2024

HOUGH TRANSFORM & SNAKE

TEAM 14

- AHMED MOHAMED ALI
- ALI SHERIF
- MUHANNAD ABDALLAH
- OSAMA MOHAMED ALI

SUBMITTED TO

- DR. AHMED M. BADAWI
- ENG. LAILA ABBAS
- ENG. OMAR DAWAH

TABLE OF CONTENTS

INTRODUCTION

Introduction:

In the realm of computer vision and image processing, edge and boundary detection play pivotal roles in various applications, ranging from object recognition and tracking to medical image analysis. In this assignment, we embark on a comprehensive exploration of edge and boundary detection techniques, focusing on two fundamental methodologies: the Canny edge detector and the Active Contour Model (Snake).

Objective:

The primary objective of this assignment is to implement and evaluate state-of-the-art algorithms for edge detection and boundary delineation. Specifically, we aim to:

1. Utilize the Canny edge detector to detect edges in grayscale and color images, identifying lines, circles, and ellipses present in the images.
2. Employ the Active Contour Model, also known as Snake, to initialize and evolve contours around specified objects within the images. We will utilize a greedy algorithm for contour evolution and represent the outputs as chain codes, enabling the computation of perimeter and area enclosed by the contours

Methodology:

1. Canny Edge Detection: We will implement the Canny edge detection algorithm, a multi-step process involving Gaussian blurring, gradient calculation, non-maximum suppression, and double thresholding with hysteresis. The algorithm will be applied to grayscale and color images to detect edges and superimpose the detected shapes on the images.
2. Active Contour Model (Snake): We will initialize contours around specified objects in the images and utilize a greedy algorithm for contour evolution. The Snake algorithm involves minimizing an energy function comprising elastic and smoothness terms. The output contours will be represented as chain codes, facilitating the computation of perimeter and enclosed area.

CANNY EDGE DETECTION

Introduction to Canny Edge Detection

In this section, we discuss the implementation of the Canny edge detector algorithm. The Canny edge detector is a popular technique used in computer vision and image processing for detecting edges in images. It is widely known for its ability to accurately detect edges while minimizing noise and false detections..

Our approach:

We implemented the Canny edge detection algorithm using a custom function with the following key components:

1. **Gaussian Blur:**

- **Purpose:** The first step is to reduce noise in the image and smooth out irregularities.
- **Implementation:** We applied a Gaussian blur to the input image, which convolves the image with a Gaussian kernel.
- **Control:** The size of the Gaussian kernel was set to (3, 3), allowing control over the amount of blurring.

2. **Gradient Calculation:**

- **Purpose:** This step computes the intensity and direction of gradients in the image.
- **Implementation:** We used the Sobel operator to calculate the derivatives of the image in both the horizontal (x) and vertical (y) directions.
- **Accuracy:** To ensure accurate gradient calculation, we used a data type of cv2.CV_64F.

3. **Non-maximum Suppression:**

- **Purpose:** To refine the edge map by thinning out the detected edges.
- **Implementation:** For each pixel, we compared its gradient magnitude with neighboring pixels along the gradient direction.

4. **Retainment:** Only pixels that represent local maxima in the gradient direction were retained, suppressing non-maximum responses.

5. **Double Thresholding and Hysteresis:**

- **Purpose:** This step categorizes pixels into strong, weak, or non-edge pixels based on their gradient magnitudes.
- **Implementation:** Pixels with gradient magnitudes above a high threshold are classified as strong edges, while those between low and high thresholds are weak edges.
- **Connection:** Weak edges that are adjacent to strong edges are identified through a process called hysteresis, ensuring the tracing of continuous edges.

```python
    def canny_edge_detection(self, image, low_threshold=0, high_threshold=60):
        # Step 1: Apply Gaussian blur to the image
        blurred = cv2.GaussianBlur(image, (3, 3), 0)

        # Step 2: Calculate gradient intensity and direction
        dx = cv2.Sobel(blurred, cv2.CV_64F, 1, 0, ksize=3)
        dy = cv2.Sobel(blurred, cv2.CV_64F, 0, 1, ksize=3)
        gradient_magnitude = np.sqrt(dx ** 2 + dy ** 2)
        gradient_direction = np.arctan2(dy, dx) * (180 / np.pi)

        # Step 3: Non-maximum suppression
        suppressed = np.zeros_like(gradient_magnitude)
        for i in range(1, gradient_magnitude.shape[0] - 1):
            for j in range(1, gradient_magnitude.shape[1] - 1):
                direction = gradient_direction[i, j]
                if (0 <= direction < 22.5) or (157.5 <= direction <= 180):
                    neighbors = [gradient_magnitude[i, j + 1], gradient_magnitude[i, j - 1]]
                elif (22.5 <= direction < 67.5):
                    neighbors = [gradient_magnitude[i - 1, j - 1], gradient_magnitude[i + 1, j + 1]]
                elif (67.5 <= direction < 112.5):
                    neighbors = [gradient_magnitude[i - 1, j], gradient_magnitude[i + 1, j]]
                else:
                    neighbors = [gradient_magnitude[i - 1, j + 1], gradient_magnitude[i + 1, j - 1]]
                if gradient_magnitude[i, j] >= max(neighbors):
                    suppressed[i, j] = gradient_magnitude[i, j]

        # Step 4: Apply double thresholding and edge tracking by hysteresis
        strong_edges = (suppressed > high_threshold).astype(np.uint8) * 255
        weak_edges = ((suppressed >= low_threshold) & (suppressed <= high_threshold)).astype(np.uint8)
* 255
        hysteresis_edges = cv2.bitwise_and(strong_edges, cv2.dilate(weak_edges, None))

        return hysteresis_edges
```

Observations:

The Canny edge detector is renowned for its ability to accurately detect edges while mitigating the effects of noise and minimizing false detections. This is achieved through a multi-step process that involves Gaussian blurring to reduce noise, gradient calculation to identify edge strength and direction, non-maximum suppression to thin out detected edges, and double thresholding followed by hysteresis to classify pixels as strong, weak, or non-edges based on intensity values. However, achieving optimal results often requires careful adjustment of parameters, such as the low and high thresholds used in the double thresholding step, to strike a balance between sensitivity and specificity in edge detection. Despite its complexity, the Canny edge detector maintains reasonable computational efficiency, making it a versatile tool for a wide range of image processing tasks.

