



Report of Task 2

Introduction

This report presents the implementation and results of a series of image-processing tasks performed on images, both grayscale and color formats, that serve as the basis for the application of various image analysis techniques.

The tasks implemented fall into two main categories:

1. **Edge and Shape Detection:** This task involves the application of the Canny edge detection algorithm to identify edges within the images. Following edge detection, further analysis is conducted to detect specific shapes present in the images, including lines, circles, and ellipses. The detected shapes are then superimposed onto the original images, providing a visual representation of the shape detection results.
2. **Active Contour Model (Snake) Evolution:** This task involves the initialization of a contour for a given object within an image, and the subsequent evolution of this contour using the Active Contour Model, also known as “snakes”. This model is evolved using a greedy algorithm. The final contour is represented as a chain code, and the perimeter and area enclosed by these contours are computed.

The following sections will detail the methodologies employed for each task, present the results obtained, and discuss the implications of these results. This report aims to provide a comprehensive overview of the image processing tasks performed and to demonstrate the effectiveness of the techniques used in analyzing the given images.

I. Canny Edge Detection

The Canny edge detector aims to **identify edges** in an image. Unlike simple gradient-based methods, Canny provides **high-quality edge localization**. It is robust to noise and minimizes false positives

- The Canny algorithm involves several stages:
 - A. Noise Reduction:** Apply a **Gaussian filter** to smooth the image and reduce noise.
 - B. Gradient Calculation:** Compute the **gradient magnitude** and direction at each pixel by **Sobel**.

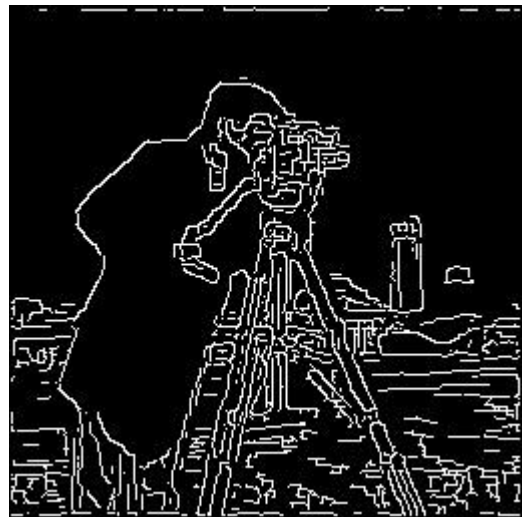
- C. Non-Maximum Suppression:** Suppress non-maximal gradient responses to obtain thin edges.
- D. Double Thresholding:** Apply two thresholds (high and low) to classify pixels as strong, weak, or non-edges.
- E. Edge Tracking by Hysteresis:** Connect strong edges to weak edges if they form continuous curves.

The user can set some parameters in the UI:

- **Kernel size:** The size of the kernel used in **Gaussian smoothing** affects the extent of blurring applied to the image. A larger kernel size results in more **extensive smoothing**, which can reduce noise and detail but may also blur edges.
- **Sigma of the Gaussian filter:** Sigma determines the standard deviation of the Gaussian distribution used in Gaussian smoothing. A larger sigma value leads to more extensive blurring, while a smaller sigma value preserves finer details.



Sigma = 1



Sigma = 50

Observation: for higher sigma value, the image becomes more blurred (more smoothness) and loss of details.

- **Low_threshold_ratio(T_{low}) and High_threshold_ratio(T_{high}):**
 - T_{low} : determines the lower threshold value used in double thresholding. Pixels with gradient magnitudes below this threshold are classified as non-edges unless they are connected to strong edges through weak edges. A **lower value** for the low threshold ratio leads to **more pixels** being classified as **weak edges**, potentially resulting in more edges being detected in the final output.
 - T_{high} : sets the higher threshold value used in double thresholding. Pixels with gradient magnitudes above this threshold are classified as strong edges.

A **higher value** for the high threshold ratio results in **fewer pixels** being classified as **strong edges**, potentially leading to fewer detected edges in the final output.



$T_{\text{low}} = 0.05$ $T_{\text{high}} = 0.09$



$T_{\text{low}} = 0.1$ $T_{\text{high}} = 0.3$

Observation:

- For lower values T_{low} and T_{high} : The weak edges appear clearly.
- For higher values T_{low} and T_{high} : The strongest edges appear clearly and most of the details are lost.

II. Lines Detection

The process of line detection in computer vision involves leveraging techniques such as Canny edge detection and the Hough transform. These techniques form the backbone of robust line detection algorithms and are based on well-established principles in image processing.

Canny Edge Detection:

Canny edge detection is a cornerstone in the field of computer vision for identifying edges in images. It consists of several key steps:

Smoothing for Noise Removal: Initially, the image is smoothed using a Gaussian filter to reduce noise and unwanted details. This step ensures that subsequent edge detection is robust to variations in pixel intensity caused by noise.

Finding Gradients: Next, gradients are calculated to determine the magnitude and direction of intensity changes in the image. This step highlights regions of significant intensity variation, which typically correspond to edges.

Non-maximum Suppression: Following gradient calculation, non-maximum suppression is applied to thin the detected edges to single-pixel width. This process helps preserve only the most significant edge pixels while suppressing weaker ones.

Double Thresholding: Edge pixels are then categorized into strong, weak, and non-edge pixels based on their gradient magnitude relative to predefined thresholds. This step facilitates the identification of potential edge candidates.

Edge Tracking by Hysteresis: Finally, edge tracking by hysteresis connects strong edge pixels to form continuous edge contours. This process ensures that only edges with consistent gradient directions are retained, further refining the detected edges.

Hough Transform for Line Detection:

The Hough transform is a powerful technique used to detect lines in images by identifying points of intersection in the parameter space. In the context of line detection:

Parameterization: Lines in an image are represented using polar coordinates (ρ, θ) , where ρ denotes the perpendicular distance from the origin to the line and θ represents the angle formed by the perpendicular line and the horizontal axis.

Accumulator Array: The Hough transform accumulates votes in an accumulator array based on edge pixels in the image. Each pixel in the edge image contributes to potential lines in the parameter space.

Peak Detection: Local maxima in the accumulator array correspond to lines in the image. The number of peaks detected can be controlled to adjust the sensitivity of line detection.

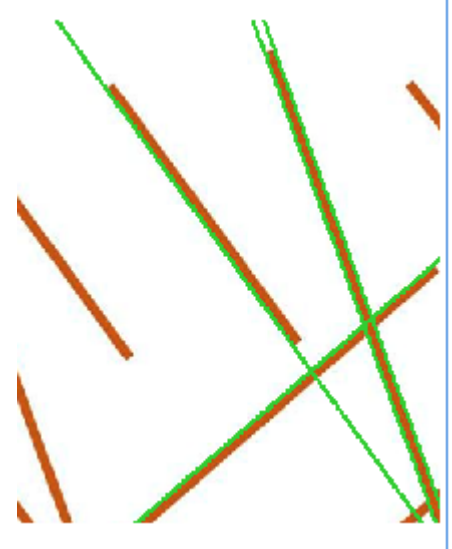
Parameter Variation Analysis for Line Detection

In our experimentation, we systematically explored the effects of varying parameters on the line detection algorithm's performance. Initially, we examined the impact of adjusting the number of peaks (lines) while maintaining a fixed neighboring size of 10. Three scenarios were tested: 4, 27, and 8 peaks. Following this, we focused on the neighboring size parameter, evaluating values of 1, 32, and 7 while fixing the number of peaks at 9. Each trial provided valuable insights into the algorithm's behavior under different conditions, offering a clearer understanding of its adaptability and robustness.

