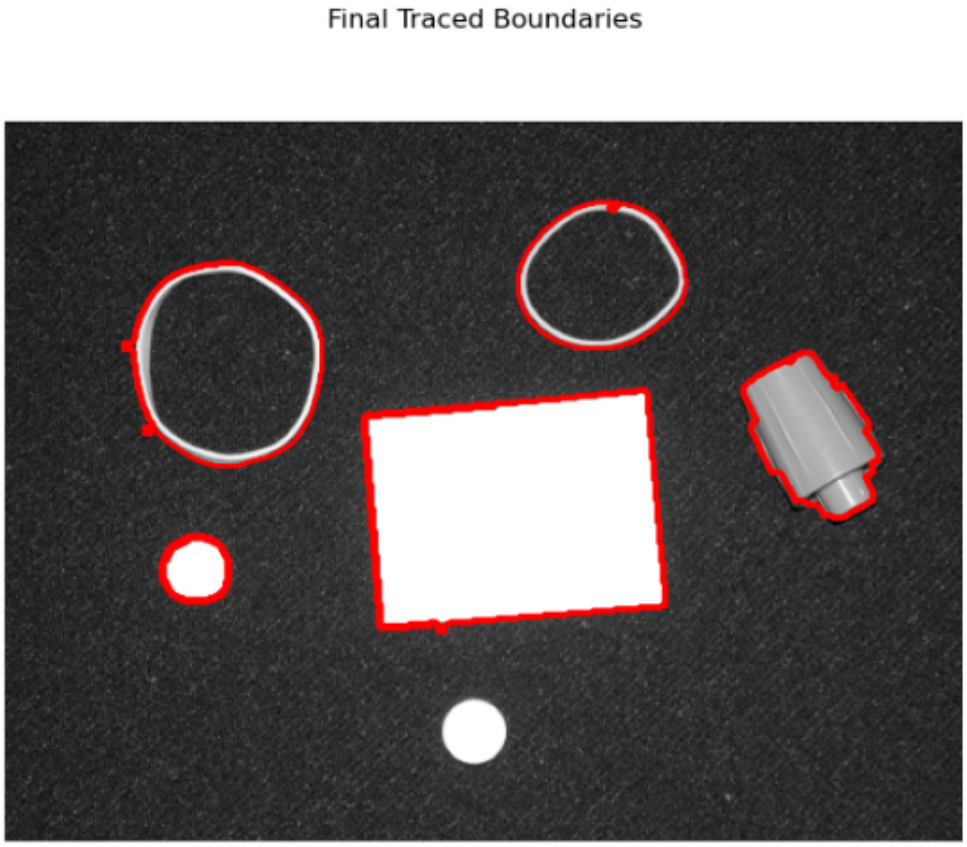


Figure 2. Manual Boundary Tracing of Objects

2.4 Conclusion

Manual boundary tracing allows for precise object outline extraction by selecting seed points for each object.The method works well for objects with distinct edges but may require additional refinement techniques for more complex or noisy images.

3 Problem 2: Corner Detection

3.1 Problem

The task is to implement the Harris corner detection algorithm to automatically detect corner points in two images. The Harris corner detector computes corner responses for each pixel, and local maxima within a specified window are identified as corners.

3.2 Implementation

1. **Image Gradients:** Compute the horizontal and vertical gradients of the image using the Sobel filters.
2. **Gradient Products:** Calculate the gradient products I_x^2, I_y^2, and $I_x I_y$.
3. **Gaussian Smoothing:** Apply Gaussian smoothing to the gradient products to reduce noise and focus on the significant corner points.
4. **Harris Response:** For each pixel, calculate the Harris corner response R using the formula:

$$R = \det(M) - k \cdot \text{trace}(M)^2$$

   where M is the second-moment matrix (or structure tensor) of the image at each pixel, $\det(M)$ is the determinant of the matrix, and $\text{trace}(M)$ is the trace (sum of diagonal elements).
5. **Local Maxima Detection:** Identify local maxima in the Harris response map within a window to detect the corner points.
6. **Visualization:** The corner points are displayed on the original images.