LINES DETECTION

Introduction to Hough Line Detection

Hough line detection is a fundamental technique in computer vision used to identify straight lines within an image. It operates on the principle of transforming points in the image space to a parameter space where lines are represented. This transformation allows robust detection of lines even in the presence of noise, occlusions, or variations in line orientation and length.

In this method, each edge point in the image contributes to a set of possible lines in the parameter space, typically represented by two parameters: rho (ρ) and theta (θ). Rho denotes the perpendicular distance from the origin to the line, while theta represents the angle between the x-axis and the perpendicular line.

The Hough transform accumulates votes in the parameter space for each edge point, forming peaks that correspond to lines in the image space. By thresholding these peaks, significant lines can be extracted, providing valuable information for tasks such as object recognition, lane detection in autonomous vehicles, and lineament analysis in geological imaging.

```
def hough_line_detection(self,threshold = 200):
        height, width = self.edged_image.shape
        diagonal = int(np.sqrt(height**2 + width**2))
        hough_space = np.zeros((2*diagonal, 180), dtype=np.uint64)
        edges_points = np.nonzero(self.edged_image)

        cos_theta = np.cos(np.deg2rad(np.arange(180)))
        sin_theta = np.sin(np.deg2rad(np.arange(180)))

        for i in range(len(edges_points[0])):
            x = edges_points[1][i]
            y = edges_points[0][i]
            rho_values = np.round(x * cos_theta + y * sin_theta).astype(int)
            np.add.at(hough_space, (rho_values + diagonal, np.arange(180)),
1)
        rows, cols = np.where(hough_space >= threshold)
        diag = rows - diagonal
        theta = cols
        return diag, theta
```

Our approach

we implemented the hough line detection using several steps:

1-**Initialization**:

- Compute the dimensions of the Hough accumulator array based on the input image.
- Calculate the diagonal of the image to ensure adequate coverage of possible lines.

2- **Edge Detection:**

- Identify edge points within the image.

3- **Hough Transform:**

- For each edge point, iterate over a range of angles (theta).
- Compute corresponding rho values using the parametric equation of a line.
- Update the Hough accumulator array by incrementing the corresponding cell.

4- **Thresholding:**

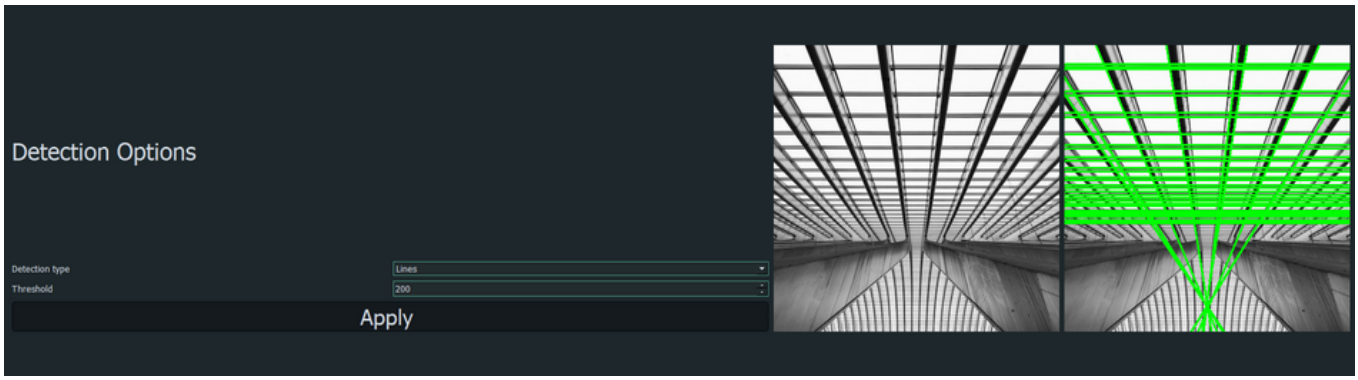
- Apply a threshold to the accumulator array.
- Identify cells exceeding the threshold as potential lines.

5- **Post-processing:**

- Determine rows and columns of cells surpassing the threshold.
- Compute diagonal offsets and theta values for detected lines.

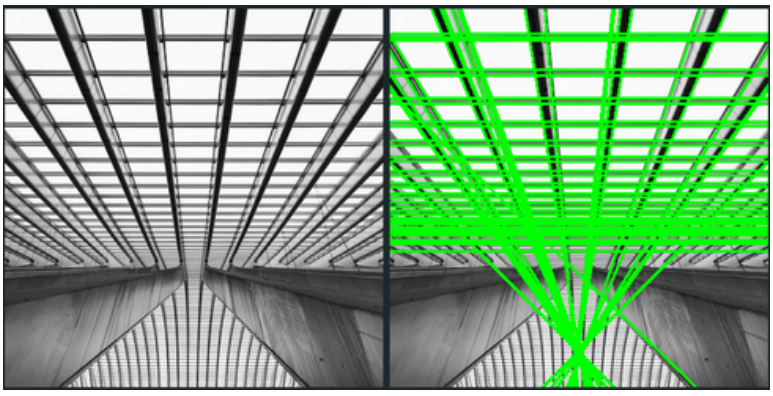
6- **Output:**

- Return detected lines represented by diagonal offsets and theta values.