Trial 1: Small Number of Peaks (4)

Observation: The line detection algorithm was applied with a small number of peaks (4), resulting in a reduced number of detected lines compared to the original lines in the image.

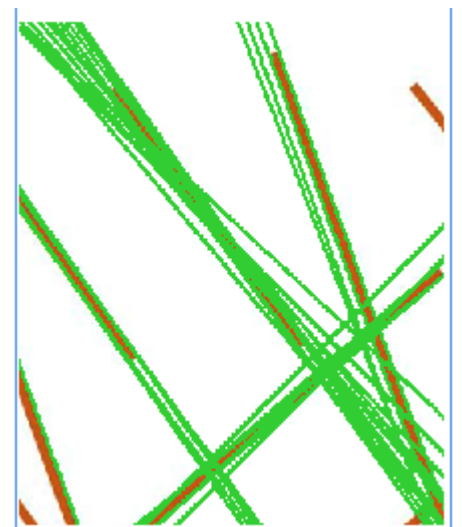
Justification: With only 4 peaks considered, the algorithm exhibited lower sensitivity to lines, leading to the detection of fewer lines overall. This reduced sensitivity may have resulted in the omission of some lines, particularly those with lower prominence or contrast in the image. Additionally, the algorithm's thresholding process may have filtered out potential line candidates, further contributing to the disparity between the detected and original lines. Overall, the observed decrease in detected lines highlights the trade-off between sensitivity and specificity in line detection, emphasizing the importance of selecting appropriate parameter values based on the desired level of line coverage and noise tolerance.



Trial 2: Large Number of Peaks (27)

Observation: The line detection algorithm was applied with a large number of peaks (27), resulting in the detection of one line being represented as multiple lines, leading to a dense cluster of lines plotted over it.

Justification: With 27 peaks considered, the algorithm exhibited high sensitivity to potential lines in the image. However, this heightened sensitivity also increased the likelihood of detecting a single line multiple times, leading to the observed cluster of lines. This phenomenon occurs when a line's edges are detected at multiple points along its length, causing each edge point to contribute to a separate peak in the Hough transform space. As a result, what should be a single line is represented by multiple lines in the output visualization. This underscores the need for parameter tuning to balance sensitivity and specificity, as excessively high sensitivity can result in redundant detections and cluttered visualizations.



Trial 3: Moderate Number of Peaks (8)

Observation: The line detection algorithm was applied with a moderate number of peaks (8), resulting in a balanced representation of lines in the image.

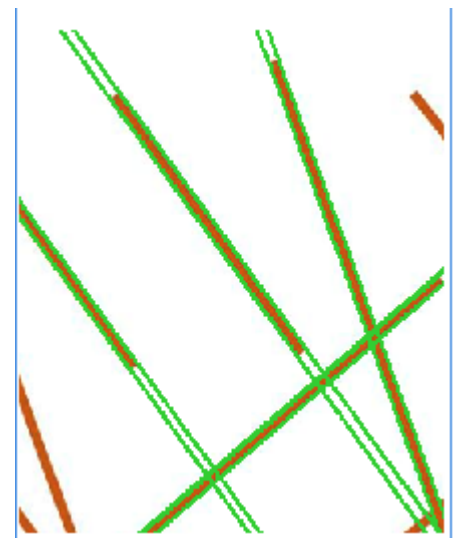
Justification: With 8 peaks considered, the algorithm achieved a balance between sensitivity and specificity in line detection. This moderate sensitivity allowed for the detection of significant features without excessive clutter or redundancy. Unlike the large number of peaks scenario, where single lines were detected multiple times, or the small number of peaks scenario, which resulted in under-detection, the moderate number of peaks provided a satisfactory coverage of lines while avoiding unnecessary clutter. This highlights the importance of selecting appropriate parameter values to achieve optimal line detection performance, striking a balance between sensitivity, specificity, and visualization clarity.



Trial 4: Small Neighboring Size (1)

Observation: The line detection algorithm was applied with a small neighboring size (1), resulting in detected lines that are closely aligned with the original ones in the image.

Justification: With a neighboring size of 1, the algorithm considered only very close neighbors during line detection. This focused approach allowed the algorithm to capture fine details and subtle variations in line orientation and position, resulting in detected lines that closely matched the original ones. By analyzing only immediate neighboring pixels, the algorithm was able to precisely localize edges and extract accurate line representations. However, this level of precision may also increase susceptibility to noise and artifacts, potentially leading to false detections or fragmented lines in more complex images. Overall, the small neighboring size facilitated precise alignment between detected and original lines, highlighting its effectiveness in capturing intricate line structures.



Trial 5: Large Neighboring Size (32)

Observation: The line detection algorithm was applied with a large neighboring size (32), resulting in a more generalized representation of detected lines compared to the original ones in the image.

Justification: With a neighboring size of 32, the algorithm considered a large neighborhood during line detection. This broader scope allowed the algorithm to capture more global features and smooth out local variations, resulting in a more generalized representation of detected lines. However, since we are specifically working on detecting straight lines, the larger neighboring size may not have a significant impact on line curvature or edge smoothness. Instead, it may affect the accuracy and precision of straight-line detection, potentially leading to the merging of adjacent lines or the detection of spurious lines in regions with varying intensity gradients. Overall, while the large neighboring size offers robustness against noise, it may also introduce challenges in accurately delineating straight lines, particularly in complex images with overlapping structures or irregular backgrounds.



III. Circles Detection

Circle detection is a process aimed at identifying circular shapes within an image. It plays a crucial role in various applications such as object recognition, image processing, robotics, and medical imaging.

1. **Pre-processing:** involves applying various techniques to the input image, such as filtering operations like Gaussian blur. These techniques aim to enhance the image quality by reducing noise.
2. **Edge Detection:** it helps in identifying potential boundaries or contours of objects in the image
3. **Generating Circle Candidates:** this includes creating a range of radii and angles to consider for circles and then generating a list of potential circle candidates based on these radii and angles. this is done as follows:
each edge point in the image "votes" for potential circle parameters in a parameter space. The parameters typically include the center coordinates (x_center , y_center) and the radius r of the circle. For each edge point, a set of possible circles is defined by varying the center coordinates and radius. These circle candidates are represented in the parameter space, where each vote contributes to the corresponding circle parameters
4. **Accumulator Array Calculation:**
The votes from the edge points accumulate in an accumulator matrix, which serves as a voting space for potential circles. Each cell in the accumulator matrix corresponds to a set of circle parameters (x_center , y_center , r). The accumulator matrix effectively accumulates votes for different circle parameters, with higher vote counts indicating a higher likelihood of a circle's presence at that location and with that radius.
5. **Post-processing:** The post-processing step is performed to refine the detected circles by removing duplicates or filtering out unwanted detections. This is done by applying a pixel threshold. The pixel threshold is a parameter set based on factors such as the image resolution, the expected size of circles in the image, and the level of noise present.
A smaller pixel_threshold value filters out circles that are very close to each other. While this can reduce the risk of false positives (incorrectly detected circles), it also risks removing valid circles that are naturally close together. Therefore, selecting an appropriate pixel threshold is crucial to balance between reducing false positives and retaining valid circle detections.

parameters:

The parameters used in the hough circle are :

1. r_min and r_max : determine the size range of circles that will be detected.
2. Δr : This parameter specifies the step size or increment for exploring different radii within the range defined by minimum radius and maximum radius.
3. num_thetas : determines the number of angles (or the resolution of angle space) to consider for circle detection using the Hough transform.