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

def load_image(image_path):
    img = Image.open(image_path).convert('L')
    img = np.array(img, dtype=np.float32)
    return img

def compute_gradients(img):

    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]], dtype=np.float32)
    sobel_y = np.array([[ 1,  2,  1],
                        [ 0,  0,  0],
                        [-1, -2, -1]], dtype=np.float32)

    grad_x = convolve(img, sobel_x)
    grad_y = convolve(img, sobel_y)
    return grad_x, grad_y

def gaussian_kernel(size, sigma=1.0):
    kernel = np.zeros((size, size), dtype=np.float32)
    center = size // 2
    sum_total = 0.0
    for i in range(size):
        for j in range(size):
            x = i - center
            y = j - center
            kernel[i, j] = np.exp(-(x**2 + y**2)/(2 * sigma**2))
            sum_total += kernel[i, j]
    kernel = kernel / sum_total
    return kernel

def convolve(image, kernel):
    img_height, img_width = image.shape
    kernel_height, kernel_width = kernel.shape
    pad_h = kernel_height // 2
```

```
    pad_w = kernel_width // 2
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='edge')
    output = np.zeros_like(image, dtype=np.float32)
    kernel_flipped = np.flipud(np.fliplr(kernel))
    for y in range(img_height):
        for x in range(img_width):
            region = padded_image[y:y+kernel_height, x:x+kernel_width]
            output[y, x] = np.sum(region * kernel_flipped)
    return output

def harris_corner_detector(img, sigma=1.0, k=0.04, window_size=3):
    grad_x, grad_y = compute_gradients(img)
    Ixx = grad_x ** 2
    Ixy = grad_x * grad_y
    Iyy = grad_y ** 2
    size = window_size
    gaussian = gaussian_kernel(size, sigma)
    Sxx = convolve(Ixx, gaussian)
    Sxy = convolve(Ixy, gaussian)
    Syy = convolve(Iyy, gaussian)
    det_M = (Sxx * Syy) - (Sxy ** 2)
    trace_M = Sxx + Syy
    R = det_M - k * (trace_M ** 2)
    return R

def find_local_maxima(R, threshold=0.01, window_size=3):
    corners = []
    R_max = np.max(R)
    threshold_value = threshold * R_max
    offset = window_size // 2
    height, width = R.shape
    for y in range(offset, height - offset):
        for x in range(offset, width - offset):
            if R[y, x] > threshold_value:
                local_window = R[y - offset:y + offset + 1, x - offset:x + offset + 1]
                if R[y, x] == np.max(local_window):
                    corners.append((y, x))
    return corners

def display_corners(img, corners):
    plt.figure(figsize=(8,6))
    plt.imshow(img, cmap='gray')
    if corners:
        y_coords, x_coords = zip(*corners)
        plt.scatter(x_coords, y_coords, color='red', s=10)
    plt.title("Detected Corners")
    plt.axis('off')
    plt.show()

def main():
    # Load images
    image_path_1 = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\2-1.jpg'
    image_path_2 = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\2-2.jpg'
    img1 = load_image(image_path_1)
    img2 = load_image(image_path_2)
    # Apply Harris corner detection
    R1 = harris_corner_detector(img1, sigma=1.5, k=0.04, window_size=3)
    R2 = harris_corner_detector(img2, sigma=1.5, k=0.04, window_size=3)
    # Find local maxima
    corners1 = find_local_maxima(R1, threshold=0.01, window_size=3)
```

```
    corners2 = find_local_maxima(R2, threshold=0.01, window_size=3)
    # Display detected corners
    display_corners(img1, corners1)
    display_corners(img2, corners2)
    print("Corner detection completed.")