fig(1)

When varying the threshold value in the Hough transform, I observed a direct correlation between the threshold and the number of detected lines. As depicted in Figure 1, a higher threshold led to fewer lines being detected, as indicated by the sparse distribution of scores in the accumulator matrix. Conversely, when the threshold was decreased, as illustrated in Figure 2, the number of detected lines increased, corresponding to a denser distribution of scores in the accumulator matrix



fig(2)

CIRCLES DETECTION

Introduction to Hough Circles Detection

Hough circle detection is a pivotal technique in computer vision utilized for identifying circular shapes within an image. Similar to line detection, it operates on the principle of transforming points from the image space to a parameter space where circles are represented. This transformation facilitates robust detection of circles amidst noise, occlusions, and variations in circle size and position.

In this method, each edge point in the image contributes to a set of possible circles in the parameter space, typically represented by three parameters: the x-coordinate of the circle center (a), the y-coordinate of the circle center (b), and the radius (r). These parameters define the position and size of candidate circles within the image.

The Hough transform accumulates votes in the parameter space for each edge point, forming peaks that correspond to circles in the image space. By thresholding these peaks, significant circles can be extracted, providing valuable information for tasks such as object detection, feature extraction, and medical image analysis.

Hough circle detection serves as a versatile tool in various applications, including object recognition, robotic navigation, and quality control in manufacturing processes. Its ability to robustly detect circular shapes makes it indispensable in many computer vision systems.

```python
def hough_circle_detection(self, min_radius=60, max_radius=100):
        edges_points = np.nonzero(self.edged_image)
        h, w = self.edged_image.shape
        hough_space = np.zeros((h, w, max_radius - min_radius + 1), dtype=np.uint64)
        x_points, y_points = edges_points

        cos_theta = np.cos(np.deg2rad(np.arange(360)))
        sin_theta = np.sin(np.deg2rad(np.arange(360)))

        for r in range(min_radius, max_radius + 1):
            for theta in range(360):
                a = np.round(x_points - r * cos_theta[theta]).astype(int)
                b = np.round(y_points - r * sin_theta[theta]).astype(int)
                valid_indices = np.where((a >= 0) & (a < w) & (b >= 0) & (b < h))
                a_valid, b_valid = a[valid_indices], b[valid_indices]
                hough_space[b_valid, a_valid, r - min_radius] += 1

        a, b, radius = np.where(hough_space >= self.threshold)
        return a, b, radius + min_radius
```

Our approach

1-**Initialization:**

- Obtain edge points from the input image using the np.nonzero function.
- Determine the dimensions of the Hough accumulator array (hough_space) based on the height and width of the input image, along with the specified range of radii (min_radius to max_radius).

2-**Edge Point Iteration:**

- Iterate over each edge point in the image, represented by their x and y coordinates (x_points and y_points).

3-**Hough Transform:**

- For each candidate radius within the specified range (min_radius to max_radius):
- Iterate over 360 degrees of theta values.
- Compute the corresponding circle center coordinates (a, b) using the parametric equation of a circle.
- Increment the corresponding cell in the Hough accumulator array (hough_space) for valid circle center coordinates.

4-**Thresholding:**

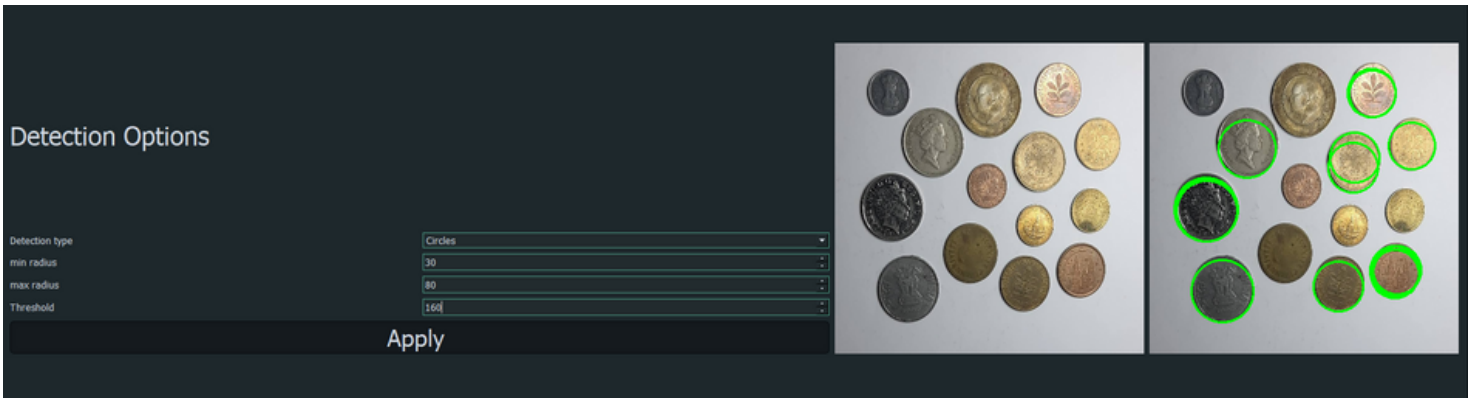
- Identify cells in hough_space that exceed the specified threshold (self.threshold), indicating potential circle centers.

5-**Post-processing:**

- Extract the coordinates of detected circle centers (a, b) and their corresponding radii.
- Adjust the radii values to match the original range (min_radius to max_radius).