4. **bin_threshold:** This parameter sets a threshold for considering a detected circle based on the number of votes it receives during the Hough transform. It determines the minimum level of support a circle must have to be considered a valid detection.
5. **pixel_threshold:** This parameter is used in post-processing to filter out circles that are too close to each other. It helps eliminate redundant or overlapping detections.

The r_{min} , r_{max} , $bin_threshold$, and $pixel_threshold$ are changed via a slider so that users can observe their effect.

Test Case 1: Narrow Range:

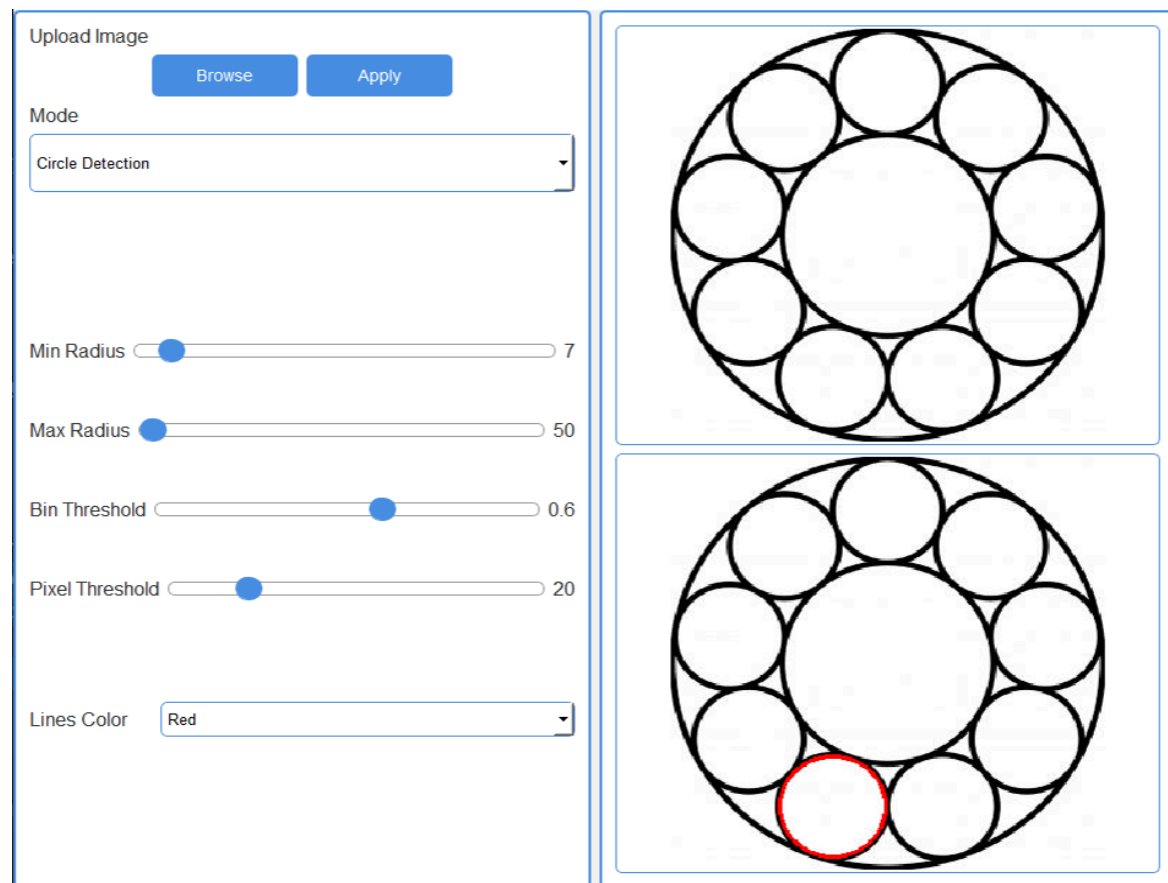
Set r_{min} to a small value (e.g., 7 pixels) and r_{max} to a larger value (e.g., 50 pixels).

Observations:

With a narrow range, the algorithm primarily detects small-sized circles.

The number of detected circles is relatively low compared to wider ranges.

the detected circle has a radius within the specified range, with few outliers.



Test Case 2: Wide Range:

Slider Settings: Set r_{min} to a larger value (e.g., 20 pixels) and r_{max} to a significantly larger value (e.g., 200 pixels).

Observations:

With a wide range, the algorithm detects circles spanning a broad range of sizes.