if __name__ == "__main__":
    main()
```

3.3 Observation

- **Chessboard Image (2-1.jpg):** The corners of the chessboard pattern were successfully detected, and the algorithm was able to identify strong corners at the intersections of the black and white squares.
- **Cow Toy Image (2-2.jpg):** The algorithm detected multiple corners on the toy, especially around the legs and spots where there are sharp changes in intensity.

The Harris corner detection algorithm is effective at identifying corners in both structured and unstructured images. In the chessboard image, the corners are easily detected due to the clear intensity changes at the intersections of squares. In the cow toy image, the algorithm still works well, detecting corners around areas with strong intensity variations, such as around the legs and body spots of the toy.

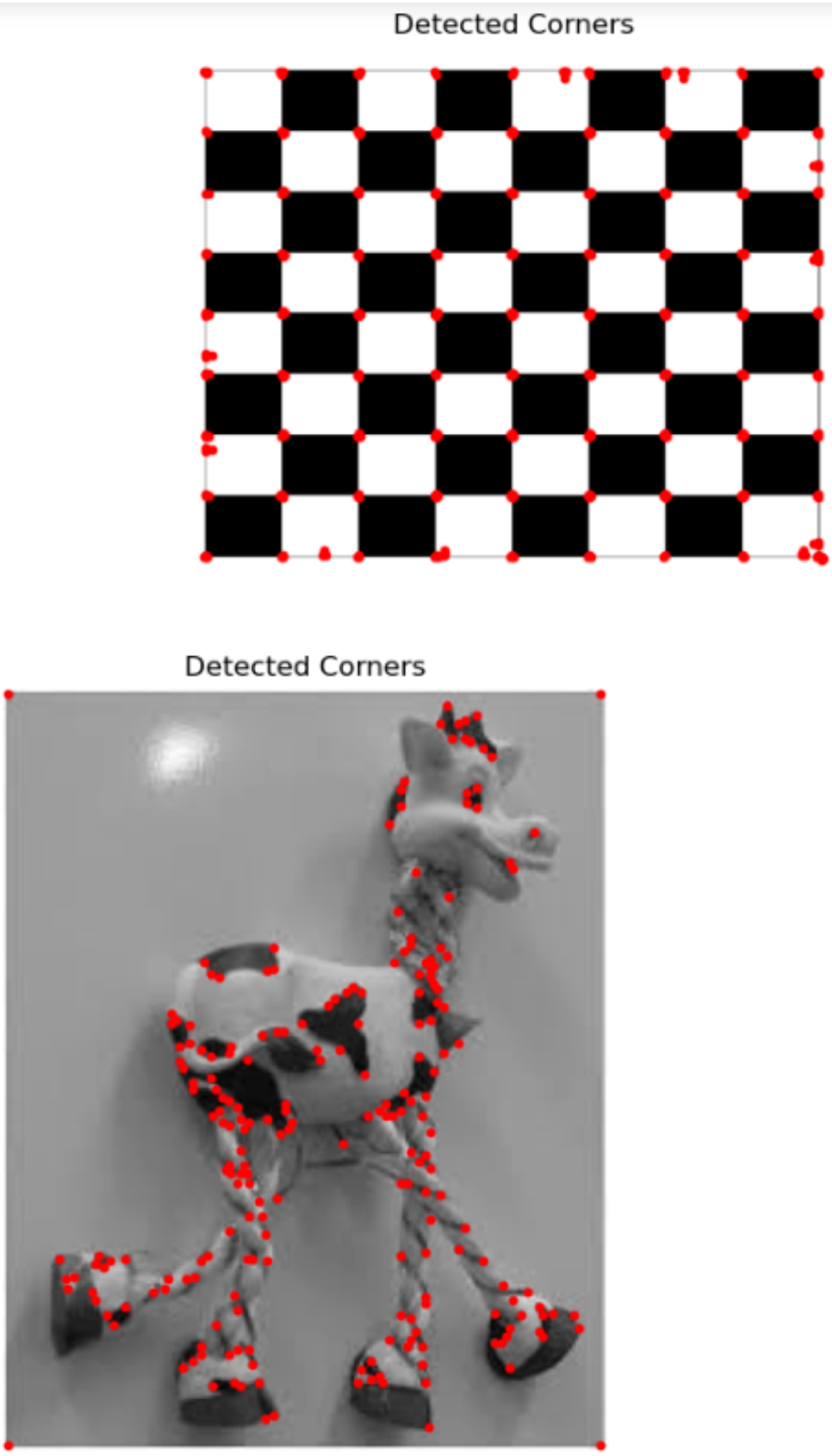


Figure 3. Detected Corners on Images

3.4 Conclusion

The Harris corner detector is a robust method for detecting corners in images with distinct intensity variations. The algorithm successfully identifies corner points in both highly structured images, like the chessboard, and more complex scenes, like the toy. Fine-tuning parameters such as the threshold and window size can further improve detection accuracy.

4 Problem 3: Subproblem 1 - Line Detection Using Hough Transform

4.1 Problem

The task is to implement the Hough Transform to detect straight lines in an image. The algorithm should compute the Hough Transform to identify line parameters and extract line segments based on the detected lines.

4.2 Implementation

The following steps outline the implementation of the Hough Transform for line detection

1. **Edge Detection**: Apply the Canny edge detector to the input image to identify potential line edges.
2. **Hough Transform**: Compute the Hough Transform accumulator array. For each edge point in the image, a sinusoidal curve is plotted in the Hough space corresponding to all possible lines that could pass through that point.
3. **Line Detection**: From the Hough Transform accumulator, identify local maxima that correspond to strong lines in the image.
4. **Line Segmentation**: Extract line segments from the detected lines by tracing the line through the edge image and selecting points that belong to the line.
5. **Visualization**: Plot the detected line segments on the original image.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, feature

