



Computer Vision

Task 2

Team 18

Ahmed Kamal Mohamed	Sec.1
Amgad Atef Abd Al-Hakeem	Sec.1
Mahmoud Mohamed Ali	Sec.2
Mahmoud Magdy Mohamed	Sec.2
Mohanad Emad El-Sayed	Sec.2

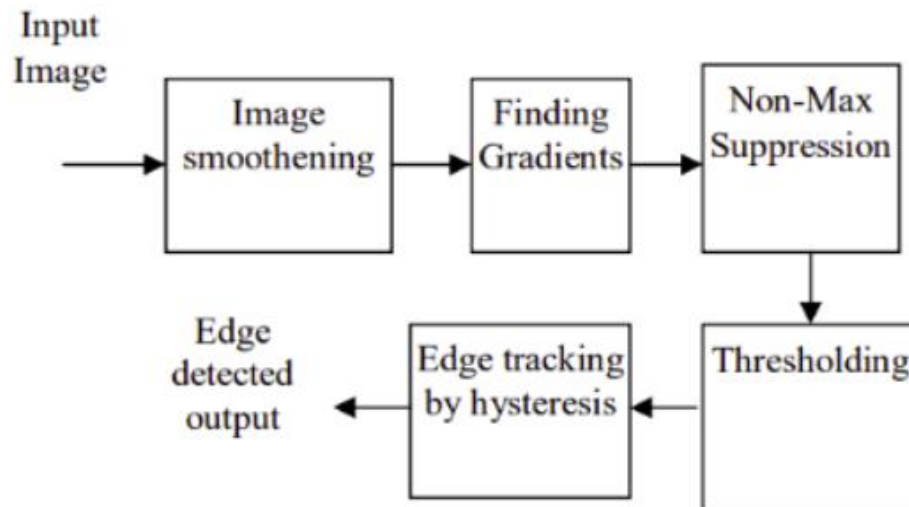
Supervised By
Dr. Ahmed Badawi
Eng. Lila Abbas
Eng. Omar Hesham

Table of Contents

1. Canny detector	3
2. Hough transform	8
2.1 Hough lines	9
2.2 Hough circles	14
2.3 Hough ellipses	18
3. Active contour	21
3.1 External Energy	22
3.2 Internal Energy	22
3.3 Balloon Energy	23
3.4 Contour perimeter	25
3.5 Contour area	25
3.6 Chain code	26

1. Canny Edge Detection:

The Canny edge detection algorithm is a multi-stage process involving Gaussian smoothing, gradient calculation, non-maximum suppression, and hysteresis thresholding. It is one of the most effective edge detection techniques, providing good detection while reducing noise.



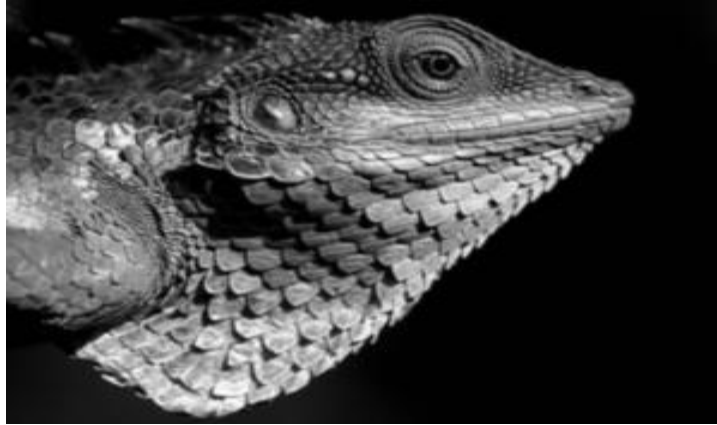
Steps to apply the Canny detector:

1. Original image:



2. Gray scaled image:

$$\text{Intensity} = 0.299 \times \text{Red} + 0.587 \times \text{Green} + 0.114 \times \text{Blue}$$



3. Gaussian Smoothing:

- we convert the image to grayscale as edge detection does not depend on colors. Then we remove the noise in the image with a Gaussian filter as edge detection is prone to noise.