6-**Output:**

- Return the detected circle centers (a, b) and radii, representing the detected circles within the image.



fig(3)

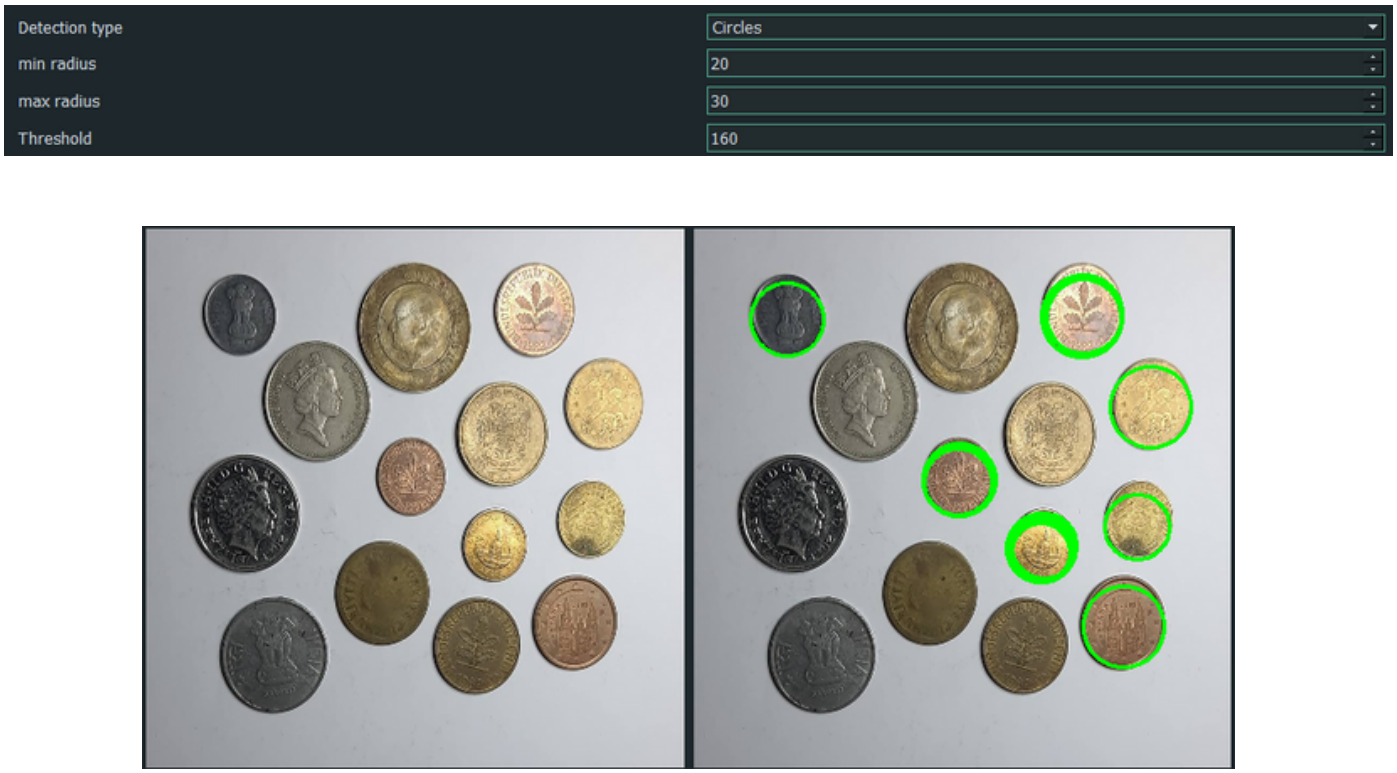
Observations

Upon applying the Hough circle detection algorithm to the image dataset, notable observations have been made regarding the detection of circular objects, particularly coins. The impact of the algorithm's parameters, namely the threshold, minimum radius, and maximum radius, on the number of detected coins is evident.

As illustrated in Figure 3, the application of the algorithm successfully identifies the coins present in the image. However, it is notable that the number of detected coins is influenced not only by the threshold but also by the range of permissible radii. By varying the minimum and maximum radius values, significant changes in the number of detected coins can be observed.

When the range of permissible radii is decreased, as demonstrated in Figure 4, a smaller number of coins are detected. This phenomenon can be attributed to the algorithm's reduced sensitivity to coins of larger radii. Conversely, widening the range of radii allows the algorithm to detect coins of varying sizes, resulting in a higher count of detected coins.

Therefore, it is essential to carefully tune the algorithm's parameters, taking into account both the threshold and the range of permissible radii, to achieve accurate and reliable coin detection results. This observation underscores the importance of parameter optimization in the context of Hough circle detection for coin recognition tasks.



FIG(4)

ELLIPSES DETECTION

Introduction to Hough Ellipses Detection

Ellipse detection is a crucial technique in computer vision used to identify elliptical shapes within an image. Similar to line and circle detection, it operates on the principle of transforming points from the image space to a parameter space where ellipses are represented. This transformation enables robust detection of ellipses amidst noise, occlusions, and variations in ellipse size, orientation, and position.

In this method, each edge point in the image contributes to a set of possible ellipses in the parameter space, typically represented by five parameters: the x-coordinate of the ellipse center (a), the y-coordinate of the ellipse center (b), the major axis length (A), the minor axis length (B), and the orientation angle (θ). These parameters define the position, size, and orientation of candidate ellipses within the image.

The Hough transform accumulates votes in the parameter space for each edge point, forming peaks that correspond to ellipses in the image space. By thresholding these peaks, significant ellipses can be extracted, providing valuable information for tasks such as object detection, feature extraction, and medical image analysis.