#Define the Hough Transform Function
def perform_hough_transform(edge_img, theta_res=1, rho_res=1):

    rows, cols = edge_img.shape

    # Define theta and rho ranges
    thetas = np.deg2rad(np.arange(-90.0, 90.0, theta_res))
    diag_len = int(np.ceil(np.sqrt(rows**2 + cols**2)))
    rhos = np.arange(-diag_len, diag_len, rho_res)

    accumulator = np.zeros((len(rhos), len(thetas)), dtype=int)

    edge_points = np.argwhere(edge_img)

    for y, x in edge_points:
        for theta_idx, theta in enumerate(thetas):
            rho = int((x * np.cos(theta)) + (y * np.sin(theta)))
            rho_idx = np.argmin(np.abs(rhos - rho))
```

```
            accumulator[rho_idx, theta_idx] += 1

    return accumulator, thetas, rhos

#Detect Lines from the Hough Accumulator
def extract_lines_from_accumulator(accumulator, thetas, rhos, num_lines=10, threshold=30):
    lines = []

    for _ in range(num_lines):
        max_idx = np.argmax(accumulator)
        rho_idx, theta_idx = np.unravel_index(max_idx, accumulator.shape)

        rho = rhos[rho_idx]
        theta = thetas[theta_idx]

        if accumulator[rho_idx, theta_idx] > threshold:
            lines.append((rho, theta))

        accumulator[max(0, rho_idx-8):min(accumulator.shape[0], rho_idx+8),
                    max(0, theta_idx-8):min(accumulator.shape[1], theta_idx+8)] = 0

    return lines

#Extract Line Segments from Detected Lines
def find_line_segments(image, edges, lines, min_length=15):
    segments = []
    height, width = image.shape

    for rho, theta in lines:
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 3000 * (-b))
        y1 = int(y0 + 3000 * (a))
        x2 = int(x0 - 3000 * (-b))
        y2 = int(y0 - 3000 * (a))

        mask = np.zeros_like(image, dtype=np.uint8)
        rr, cc = np.linspace(y1, y2, num=3000, dtype=int), np.linspace(x1, x2, num=3000, dtype=
    int)
        mask[rr.clip(0, height-1), cc.clip(0, width-1)] = 1

        intersections = np.logical_and(mask, edges)
        points = np.argwhere(intersections)

        if len(points) >= 2:
            dist = np.linalg.norm(points[-1] - points[0])
            if dist >= min_length:
                segments.append((tuple(points[0][::-1]), tuple(points[-1][::-1])))

    return segments

#Ploting Detected Line Segments
def plot_line_segments(image, segments):
    plt.imshow(image, cmap='gray')

    for (x1, y1), (x2, y2) in segments:
        plt.plot([x1, x2], [y1, y2], 'r-', linewidth=2)