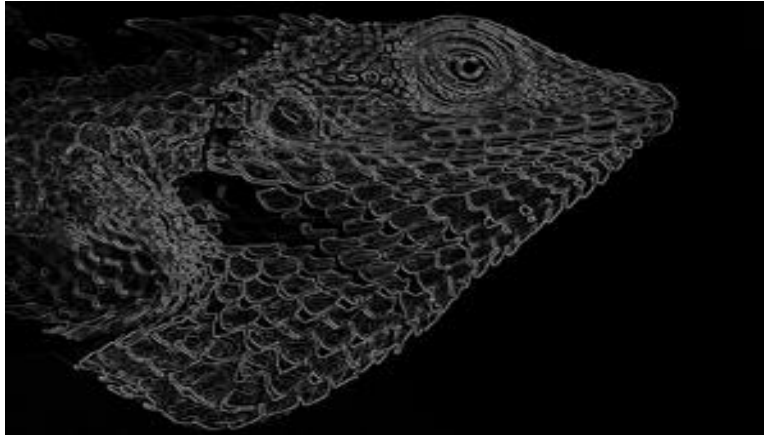
4. Gradient Calculation:

- Compute the gradient magnitude and direction using Sobel operators.
- We know that the gradient direction is perpendicular to the edge. We round the angle to one of four angles representing vertical, horizontal, and two diagonal directions.



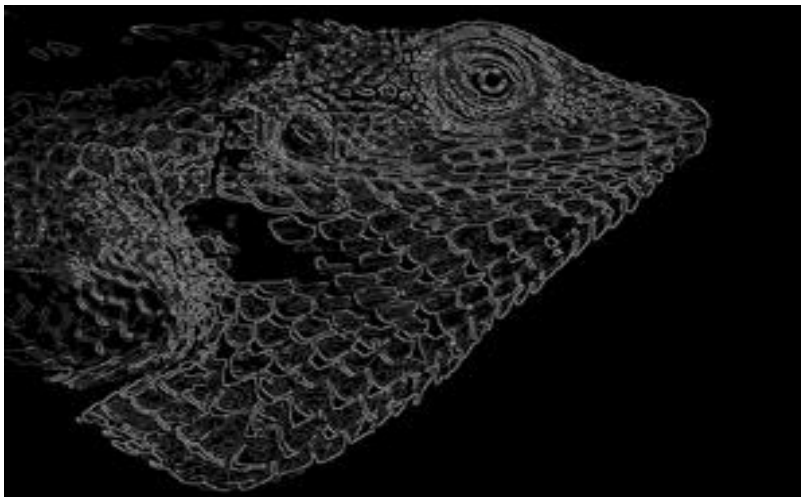
5. Non-maximum Suppression:

- Suppress non-maximum pixels to thin the edges:
- Compare pixel gradient magnitudes with neighboring pixels along the gradient direction.
- Keep pixels with local maximum gradient values, set others to zero.



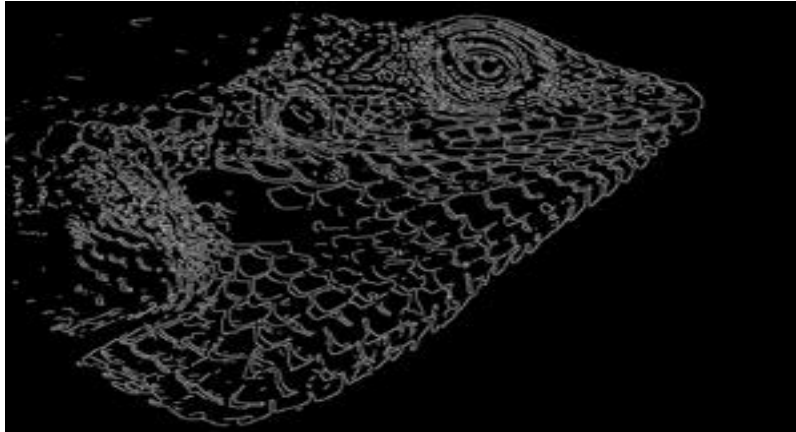
6. Thresholding:

- Pixels due to noise and color variation would persist in the image. So, We perform thresholding to define strong and weak edges:
- Define two thresholds: high and low.
- If a pixel's gradient is above the high threshold, it is marked as a strong edge.
- If a pixel's gradient is below the low threshold, it is not an edge.
- If a pixel's gradient is between the low and high thresholds and connected to a strong edge, it is marked as an edge.