Ellipse detection serves as a versatile tool in various applications, including object recognition, robotic navigation, and quality control in manufacturing processes. Its ability to robustly detect elliptical shapes makes it indispensable in many computer vision systems.

```
  def hough_ellipse_detection(self, min_radius_1=60,
max_radius_1=100,min_radius_2=60,max_radius_2=100):
        edges_points = np.nonzero(self.edged_image)
        h, w = self.edged_image.shape
        print(h,w,max_radius_1 - min_radius_1 + 1,max_radius_2 - min_radius_2 + 1)
        hough_space = np.zeros((h, w, max_radius_1 - min_radius_1 + 1,max_radius_2 - min_radius_2 + 1),
dtype=np.uint64)
        x_points, y_points = edges_points

        cos_theta = np.cos(np.deg2rad(np.arange(360)))
        sin_theta = np.sin(np.deg2rad(np.arange(360)))

        for r1 in range(min_radius_1, max_radius_1 + 1):
            for r2 in range(min_radius_2, max_radius_2 + 1):
                for theta in range(360):
                    a = np.round(x_points - r1 * cos_theta[theta]).astype(int)
                    b = np.round(y_points - r2 * sin_theta[theta]).astype(int)
                    valid_indices = np.where((a >= 0) & (a < w) & (b >= 0) & (b < h))
                    a_valid, b_valid = a[valid_indices], b[valid_indices]
                    hough_space[b_valid, a_valid, r1 - min_radius_1,r2 - min_radius_2] += 1

        a, b, radius_1,radius_2 = np.where(hough_space >= self.threshold)
        return a, b, radius_1 + min_radius_1,radius_2 + min_radius_2
```

Our approach

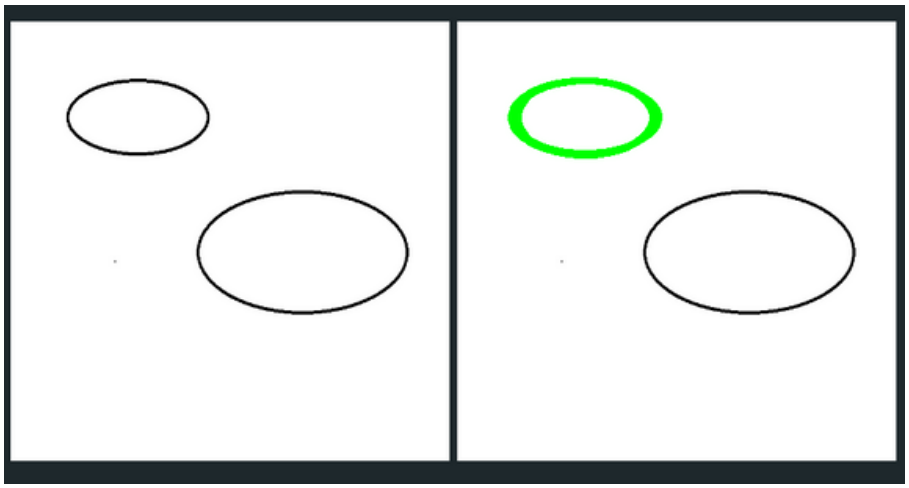
1. **Initialization:**
   - Obtain edge points from the input image using the np.nonzero function.
   - Determine the dimensions of the Hough accumulator array (hough_space) based on the height and width of the input image, along with the specified ranges of radii for the major and minor axes (min_radius_1 to max_radius_1 and min_radius_2 to max_radius_2).
2. **Edge Point Iteration:**
   - Iterate over each edge point in the image, represented by their x and y coordinates (x_points and y_points).
3. **Hough Transform:**
   - For each candidate combination of major and minor radii within the specified ranges:
     - Iterate over 360 degrees of theta values.
     - Compute the corresponding ellipse center coordinates (a, b) using the parametric equation of an ellipse.
     - Increment the corresponding cell in the Hough accumulator array (hough_space) for valid ellipse center coordinates.
4. **Thresholding:**
   - Identify cells in hough_space that exceed the specified threshold (self.threshold), indicating potential ellipse centers.
5. **Post-processing:**
   - Extract the coordinates of detected ellipse centers (a, b) and their corresponding major and minor radii.
   - Adjust the radii values to match the original range (min_radius_1 to max_radius_1 and min_radius_2 to max_radius_2).
6. **Output:**
   - Return the detected ellipse centers (a, b) and radii for the major and minor axes, representing the detected ellipses within the image.

FIG(5)

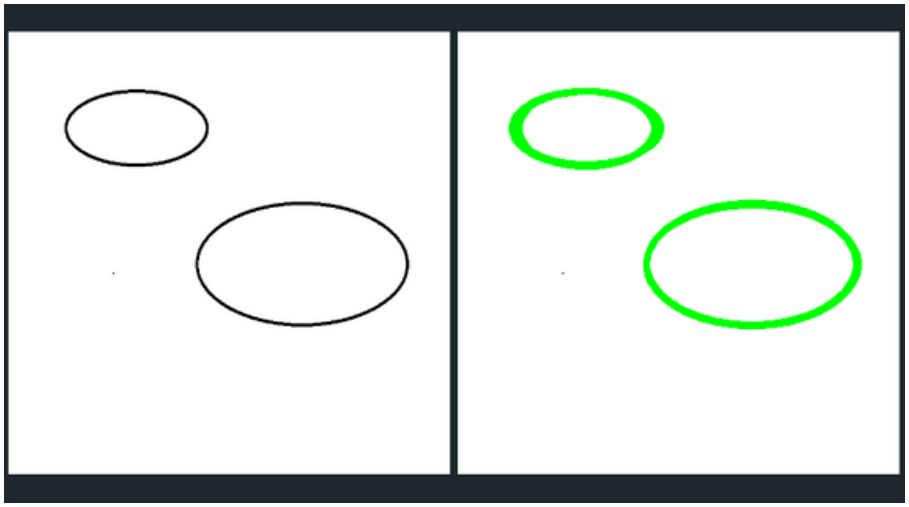
Observations

Upon applying the Hough ellipse detection algorithm to the image dataset, notable observations have been made regarding the detection of elliptical shapes, particularly oval shapes. The algorithm's performance is influenced by multiple parameters, including the range of radii for both the major and minor axes.