```

```
    plt.title("Detected Line Segments")
    plt.axis('off')
    plt.show()

#Main Function to Run the Extended Hough Transform
def hough_transform_pipeline(image_path, sigma=2.0, num_lines=10, min_length=15):
    image = io.imread(image_path, as_gray=True)

    edges = feature.canny(image, sigma=sigma)
    accumulator, thetas, rhos = perform_hough_transform(edges)
    lines = extract_lines_from_accumulator(accumulator, thetas, rhos, num_lines)
    line_segments = find_line_segments(image, edges, lines, min_length)
    plot_line_segments(image, line_segments)


image_path = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\3.png'
hough_transform_pipeline(image_path, sigma=2.0, num_lines=10, min_length=15)
```

4.3 Observation

Following are the observations after applying Hough Transform

- The algorithm successfully detected the straight edges of the objects in the image. The detected line segments match well with the edges of the objects.
- The line segments are visualized by plotting them on the original image.

The Hough Transform is effective for detecting straight lines in an image. The method converts the detection problem into a voting scheme in Hough space, where lines are detected based on maxima in the accumulator array. The extraction of line segments further refines the detected lines by finding the intersection of the line with edge points in the image.

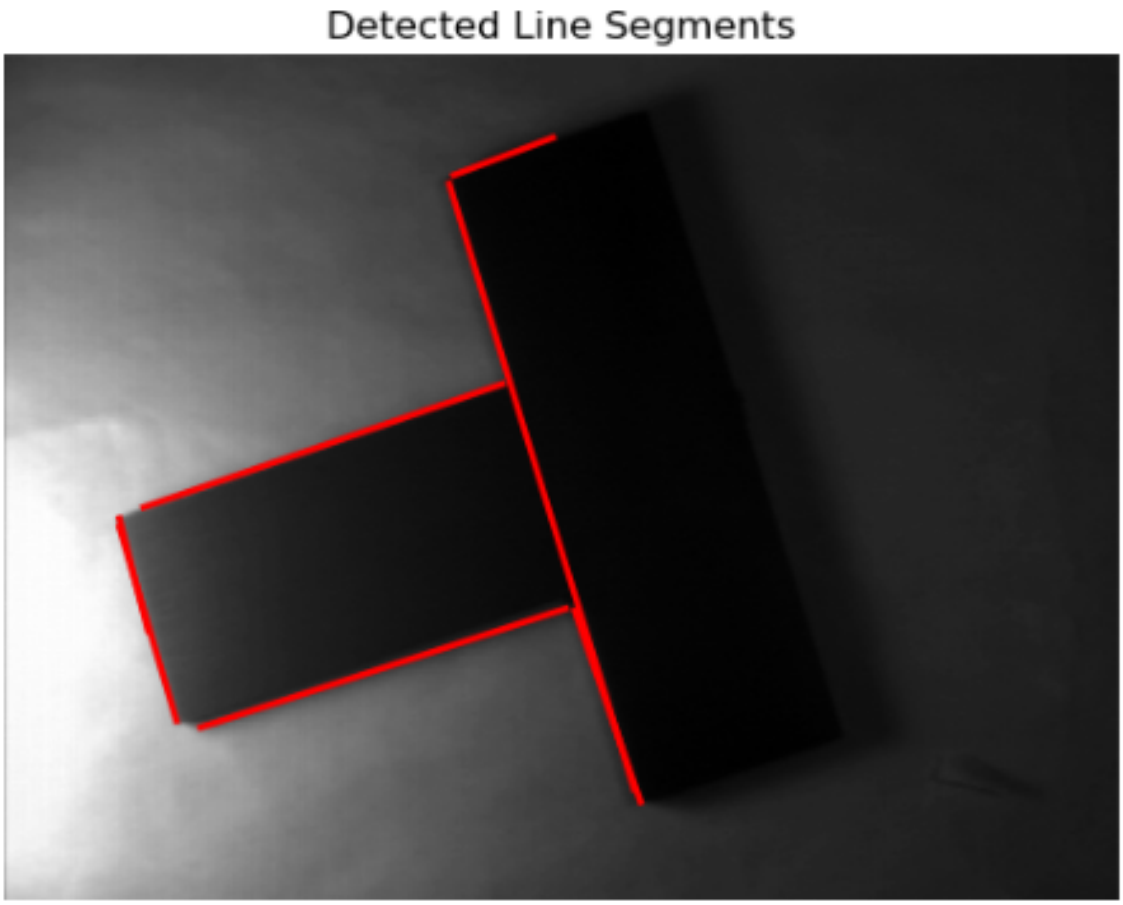


Figure 4. Detected Line Segments Using Hough Transform

4.4 Conclusion

Hough Transform effectively detects lines in an image. The detected line segments align with the edges present in the image.
The Hough Transform algorithm provides a robust way to detect straight lines in images. By using the accumulator to identify line parameters and refining them into line segments, the method can accurately detect prominent lines in structured objects. The performance could be enhanced by fine-tuning the threshold and adjusting the minimum line length for different types of images.