Upload Image

Browse

Apply

Mode

Circle Detection

Min Radius

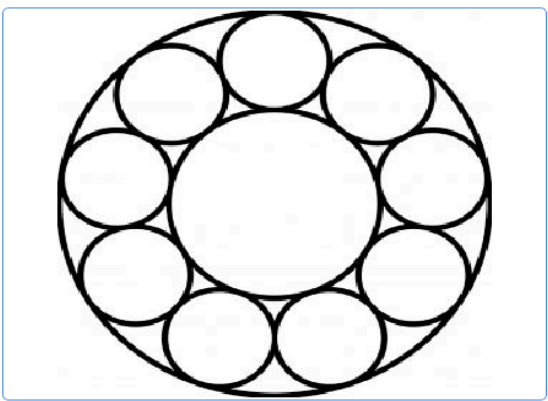
Max Radius

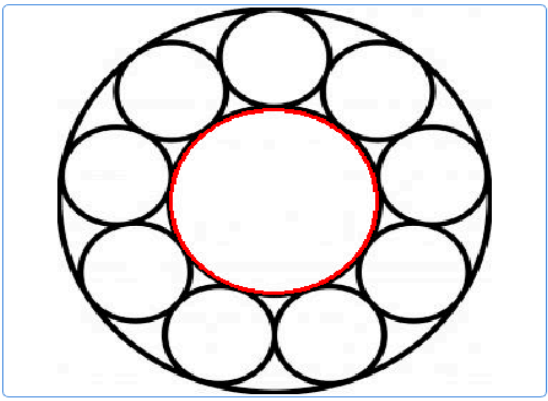
Bin Threshold

Pixel Threshold

Lines Color

Red





Upload Image

Browse

Apply

Mode

Circle Detection

☐ Square Contour

☐ Circle Contour

Min Radius

Max Radius

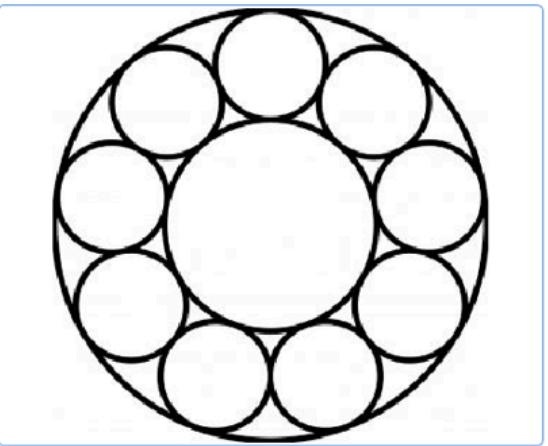
Bin Threshold

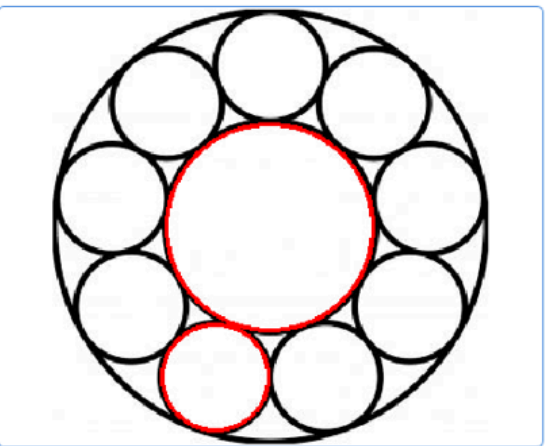
Pixel Threshold

Lines Color

T_high

T_low





Test Case 3: Bin Threshold:

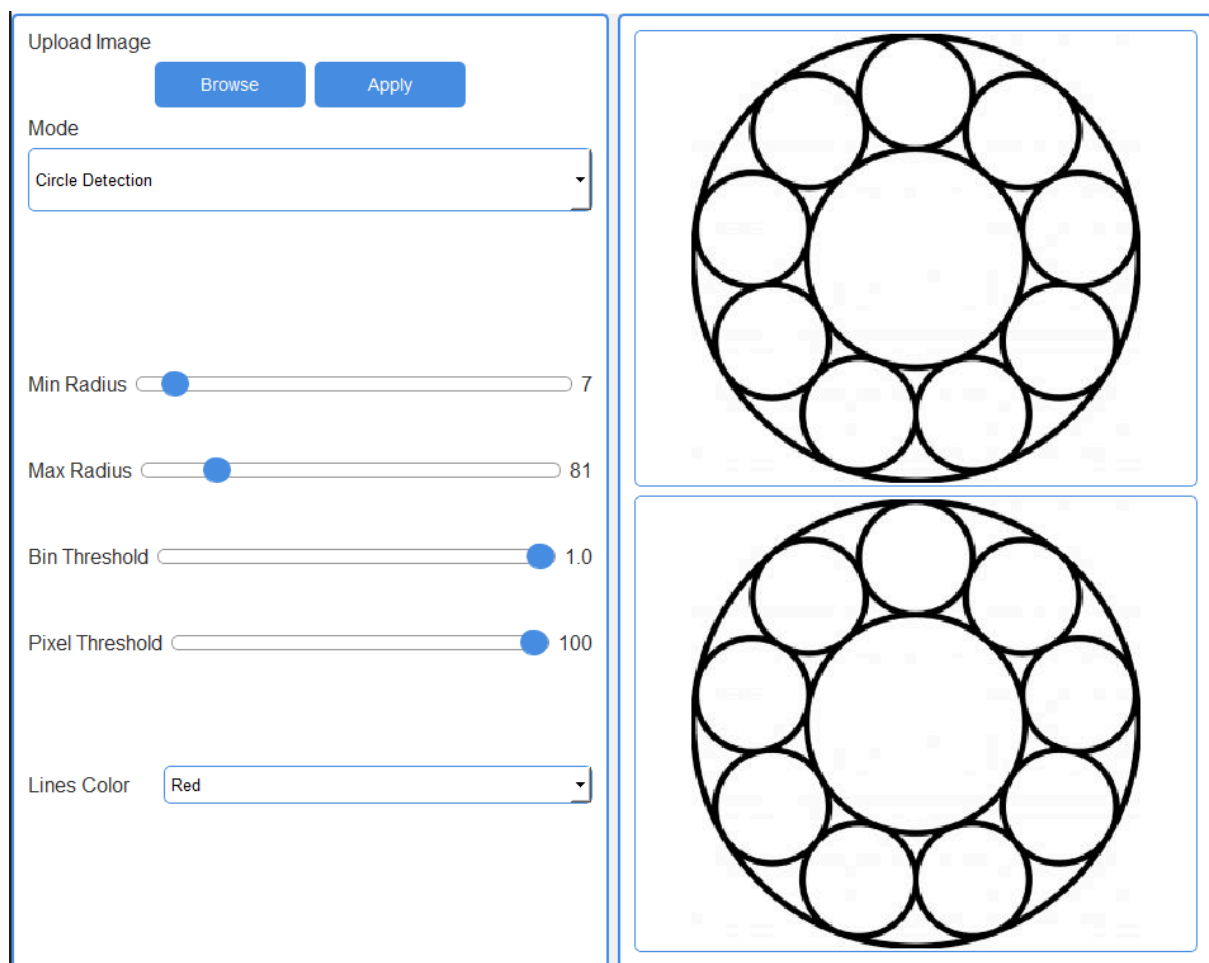
A low bin threshold increases the sensitivity to circle candidates, resulting in a higher detection rate.

The higher the bin threshold, the fewer circles are detected

With a bin_threshold set to 1, this imposes an extremely stringent criterion for circle detection.

The algorithm only considers circle candidates that receive the maximum number of votes in the accumulator.

This means that any circle candidate that is even slightly less supported than others is excluded from detection.



Here no circles are detected as all circles are excluded from detection because they have lower votes.

IV. Ellipses Detection

Ellipse detection is a computer vision technique used to identify ellipses with an image. It is particularly used in applications such as object recognition and medical imaging. Ellipse detection algorithms analyze the shape and intensity distribution of pixels in an image to locate ellipses that may represent objects, patterns, or features of interest. These algorithms can be based on techniques like the Hough Transform.

This technique has seven steps:

1. Edge detection:
Apply an edge detection algorithm, such as a Canny edge detector, to the input image after converting it to grayscale to detect edges.
2. Parameter Initialization:
Parameters such as the minimum and maximum size of ellipses to detect, and the threshold for voting.
3. Accumulator Initialization:
An accumulator array is initialized to store votes for potential ellipses. The dimensions of the accumulator array are the same as the grayscale image but with an additional dimension for the range of ellipse sizes (from min_size to max_size).
4. Voting:
For each edge pixel in the Canny edge image, a vote is cast in the accumulator array for possible ellipses passing through that pixel. This is done by iterating over possible ellipse sizes (**a and b**) and orientations (**theta**).
5. Peak Detection:
After all edge pixels have voted, peaks in the accumulator array are identified. A **threshold** is applied to determine which peaks correspond to significant votes for ellipses.
6. Ellipse Extraction:
For each significant peak in the accumulator array, the corresponding ellipse parameters (x, y, a, b) are extracted.
7. Return Ellipses: Finally, the detected ellipses are returned as a list of tuples containing their center coordinates (x, y) and major and minor axes lengths (a, b).

- **Min_size:**

This parameter sets the minimum size of the ellipses that the algorithm will attempt to detect. It determines the smallest ellipse that the algorithm will consider during the detection process. Smaller values of min_size allow the algorithm to detect smaller ellipses in the image.