As depicted in Figure 5, the algorithm successfully identifies oval shapes within the image. It is worth noting that the number of detected ovals is not solely determined by a single threshold value, but rather by four parameters: the minimum and maximum radii for both the major and minor axes.



FIG(6)



FIG(7)

By adjusting the range of radii, significant changes in the number of detected ovals can be observed. For instance, as demonstrated in Figure 6, narrowing the range of radii results in fewer detected ovals, as the algorithm becomes less sensitive to larger or smaller oval shapes. Conversely, widening the range of radii, as shown in Figure 7, leads to the detection of additional ovals, accommodating a broader spectrum of oval sizes.

Therefore, the algorithm's ability to detect oval shapes is contingent upon careful parameter tuning, particularly the range of radii for both axes. This observation underscores the importance of parameter optimization in Hough ellipse detection for accurate and reliable oval shape recognition tasks.

Upon increasing the range of radii in the Hough ellipse detection algorithm, a longer processing time is observed. Conversely, reducing the range may lead to undetected oval shapes that fall outside the specified range. Thus, there exists a trade-off between computational efficiency and detection accuracy when selecting the range of radii for the algorithm.

ACTIVE CONTOUR MODEL

IIntroduction to Active Contours

Active contours, also known as snakes, are computational models used in image processing and computer vision to detect and delineate object boundaries within images. These models operate by iteratively deforming a parametric curve or surface to minimize an energy function, typically composed of terms related to image features, such as intensity gradients, and internal constraints, such as smoothness and elasticity.

Approach

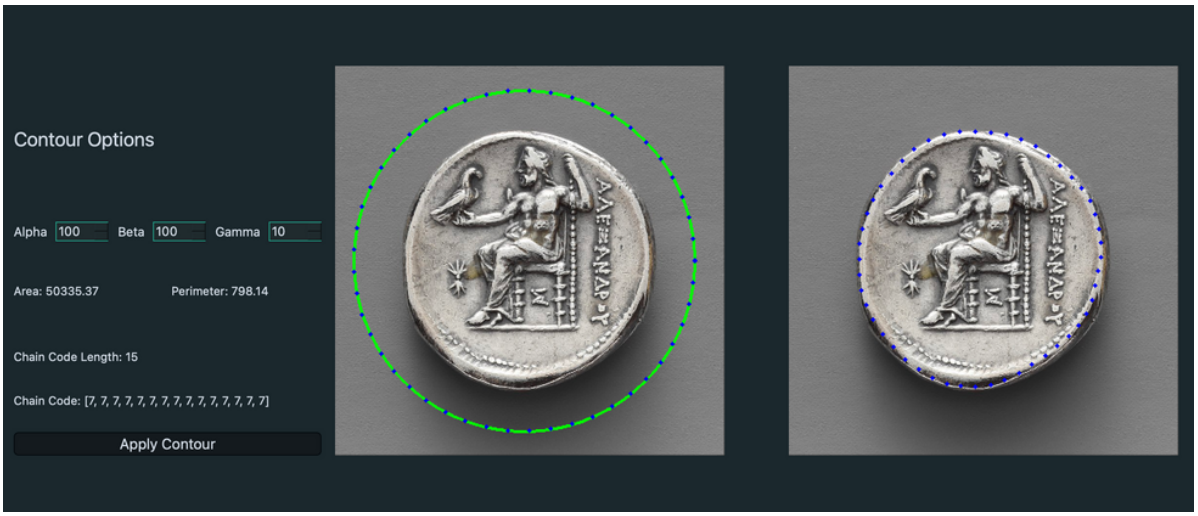
Our implementation of active contours follows a Greedy algorithmic approach, which iteratively updates the snake's coordinates to minimize the energy function. The key steps involved in our approach are as follows:

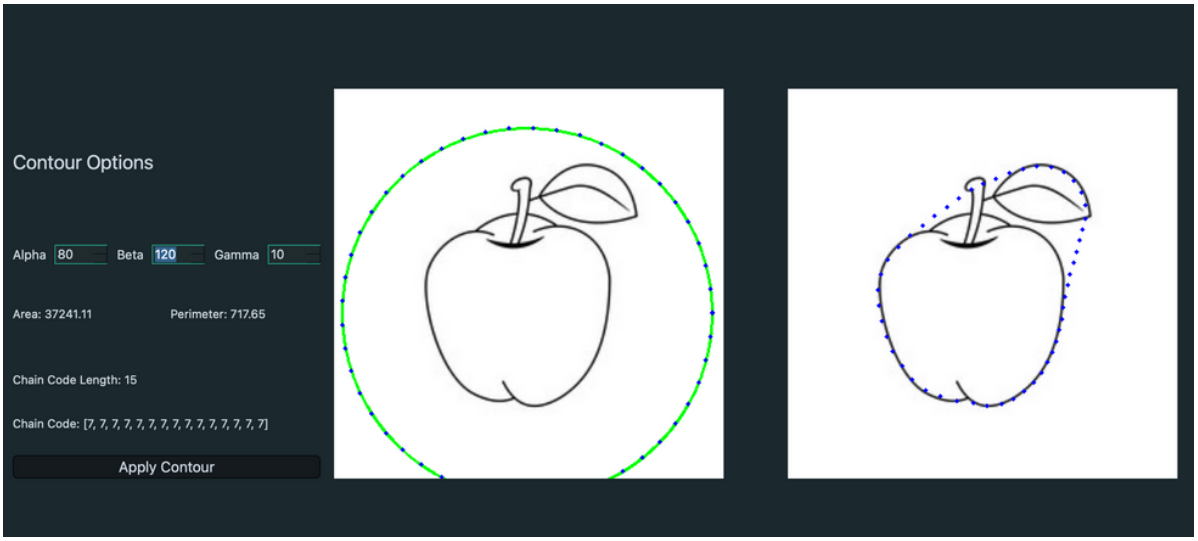
1. **Initialization:**
- The process begins with initializing the snake's initial contour, typically provided by the user or generated using some predefined algorithm.
2. **Preprocessing:**
- Before optimizing the contour, the input image undergoes preprocessing steps, including converting to floating-point representation and possibly enhancing edges using filters like the Sobel operator.
3. **Energy Minimization:**
- The main loop iteratively updates the snake's coordinates to minimize the energy function. The energy function consists of terms related to the image features (edge information) and internal constraints (smoothness and elasticity). The balance between these terms is controlled by parameters such as alpha, beta, w_line, and w_edge.
4. **Gradient Computation:**
- At each iteration, the gradient images along the x and y directions are computed to determine the direction in which the snake should move. This step utilizes interpolation techniques to estimate gradient values at snake control points.
5. **Coordinate Update:**
- The snake's coordinates are updated based on the computed gradients and parameters alpha and beta. These parameters influence the trade-off between edge attraction and contour smoothness.
6. **Convergence:**
- The process continues until a convergence criterion is met, typically based on the change in the snake's position between iterations or the overall energy of the contour.

```
    def ActiveContourGreedy(self, image, snake, alpha=0.01, beta=0.01, w_line=0, w_edge=1, gamma=0.1,
convergence=0.1):
        convergence_order = 16
        max_move = 1.0
        img = img_as_float(image)
        float_dtype = _supported_float_type(image.dtype)
        img = img.astype(float_dtype, copy=False)
        edges = [sobel(img)]
        img = w_line * img + w_edge * edges[0]

        # Interpolate for smoothness:
        interpolated_img = RectBivariateSpline(np.arange(img.shape[1]),
                                               np.arange(img.shape[0]),
                                               img.T, kx=2, ky=2, s=0)

        snake_coords = snake[:, ::-1]
        x_coords = snake_coords[:, 0].astype(float_dtype)
        y_coords = snake_coords[:, 1].astype(float_dtype)
        n = len(x_coords)
        x_prev = np.empty((convergence_order, n), dtype=float_dtype)
        y_prev = np.empty((convergence_order, n), dtype=float_dtype)

        # Explicit time stepping for image energy minimization:
        for i in range(2500):
            # Compute gradient images with the same shape as control points
            gradient_x = interpolated_img(x_coords, y_coords, dx=1, grid=False).astype(float_dtype,
copy=False)
            gradient_y = interpolated_img(x_coords, y_coords, dy=1, grid=False).astype(float_dtype,
copy=False)

            # Update snake coordinates
            x_coords += alpha * gradient_x + beta * (np.roll(x_coords, -1) - x_coords)
            y_coords += alpha * gradient_y + beta * (np.roll(y_coords, -1) - y_coords)

            # Convergence criteria needs to compare to a number of previous configurations since
oscillations can occur.
            j = i % (convergence_order + 1)
            if j < convergence_order:
                x_prev[j, :] = x_coords
                y_prev[j, :] = y_coords
            else:
                dist = np.min(np.max(np.abs(x_prev - x_coords[None, :])
                                     + np.abs(y_prev - y_coords[None, :]), 1))
                if dist < convergence:
                    break

        return np.stack([y_coords, x_coords], axis=1)
```

Parameters:

- **Alpha and Beta:** Control the influence of image features (edges) and internal constraints (smoothness) on the energy function, respectively.
- **w_line and w_edge:** Weights assigned to the image intensity and edge information, respectively, in the preprocessing step.
- **Gamma:** Parameter influencing the implicit spline energy minimization, although not directly utilized in the current implementation.
- **Convergence:** Threshold for determining convergence based on the change in snake coordinates.





Observations:

Upon experimenting with different values of the parameters alpha and beta in the active contour model, two specific cases were examined: one with alpha and beta set to 100, and another with alpha set to 80 and beta set to 120.

In the first case, where both alpha and beta were set to 100, the active contour exhibited a strong preference for following image features (edges), resulting in sharp and well-defined contours. However, the contour smoothness appeared to be slightly compromised, leading to jagged edges in regions with low edge information.

Conversely, in the second case, where alpha was reduced to 80 and beta was increased to 120, the active contour prioritized smoothness over edge detection. This led to contours with smoother transitions but potentially missed or less well-defined edges, particularly in regions with subtle or low-contrast features.

These observations highlight the significant impact of alpha and beta on the behavior and performance of the active contour model. By varying these parameters, it becomes evident that achieving an optimal balance between edge detection and contour smoothness is crucial for accurate boundary delineation.

Further experimentation with different parameter combinations is warranted to fine-tune the active contour model and enhance its accuracy for specific image analysis tasks. and to enhance the accuracy way more, we could explore alternative gradient estimation methods or investigate alternative optimization algorithms for more robust and efficient contour optimization

CHAIN CODE

Introduction to Chain Code

Chain Code is a method employed to describe the boundaries or contours of objects within an image. It serves as a compact and efficient means of representing shapes, particularly in scenarios where contour information is essential, such as in object recognition and shape analysis.