5 Problem 3: Subproblem 2 - Circle Detection Using Hough Transform

5.1 Problem

The task is to extend the Hough Transform to detect circular shapes. A training image with a single circular object is used to build a circle template. This template is then used to detect circles in a new binary image by applying the Generalized Hough Transform.

5.2 Implementation

The implementation involves the following steps

1. **Training (Circle Template Construction):** A binary image containing a single circular object is provided. The center of the circle and boundary points are used to compute the average radius of the circle.
2. A template of the circle is created based on the computed radius using angular coordinates.
3. **Circle Detection in a New Image:** The circle template is used to detect similar circles in a new binary image. For each edge point in the image, potential circle centers are accumulated, and the most frequent points in the accumulator array are identified as the centers of the circles.
4. **Visualization:** The detected circles are plotted on the original image, and the coordinates of the circle centers are printed.

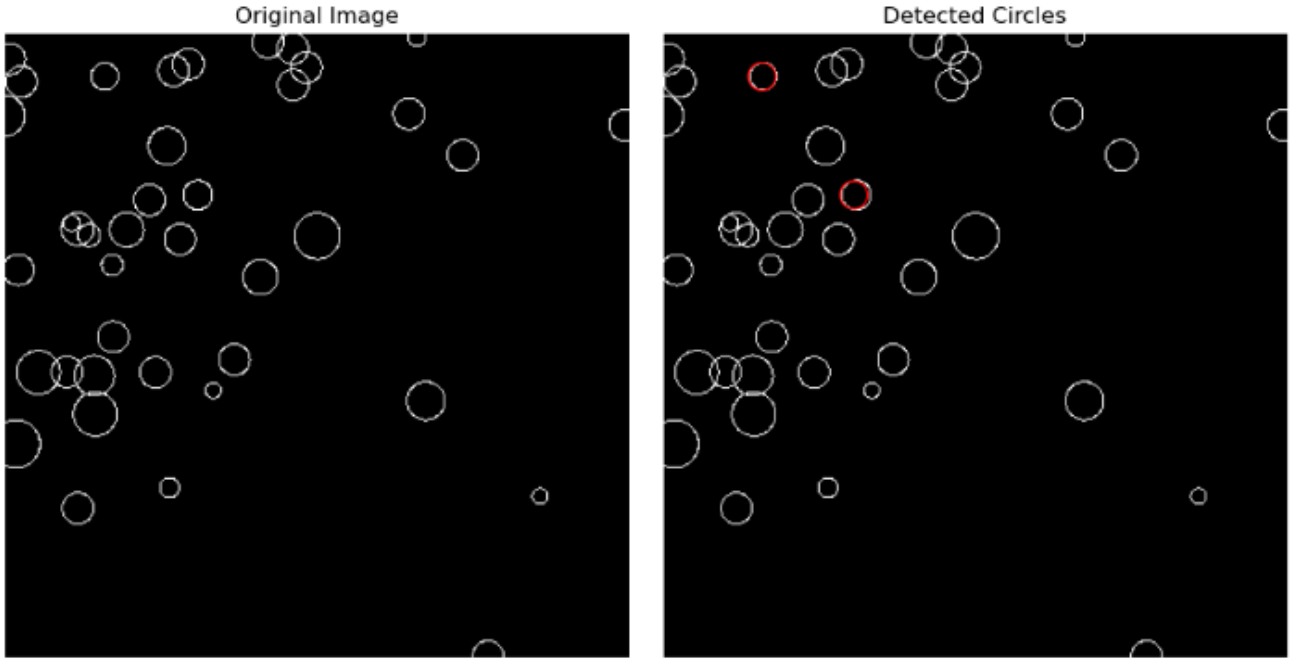
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import scipy.io as sio

def train_circle_detector(binary_image, center_point, boundary_points):

    # Computing radius based on boundary points
    radius_values = [np.sqrt((x - center_point[0])**2 + (y - center_point[1])**2) for x, y in
    boundary_points]
    estimated_radius = np.mean(radius_values)