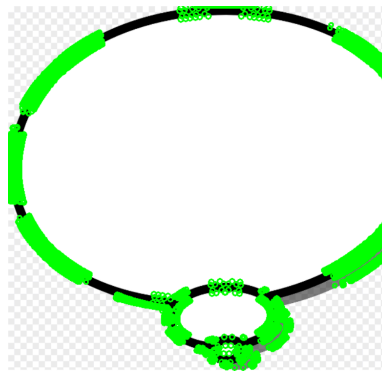
Conversely, larger values of min_size will exclude smaller ellipses from consideration.

If you decrease min_size: The algorithm will be more sensitive to smaller ellipses. It will attempt to detect even smaller ellipses in the image.

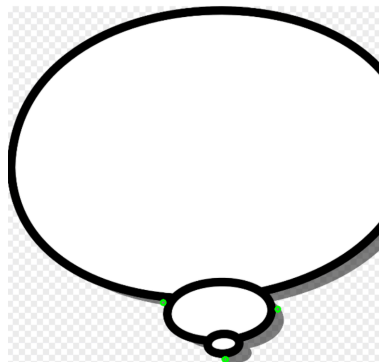
If you increase min_size: The algorithm will become less sensitive to smaller ellipses. It will only consider larger ellipses during the detection process, potentially excluding smaller ellipses from detection.

Trial :

- Decrease min_size



- Increase min_size



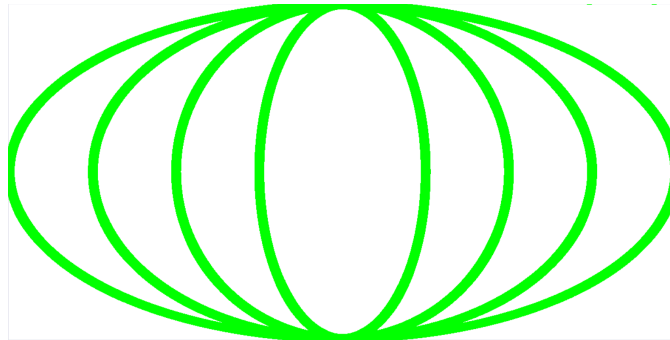
- Max_size:
This parameter sets the maximum size of the ellipses that the algorithm will attempt to detect. It determines the largest ellipse that the algorithm will consider during the detection process. Larger values of max_size allow the algorithm to detect larger ellipses in the image. Conversely, smaller values of max_size will exclude larger ellipses from consideration.

If we decrease max_size: The algorithm will be more sensitive to smaller ellipses. It will attempt to detect even smaller ellipses in the image.

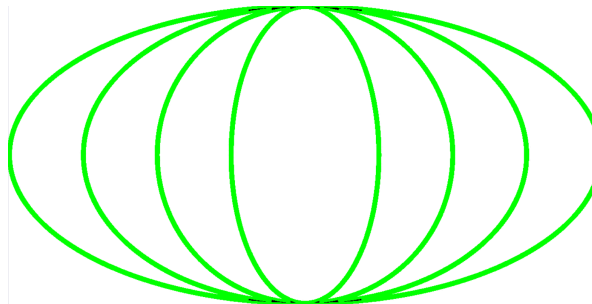
If you increase max_size: The algorithm will become less sensitive to smaller ellipses. It will only consider larger ellipses during the detection process, potentially excluding smaller ellipses from detection.

Trial :

- increase max_size



- Decrease max_size



Effect: there are empty parts in the large ellipse.

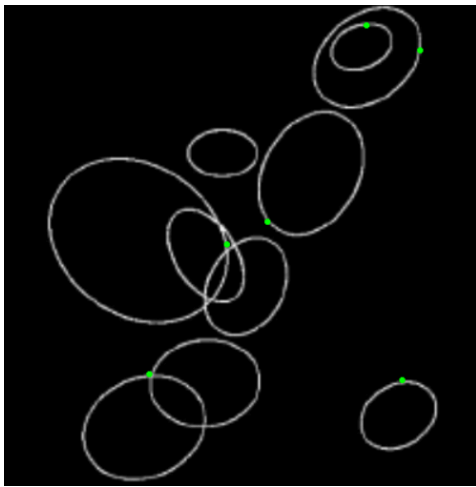
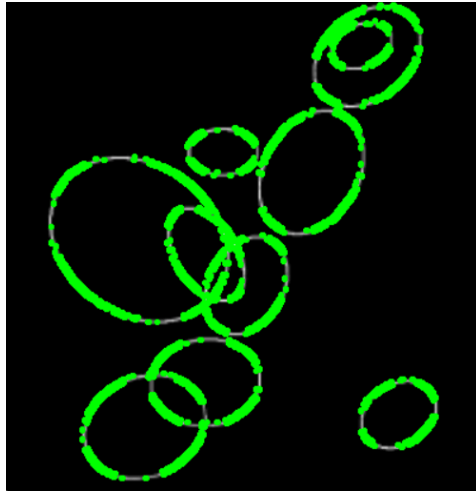
- **Threshold:**

This variable represents the threshold for voting. During the Hough Transform process, accumulator cells accumulate votes corresponding to potential ellipse parameters. This threshold determines the minimum number of votes required for a set of parameters to be considered as representing a detected ellipse.

Trial :

- increase threshold :

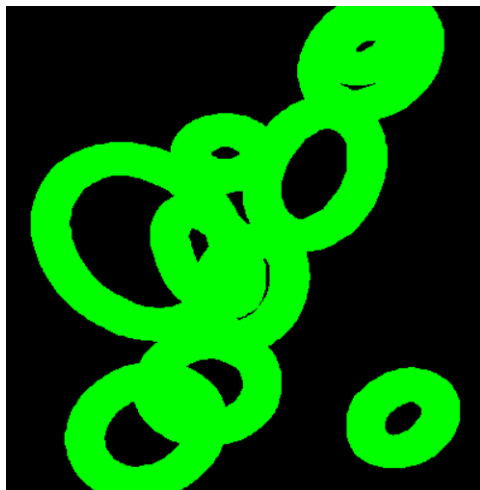
If you increase the threshold, you are raising the minimum number of votes required for a set of parameters to be considered as representing a detected ellipse. This makes the detection process more strict.



Effect: More ellipses are likely to be detected, including those with fewer supporting votes. This can lead to a higher chance of false positives, where non-elliptical shapes or noise in the image might be detected as ellipses.

- Decrease threshold:

If you decrease the threshold, you are lowering the minimum number of votes required for a set of parameters to be considered as representing a detected ellipse. This makes the detection process less strict.



Effect: More ellipses are likely to be detected, including those with fewer supporting votes. This can lead to a higher chance of false positives, where non-elliptical shapes or noise in the image might be detected as ellipses.

V. Active Contour Model (Snakes)

The Active Contour Model, also known as "Snakes", is a powerful technique used in image processing for tasks such as object tracking and segmentation. The model works by iteratively adjusting a contour to fit the edges in an image, based on an energy minimization process. This report presents a detailed implementation of the Active Contour Model.

The implementation begins by creating an initial contour on the image. The shape of this contour (either a square or an ellipse) is determined by the user's choice. This initial contour is then displayed on the Graphical User Interface (GUI).