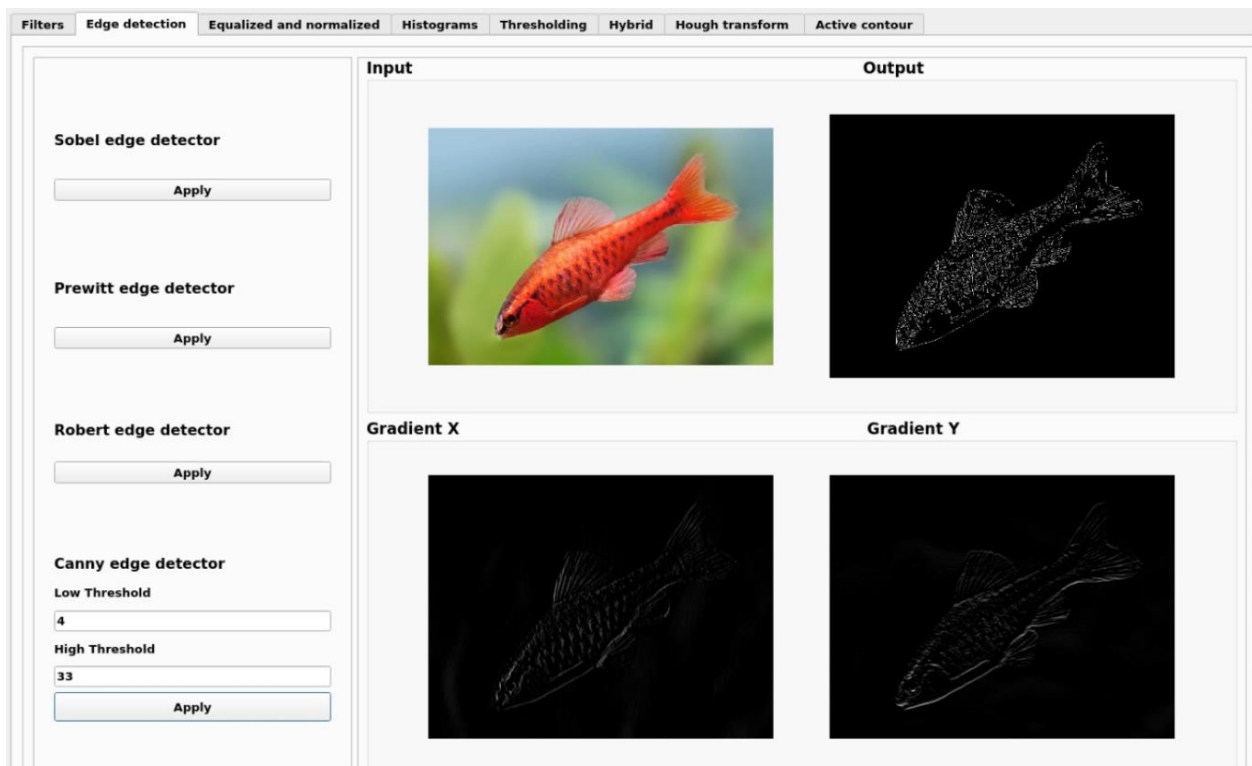


7. Edge Tracking by Hysteresis

- A pixel is made as a strong pixel if either of the 8 pixels around it is strong(pixel value=255) else it is made as 0.
- Finally , We produce final edges as the filter output



Our UI page:



Our result:

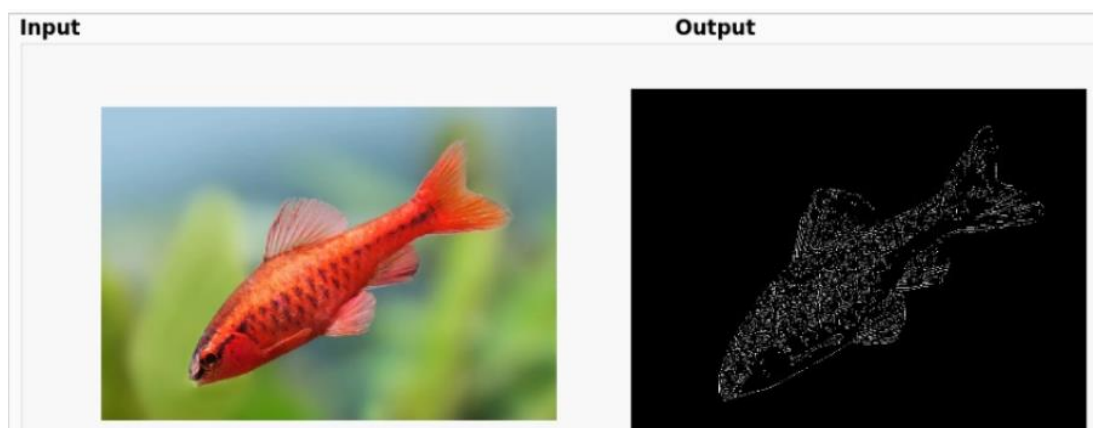
Canny edge detector

Low Threshold

High Threshold

Apply

Example 1:



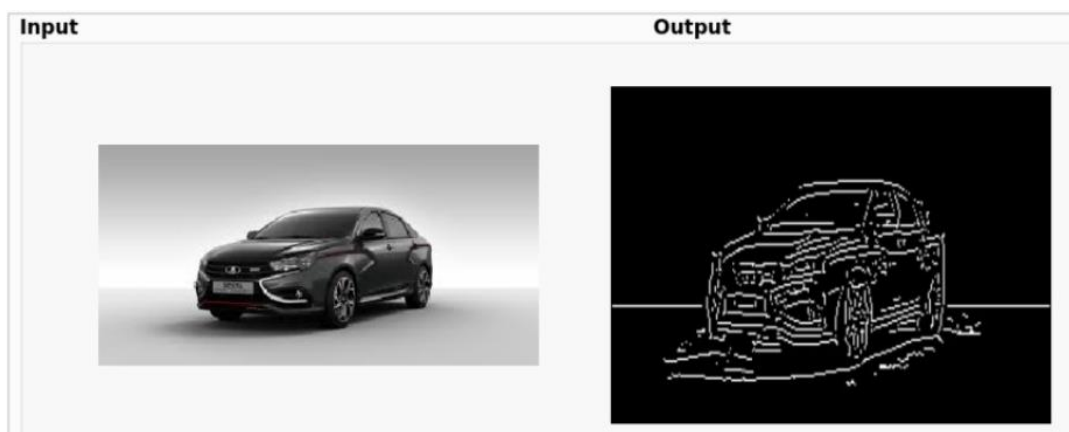
Canny edge detector

Low Threshold

High Threshold

Apply

Example 2:



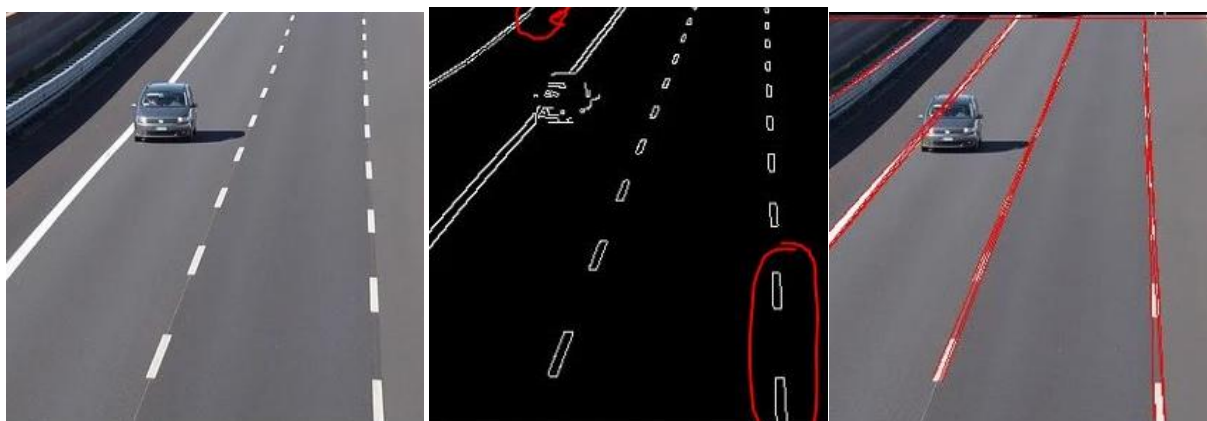
2. Hough transform:

The Hough Transform is a pivotal technique in computer vision and image processing used to detect geometric shapes, notably lines and curves, within digital images. It operates by mapping image points to parameter space, where shapes are represented by curves or parametric equations. By accumulating contributions from edge points in parameter space, the Hough Transform identifies peaks or local maxima corresponding to shapes in the image.