    # Creating a circle template based on the calculated radius
    angles = np.linspace(0, 2 * np.pi, 100)
    x_circle = estimated_radius * np.cos(angles)
    y_circle = estimated_radius * np.sin(angles)

    # Store results in a dictionary
    detector_data = {
        'radius': estimated_radius,
        'x_template': x_circle,
        'y_template': y_circle
    }

    return detector_data

def detect_circles_in_image(new_binary_image, trained_data):

    radius = trained_data['radius']
    x_template = trained_data['x_template']
    y_template = trained_data['y_template']

    img_height, img_width = new_binary_image.shape
    accumulator_array = np.zeros((img_height, img_width))

    # Identifing edge pixels in the binary image
    edge_pixels = np.argwhere(new_binary_image > 0)
    for y_pixel, x_pixel in edge_pixels:
        for dx, dy in zip(x_template, y_template):
            center_x = int(x_pixel - dx)
            center_y = int(y_pixel - dy)
            if 0 <= center_x < img_width and 0 <= center_y < img_height:
                accumulator_array[center_y, center_x] += 1

    # Find the top 2 circle centers from the accumulator
    detected_centers = []
    for _ in range(2):
        max_index = np.unravel_index(np.argmax(accumulator_array), accumulator_array.shape)
```

```
        detected_centers.append((max_index[1], max_index[0]))
        y, x = max_index
        accumulator_array[max(0, y-10):min(img_height, y+11), max(0, x-10):min(img_width, x+11)] = 0

    return detected_centers

def visualize_detection(original_image, circle_centers, radius):
    plt.figure(figsize=(10, 5))

    # original image
    plt.subplot(1, 2, 1)
    plt.imshow(original_image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')

    # Detected circles
    plt.subplot(1, 2, 2)
    plt.imshow(original_image, cmap='gray')
    plt.title('Detected Circles')
    plt.axis('off')

    #Circles on the image
    for center in circle_centers:
        circle_outline = plt.Circle(center, radius, color='r', fill=False)
        plt.gca().add_artist(circle_outline)

    plt.tight_layout()
    plt.show()

train_data = sio.loadmat(r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\train.mat')
circle_center = (train_data['c'][0][0], train_data['c'][0][1])  # Use center point from mat
boundary_coords = [(x[0], x[1]) for x in train_data['ptlist']]  # Convert boundary points

train_image_file = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\train.png'  # Path to your training image
train_img = np.array(Image.open(train_image_file).convert('L'))
train_binary_img = (train_img > 128).astype(np.uint8)

trained_circle_data = train_circle_detector(train_binary_img, circle_center, boundary_coords)

test_image_file = r'C:\Users\Shishir yadav\Desktop\Folders\Fall24\comp v\Project2\test.png'  # Path to your test image
test_img = np.array(Image.open(test_image_file).convert('L'))
test_binary_img = (test_img > 128).astype(np.uint8)

circle_centers = detect_circles_in_image(test_binary_img, trained_circle_data)

visualize_detection(test_img, circle_centers, trained_circle_data['radius'])

print("Detected circle centers:")
for i, center in enumerate(circle_centers, 1):
    print(f"Circle {i}: center at {center}")
```

5.3 Observation

Two circles were detected based on the trained circle data.The detected circles are highlighted on the test image.The output shows the detected circle centers for both the circle.

The Generalized Hough Transform for circle detection works by accumulating potential circle centers based on the boundary points of the detected edges. The circle template constructed during the training phase allows for accurate detection of circles with a similar radius in new images. The accumulator array helps in identifying the most likely centers of circular objects, which are then highlighted in the output.




```
Detected circle centers:
Circle 1: center at (63, 27)
Circle 2: center at (122, 103)
```

Figure 5. Detected Circles Using Hough Transform

5.4 Conclusion

The Hough Transform successfully detects circles based on the radius and boundary points provided during the training phase.
The circle detection algorithm based on the Generalized Hough Transform is effective for detecting circular shapes in images. By using a circle template constructed from a training image, the method can reliably detect similar circles in new images. Fine-tuning the accumulator resolution and the radius estimation could further improve the detection accuracy for different types of circles.