The contour's position is optimized based on internal and external energies. The internal energy, calculated using the `'calculate_internal_energy'` function, is a measure of the contour's continuity and smoothness, which helps to maintain the contour's shape during the iterative process of the model. The external energy, calculated using the `'calculate_external_energy'` function, is a measure of the image features, such as edges, that the contour is trying to fit.

In each iteration of the model, every point on the contour is potentially moved to a new location within a neighborhood window that minimizes the energy function. The new location for each point is chosen from a set of candidate locations in the vicinity of the original point. This set of candidate locations, or the "neighborhood window", is generated using the `'generate_window_coordinates'` function.

The contour iteratively adjusts its position to better fit the edges in the image, based on this energy minimization process. The result is a contour that accurately represents the boundaries of the object of interest in the image.

1. `active_contour_model(self)`: This function applies the Active Contour Model (Snake) to an image. It reads an image, creates an initial square contour, calculates the external energy of the image, and then applies the active contour model for a chosen number of iterations. The initial and final contours are plotted on the image.
 - a. It first retrieves the contour parameters from the GUI sliders.
 - b. Then it sets the number of points for the ellipse contour and the square contour.
 - c. It starts a timer to calculate the function run time.
 - d. It creates an initial contour based on the user's choice (square or ellipse) and displays it on the GUI.
 - e. It calculates the external energy that will be used in each iteration of the greedy algorithm.
 - f. It applies the active contour algorithm for the specified number of iterations, updating the contour coordinates and displaying the updated contour after each iteration.

- g. Finally, it calculates the chain code for the final contour, the contour perimeter, and the contour area, and prints the elapsed time.
2. `create_square_contour(image: np.ndarray, num_x_points: int, num_y_points: int) -> Tuple[np.ndarray, np.ndarray, list]`: This function creates a square contour and a neighborhood window around each point on the contour. It returns the x and y coordinates of the contour and the coordinates of the neighborhood window.
 - a. It first creates x points lists for each side of the square (`top_x`, `right_x`, `bottom_x`, `left_x`) and y points lists for each side of the square (`top_y`, `right_y`, `bottom_y`, `left_y`).
 - b. It concatenates all the lists in one array for the x and y coordinates of the contour (`contour_x`, `contour_y`).
 - c. It shifts the shape to a specific location in the image.
 - d. It creates a neighborhood window around each point on the contour using `generate_window_coordinates(5)`.
 - e. It returns the x and y coordinates of the contour and the coordinates of the neighborhood window.
 3. `create_ellipse_contour(image: np.ndarray, num_points: int) -> Tuple[np.ndarray, np.ndarray, list]`: This function creates an ellipse contour and a neighborhood window around each point on the contour. It returns the x and y coordinates of the contour and the coordinates of the neighborhood window.
 - a. It first creates x and y lists coordinates to initialize the contour (`contour_x`, `contour_y`). The coordinates are generated using the parametric equations of an ellipse, with the number of points specified by `num_points`.
 - b. It converts the coordinates to integer type.
 - c. It creates a neighborhood window around each point on the contour using `generate_window_coordinates(5)`.
 - d. It returns the x and y coordinates of the contour and the coordinates of the neighborhood window.
 4. `draw_contour_on_image(image: np.ndarray, points_x: np.ndarray, points_y: np.ndarray) -> np.ndarray`: This function draws a contour on an image. It takes as input the source image and the x and y coordinates of the contour points. It returns the image with the contour drawn on it.
 - a. It first makes a copy of the image to prevent modifying the original image.
 - b. It combines the x and y coordinates into a list of points and reshapes the points array to the required format for `cv2.polylines`.
 - c. It checks if the image is grayscale or color and sets the contour color accordingly.
 - d. It draws the contour on the image and returns the image with the contour drawn on it.

5. `calculate_external_energy(image, w_line, w_edge)`: This function calculates the external energy of an image.
 - a. It first converts the image to grayscale if it's not already in grayscale.
 - b. It applies a Gaussian Filter to smooth the image.
 - c. It gets the Gradient Magnitude & Direction.
 - d. It returns the sum of the line energy and the edge energy.
6. `sobel_edge(source: np.ndarray, GetMagnitude: bool = True, GetDirection: bool = False)`: This function applies the Sobel operator to an image to detect edges.
 - a. It first defines the horizontal and vertical Sobel kernels.
 - b. It applies these kernels to the image using the `apply_kernel` function.
 - c. If `GetMagnitude` is set to `False`, the function returns the horizontal and vertical edges.
 - d. If `GetDirection` is set to `True`, the function also calculates and returns the direction of the gradient.
7. `iterate_contour(image: np.ndarray, contour_x: np.ndarray, contour_y: np.ndarray, external_energy: np.ndarray, window_coordinates: list, alpha: float, beta: float) -> Tuple[np.ndarray, np.ndarray]`: This function iterates over a contour and optimizes its position based on internal and external energies. It returns the optimized x and y coordinates of the contour.
 - a. It first makes a copy of the image and contour coordinates.
 - b. It then iterates over each point in the contour. For each point, it iterates over each coordinate in the window around the original position and computes the energy at each new position. The point is then moved to the position with the lowest energy.
 - c. It returns the optimized x and y coordinates of the contour.

- **Chain code**

The chain code is a compact representation used in image processing and computer vision to describe the shape of a contour. It simplifies the contour by encoding the direction of transitions between adjacent points along the contour.

How it works:

1. **Contour Points:** A contour is a sequence of points that form a closed curve (such as the outline of an object). These points are typically obtained from edge detection or other contour extraction techniques.
2. **Calculating Differences:** For each pair of adjacent points in the contour, we calculate the difference in their x and y coordinates
3. **Assigning Chain Codes:** Based on the differences between dx and dy, we assign a chain code to represent the transition from one point to the next

- **4-Connected Chain Code:**
 - In the 4-connected chain code, we use a set of four directions to represent transitions between adjacent points in a contour.
 - The four directions are:
 - 0: Step to the right
 - 1: Step up
 - 2: Step to the left
 - 3: Step down
 - This chain code is commonly used for representing contours in a concise manner.
- **8-Connected Chain Code:**
 - The 8-connected chain code extends the concept to include **eight directions** for transitions.
 - The eight directions are:
 - 0: Step to the right
 - 1: Step right up (diagonal)
 - 2: Step up
 - 3: Step left-up (diagonal)
 - 4: Step to the left
 - 5: Step left-down (diagonal)
 - 6: Step down
 - 7: Step right-down (diagonal)
 - This chain code provides more detailed information about the contour's shape and orientation.
- **The perimeter of the contour**

The perimeter of the contour refers to the total length of its boundary. It represents the distance around the shape.

How the perimeter of a contour is calculated:

1. Contour Points:

- A contour is a sequence of points that form a closed curve, such as the outline of an object.
- These points are typically obtained from edge detection or other contour extraction techniques.

2. Euclidean Distance:

- For each pair of adjacent points in the contour, we calculate the Euclidean distance between them.

3. The sum of Distances:

- We compute the Euclidean distance for each adjacent pair of points along the contour.
- The total perimeter is the sum of all these distances.

4. Wrap Around:

- Since the contour is closed (i.e., it starts and ends at the same point), we wrap around to the first point when calculating distances.
- This ensures that the last point connects back to the starting point.

- **The area of the contour**

The area of a contour represents the total surface area enclosed by the contour. It is a fundamental geometric property used in various fields such as computer vision, image processing, and geometry.