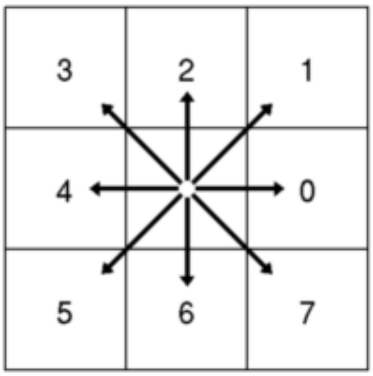
Our approach

1-Pixel Tracing:

Chain Code begins by tracing the boundary of an object within an image. This process involves sequentially following the path of adjacent pixels along the object's contour provided from ActiveContourSnake function by looping over the x and y coordinates of the contour by tracing two points in each iteration starting from first point and the on next to it till the end.

2-Directional Encoding:

At each step of the tracing process, the direction from the current pixel to the next pixel on the object's boundary is determined by comparing the x and y of the current and next pixels. This direction is encoded using a compact scheme, typically based on a predefined set of directional codes representing the eight possible movement directions represented in the following figure (e.g., up, down, left, right, diagonals).



3-Sequential Representation:

The sequence of directional codes generated during the tracing process forms the Chain Code representation of the object's boundary. This sequence captures the contour information in a concise and sequential manner.

```
    def calculate_chain_code(self, contour):
        chain_code = []
        for i in range(len(contour) - 1):
            x1 = contour[i][0]
            y1 = contour[i][1]
            x2 = contour[i + 1][0]
            y2 = contour[i + 1][1]
            if x1 == x2:
                if y1 > y2:
                    for i in range(int(y2), int(y1)):
                        chain_code.append(2)
                else:
                    for i in range(int(y1), int(y2)):
                        chain_code.append(6)
            elif y1 == y2:
                if x1 > x2:
                    for i in range(int(x2), int(x1)):
                        chain_code.append(0)
                else:
                    for i in range(int(x2), int(x1)):
                        chain_code.append(4)
            elif x1 > x2 and y1 > y2:
                for i in range(int(x2), int(x1)):
                    chain_code.append(1)
            elif x1 < x2 and y1 > y2:
                for i in range(int(y2), int(y1)):
                    chain_code.append(3)
            elif x1 < x2 and y1 < y2:
                for i in range(int(x1), int(x2)):
                    chain_code.append(5)
            elif x1 > x2 and y1 < y2:
                for i in range(int(y1), int(y2)):
                    chain_code.append(7)

        print("Length of chain code: ", len(chain_code))
        print("Chain code: ", chain_code)
        # Clip the chain code to 15 elements
        chain_code = chain_code[:15]
        self.chaincode_lbl.setText(f"Chain Code: {chain_code}")
        self.chaincode_len_lbl.setText(f"Chain Code Length: {len(chain_code)}")
```

Our approach

Code calculates the chain code representation of a contour, where each point in the contour is represented by a direction code indicating the direction from the current point to the next point. It iterates through each pair of adjacent points in the contour, determines the direction between them, and appends the corresponding direction code to the chain_code list. The direction codes are assigned based on the relative positions of the points: 0 for right, 1 for up-right, 2 for up, 3 for up-left, 4 for left, 5 for down-left, 6 for down, and 7 for down-right.

AREA AND PERIMETER

Introduction:

The contour, representing the boundary of an object within the image, is obtained through edge detection and contour extraction from ActiveContourSnake function. Once the contour is obtained, we employ mathematical methods to determine its area and perimeter. To calculate the area, we utilize the Green's theorem or the shoelace formula, which involves summing the areas of individual triangles formed by connecting consecutive contour points to a reference point. For perimeter calculation, we iterate through each pair of adjacent points in the contour, computing the Euclidean distance between them, and summing up these distances. The calculated area and perimeter provide valuable quantitative information about the shape and size of the object represented by the contour, essential for various applications such as object recognition, shape analysis, and geometric modeling.

```
def display_area_and_perimeter(self, contour):
        area = 0
        perimeter = 0
        for i in range(len(contour) - 1):
            x1, y1 = contour[i]
            x2, y2 = contour[i + 1]
            area += (x1 * y2 - x2 * y1)
            perimeter += np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

        x1, y1 = contour[-1]
        x2, y2 = contour[0]

        area += (x1 * y2 - x2 * y1)
        perimeter += np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
        area = round(abs(area) / 2, 2)

        perimeter = round(perimeter, 2)

        print("Area: ", area)
        print("Perimeter: ", perimeter)

        # set textlabel with the area and perimters
        self.area_lbl.setText(f"Area: {area}")
        self.perimeter_lbl.setText(f"Perimeter: {perimeter}")
```

AREA AND PERIMETER

Our approach

The code calculates both the area and perimeter of a polygon represented by a contour, which is a sequence of points in the two-dimensional plane. The `display_area_and_perimeter` function takes the contour as input and iterates through each pair of adjacent points.

For the area calculation, the code computes the cross product of vectors from the current point to the next point and from the current point to the origin, summing these cross products. This operation effectively calculates the signed area of the polygon formed by the contour points.

For the perimeter calculation, the code computes the Euclidean distance between consecutive points, summing up these distances.

Since the last point is connected to the first point to form a closed polygon, the code includes this connection in both area and perimeter calculations.

Finally, the code adjusts the area value by dividing it by 2, as in the loop where we calculate the area, we consider each line segment of the polygon, and for each segment, we calculate the area of a parallelogram, which is twice the area of the triangle formed by that line segment and the x-axis.
By summing up these areas, we get a sum of twice the areas of triangles, so we need to divide by 2 to get the total area of the polygon.

The resulting area and perimeter values are then rounded to two decimal places and displayed.

This code provides a straightforward and efficient method for computing both the area and perimeter of a polygonal contour, essential for various image processing and geometric analysis tasks.