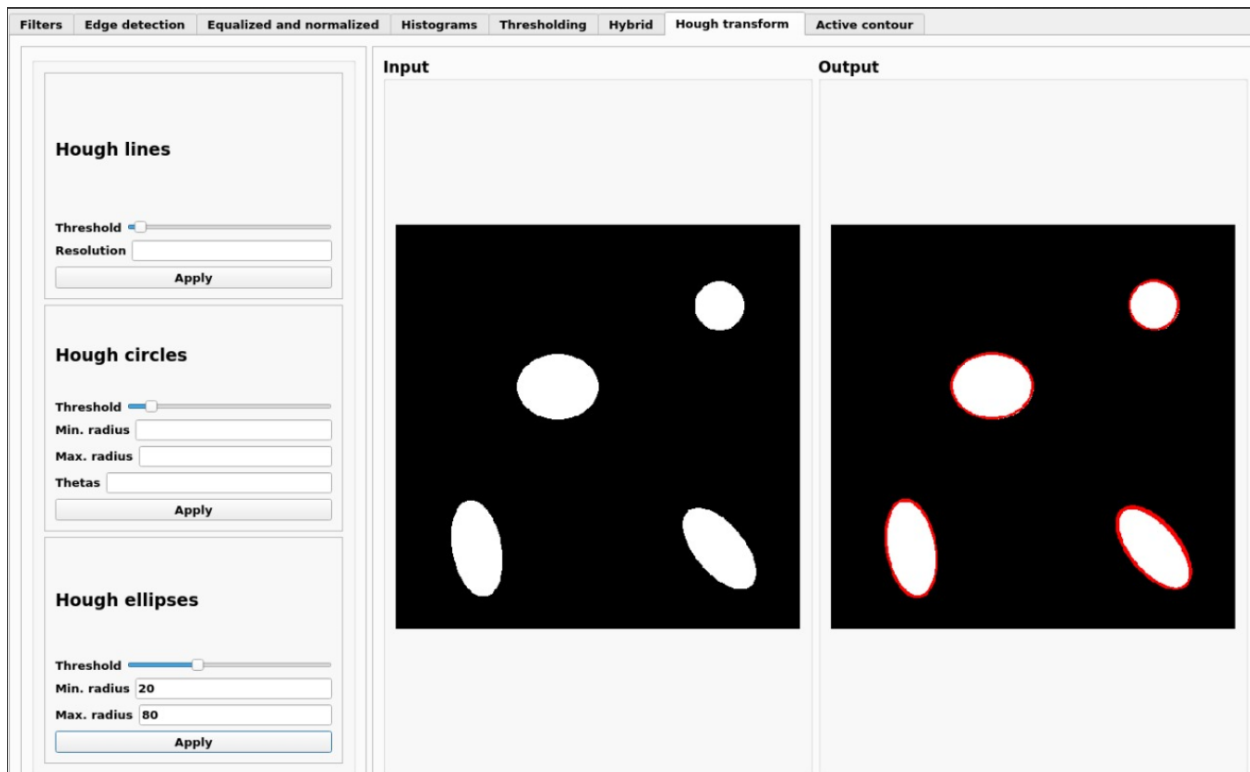
Key features of the Hough Transform include its robustness to noise and partial occlusion, making it suitable for various applications such as object detection, feature extraction, and image analysis. Its flexibility allows for the detection of different shapes by adjusting the parametric representation in parameter space.

In many circumstances, an edge detector can be used as a pre-processing stage to get picture points or pixels on the required curve in the image space. However, there may be missing points or pixels on the required curves due to flaws in either the image data or the edge detector and

spatial variations between the ideal line/circle/ellipse and the noisy edge points acquired by the edge detector. As a result, grouping the extracted edge characteristics into an appropriate collection of lines, circles, or ellipses is frequently difficult.



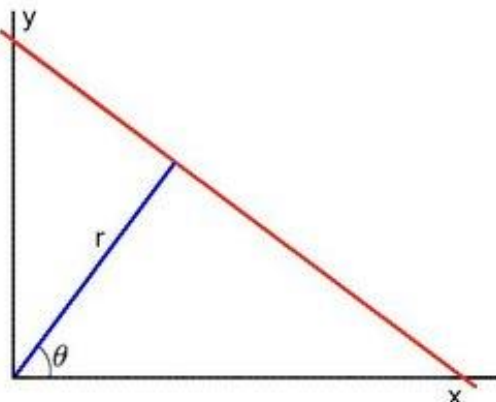
Our UI page:



2.1 Hough line:

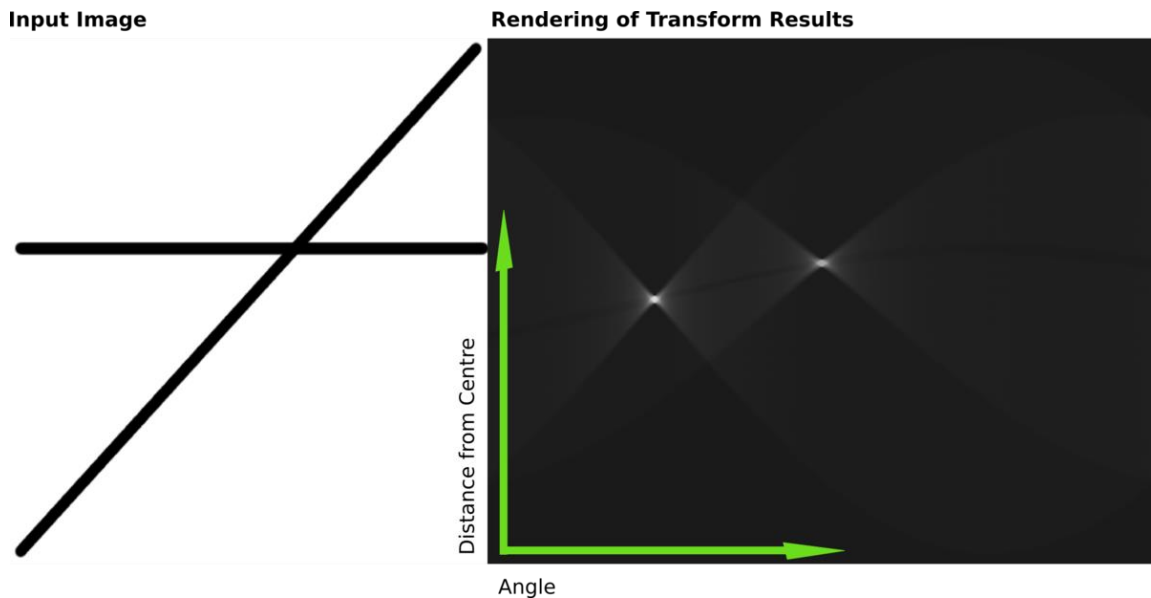
Description:

The Hough line transform function performs the Hough line transform on an input image to detect straight lines within it. It utilizes the Canny edge detection algorithm to find edges in the image and then applies the Hough transform to identify lines. The function allows users to set parameters such as the edge detection threshold and the resolution of the lines detected.



$$y = \left(-\frac{\cos \theta}{\sin \theta} \right) x + \left(\frac{r}{\sin \theta} \right)$$

Example:



The results of this transform were stored in a matrix. Cell value represents the number of curves through any point. Higher cell values are rendered brighter. The two distinctly bright spots are the Hough parameters of the two lines. From these spots' positions, angle and distance from image center of the two lines in the input image can be determined.

Pseudocode:

Function `Hough_Line_Transform(image, threshold, lineResolution):`

1. Convert the image to grayscale.
2. Perform edge detection using the Canny algorithm with the specified threshold.
3. Initialize an accumulator matrix to store votes for potential lines.
4. Loop over each edge pixel in the binary edge image.
 - For each edge pixel:
 - Loop over possible theta values.
 - Calculate rho value corresponding to the edge pixel and current theta.
 - Increment the accumulator at the (rho, theta) bin.

5. Identify peaks in the accumulator matrix above the specified threshold.
6. For each peak:
 - Calculate the corresponding rho and theta values.
 - Convert rho and theta into line equations.
 - Draw the detected line on a copy of the original image.
7. Return the resulting image with detected lines drawn on it.

Explanation:

The function begins by converting the input image to grayscale and performing edge detection using the Canny algorithm to obtain a binary edge image.

It then initializes an accumulator matrix to store votes for potential lines.

The function loops over each edge pixel in the image and for each pixel, iterates over possible theta values, calculating the corresponding rho value and incrementing the accumulator at the corresponding (rho, theta) bin.

After accumulating votes, the function identifies peaks in the accumulator matrix that surpass the specified threshold.

For each peak found, the function calculates the corresponding rho and theta values and converts them into line equations. These lines are then drawn onto a copy of the original image.

Finally, the function returns the resulting image with detected lines drawn on it.

This function provides a robust method for detecting straight lines in an image using the Hough line transform, with customizable parameters such as the edge detection threshold and line resolution.

Our results:

Example 1: with a high threshold and high resolution

Hough lines

Threshold

Resolution



Example 2: with a high threshold and high resolution

Hough lines

Threshold

Resolution



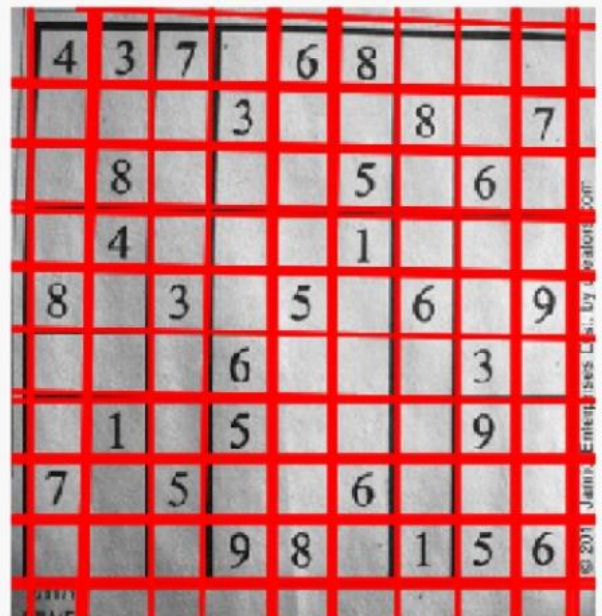
Example 2: with a low threshold and low resolution