How the area of a contour is calculated:

1. Contour Points:

- A contour is a sequence of points that form a closed curve, typically representing the boundary of an object or shape.
- These points are obtained from edge detection or other contour extraction techniques.

2. Shoelace Formula:

- The shoelace formula (also known as the polygon area formula) is used to compute the area of a simple polygon defined by its contour points.
- It derives its name from the way shoelaces are tied, as it involves calculating the “crossing” areas.

3. Algorithm:

- Given a set of contour points $((x_1, y_1), (x_2, y_2), \dots, (x_n, y_n))$, where (n) is the total number of points:

$$\text{Area} = \frac{1}{2} \left| \sum_{i=1}^n x_i y_{i+1} - x_{i+1} y_i \right| *$$

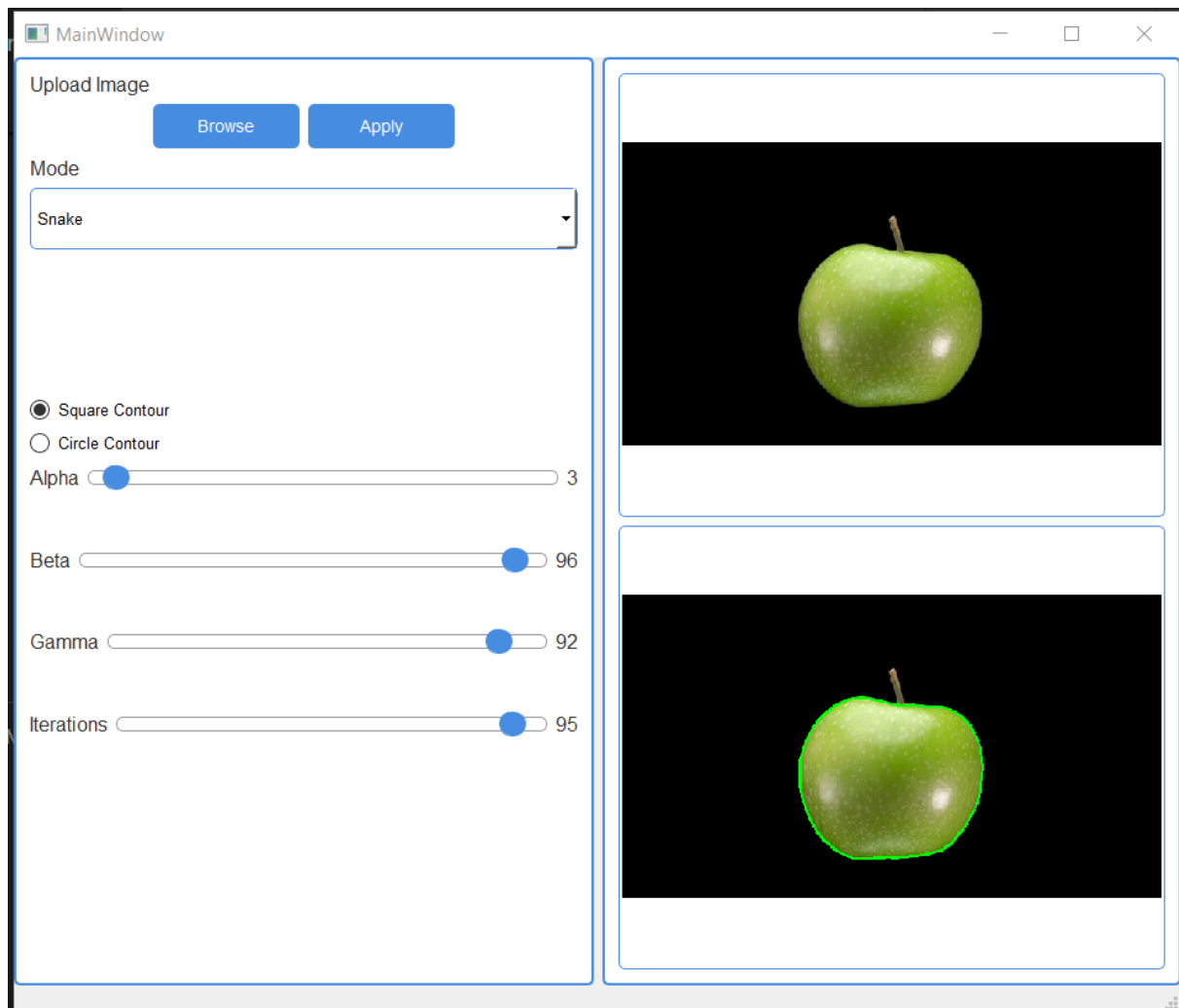
$$\text{Area} = \frac{1}{2} |x_1 y_2 - x_2 y_1 + x_2 y_3 - x_3 y_2 + \dots - \dots + x_{n-1} y_n - x_n y_{n-1} + x_n y_1 - x_1 y_n|$$

* where $x_{n+1} = x_1$ and $y_{n+1} = y_1$

The Shoelace Formula

- The summation involves pairs of adjacent points, and the result is divided by 2 to obtain the positive absolute value of half the accumulated area.

Results

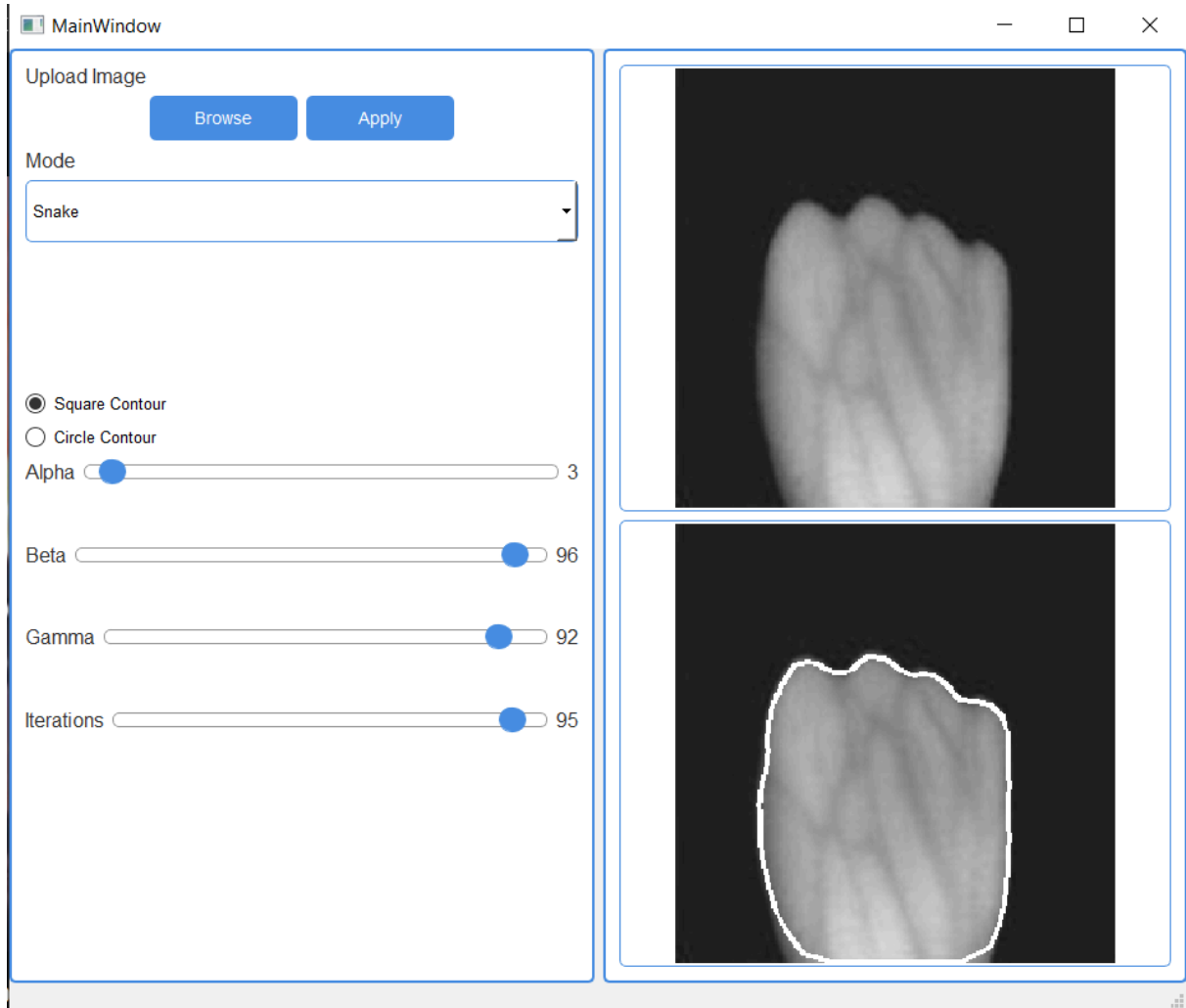


chain code 8: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 7, 7, 7, 7, 7, 0, 7, 0, 7, 0, 1, 7, 0, 0, 7, 0, 0, 0, 7,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 4, 5, 4, 4, 5, 4, 4, 5, 4, 3, 5, 4, 4, 5, 4, 4, 4, 4, 3, 7,
0, 4, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1]

chain code 4: [0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 1, 1, 1, 1, 1]

Contour perimeter: 625.86

Contour area: 28925.50 square units

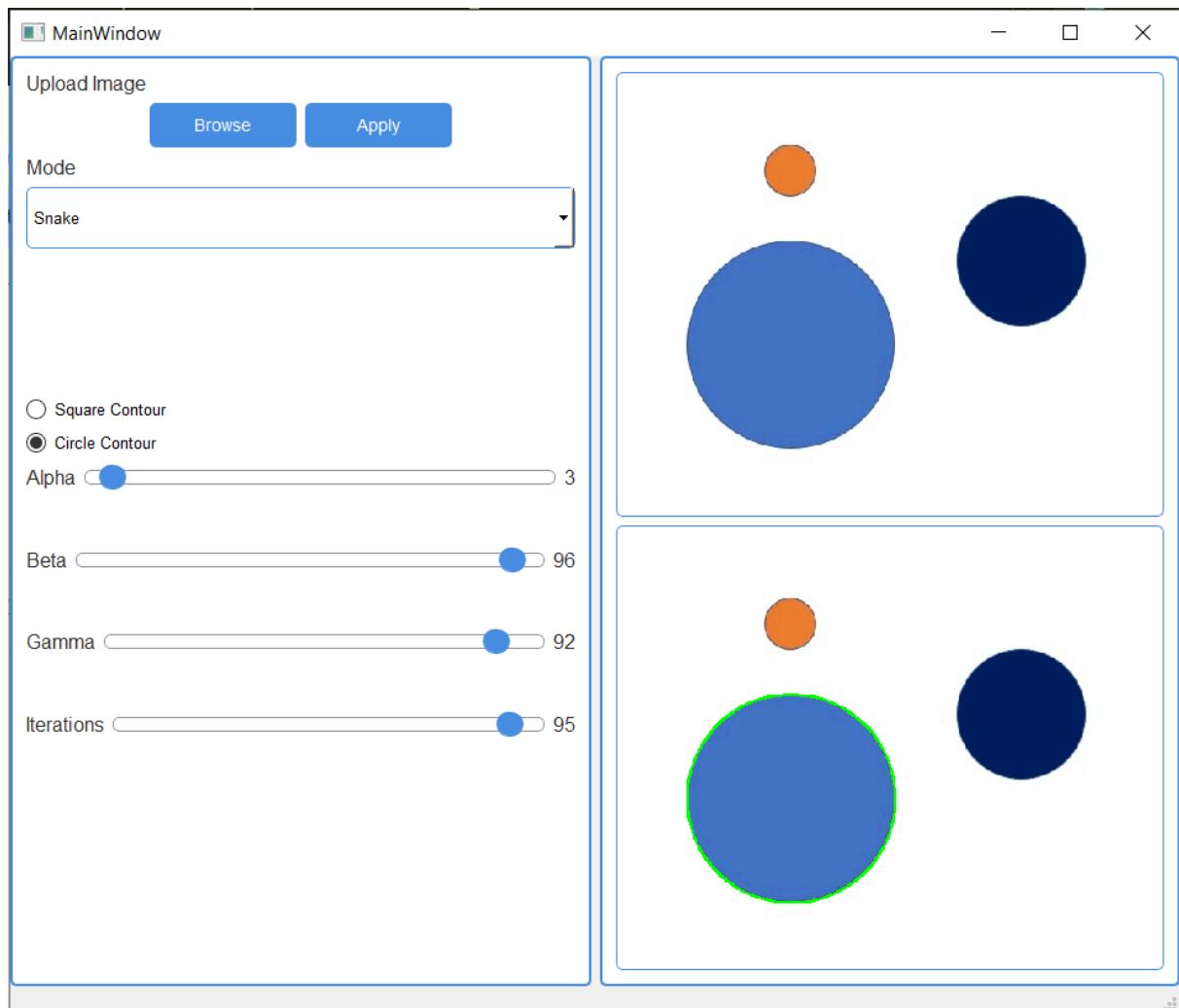


chain code 8: [2, 1, 5, 1, 1, 1, 1, 1, 1, 7, 7, 7, 0, 1, 1, 1, 7, 7, 7, 7, 7, 1, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 3, 2, 3, 7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 7, 5, 6, 6, 6, 7, 6, 6, 5, 6, 6, 5, 5, 5, 5, 5, 4, 4, 4, 0, 4, 0, 4, 4, 4, 4, 4, 0, 4, 4, 0, 0, 4, 4, 0, 0, 4, 4, 0, 0, 4, 4, 4, 4, 4, 0, 0, 4, 4, 0, 0, 4, 4, 0, 0, 4, 4, 0, 0, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 3, 6, 1, 2, 1, 1, 1, 2, 1, 3, 2, 1, 2, 1, 2, 1, 1, 1]

chain code 4: [1, 0, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 0, 2, 0, 2, 2, 2, 2, 2, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 2, 2, 2, 1, 1, 1, 3, 1, 1, 1, 1, 1]

Contour perimeter: 665.76

Contour area: 22442.50 square units



chain code 8: [6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 6, 6, 2]

chain code 4: [3, 2, 2, 2, 1, 1, 0, 3, 3, 1]

Contour perimeter: 663.46

Contour area: 33908.00 square units