Hough lines

Threshold

Resolution

Apply



2.2 Hough Circle:

Description:

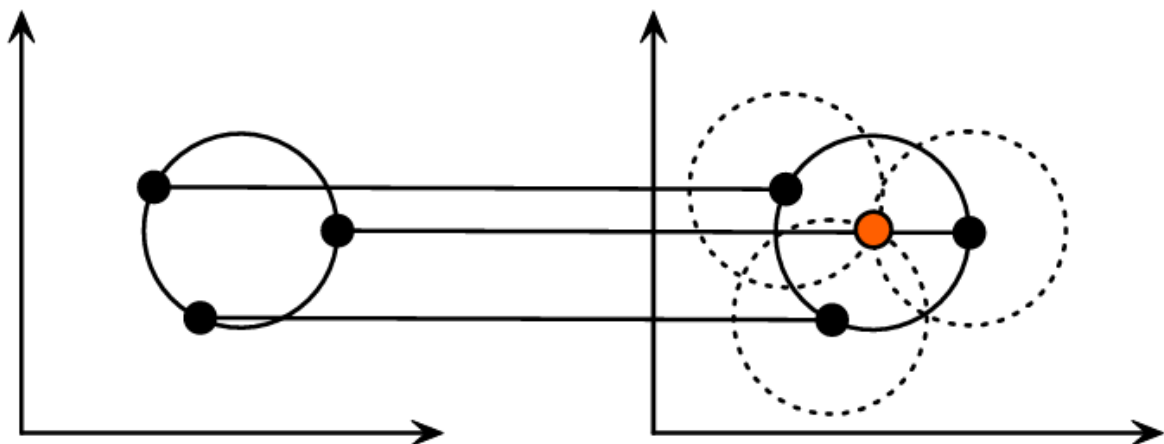
The Hough circle transform function performs the Hough circle transform on an input grayscale image to detect circles within it. It utilizes edge detection to identify potential circle centers and radius, then accumulates votes in a 2D accumulator array based on the detected edges. Circles are detected based on the accumulated votes exceeding a specified threshold, and they are drawn onto a copy of the original image.

In a two-dimensional space, a circle can be described by:

$$(x - a)^2 + (y - b)^2 = r^2 \quad (1)$$

$$x = a + R \cos(\theta)$$

$$y = b + R \sin(\theta)$$



Each point in geometric space (left) generates a circle in parameter space (right). The circles in parameter space intersect at the (a, b) that is the center in geometric space.

Pseudocode:

function Hough_circle_transform(image, threshold, min_radius, max_radius, thetas):

1. Convert the input image to grayscale.
2. Perform edge detection using Canny edge detector with predefined thresholds.
3. Initialize a 2D accumulator array to store votes for circle parameters.
4. Loop over the edge-detected image to identify potential circle centers:
 - Sample edge points to reduce computational complexity.
 - For each edge point:
 - Loop over possible radii within the specified range:
 - Loop over discrete angles (thetas) around the circle:
 - Compute the center coordinates of the circle based on the edge point, radius, and angle.
 - Increment the accumulator at the computed center coordinates.
5. Draw circles based on accumulated votes exceeding the threshold:
 - Loop over each pixel in the accumulator array:
 - If the vote count exceeds the threshold:
 - Draw the circle center.
 - For each radius within the specified range:
 - Draw the circle with the center and radius.
6. Return the image with detected circles drawn on it.

Our results:

Example 1:

Hough circles

Threshold

Min. radius

Max. radius

Thetas



Example 2: we detect the big circle only with low threshold

Hough circles

Threshold

Min. radius

Max. radius

Thetas



Example 3: we detected the small circles using a small minimum radius

Hough circles

Threshold

Min. radius

Max. radius

Thetas



2.3 Hough ellipse:

Description:

The provided functions enable automated detection of ellipses in digital images, serving as a valuable tool in various computer vision and image processing tasks.

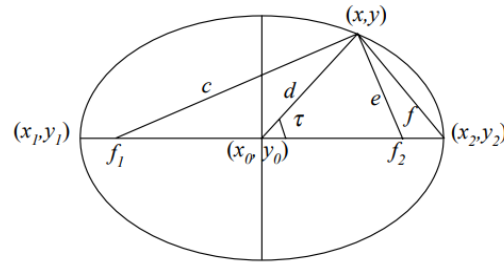
We get the equations of the ellipse in parameter space from this research paper “[A NEW EFFICIENT ELLIPSE DETECTION METHOD](#)“, because the parameters of the ellipses is a lot (5-parameters) we use a parameter space reduction method from [this reference](#)

$$x_0 = (x_1 + x_2)/2$$

$$y_0 = (y_1 + y_2)/2$$

$$a = [(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2}/2$$

$$\alpha = \text{atan} [(y_2 - y_1)/(x_2 - x_1)],$$



And for simplicity we create these three main components: boundary detection, ellipse fitting, and ellipse visualization.

Boundary Detection: This function utilizes the Canny edge detection algorithm to identify edge pixels in the image. These edge pixels are then used to extract contours, forming the basis for subsequent ellipse fitting.

Hough Ellipse Detection: Using the extracted contours, this function fits ellipses to regions of interest within the image. A user-defined threshold ensures robustness against noise and clutter, allowing for accurate ellipse detection.

Draw Ellipses: After detecting ellipses, this function visualizes them on the original image. It enables qualitative analysis and interpretation of the detected geometric features.

Pseudocode:

Hough Ellipse Function:

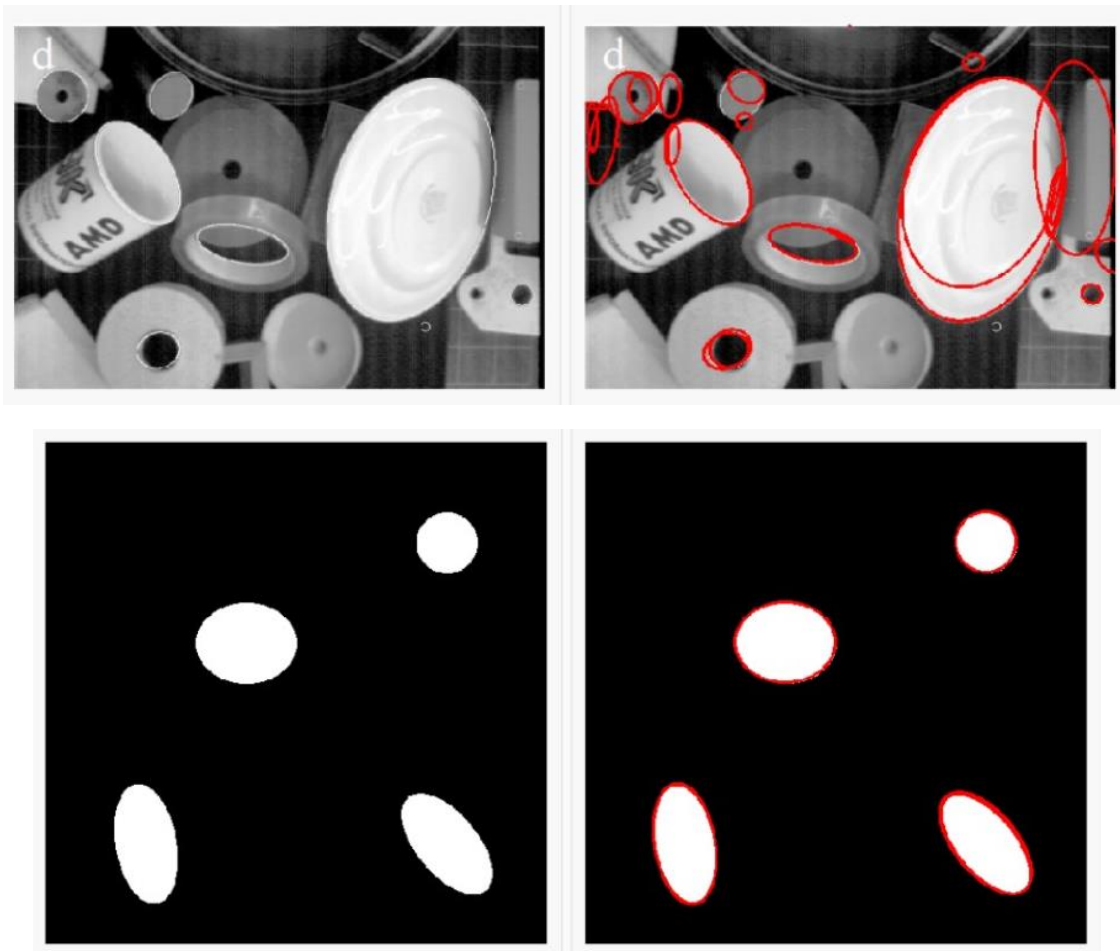
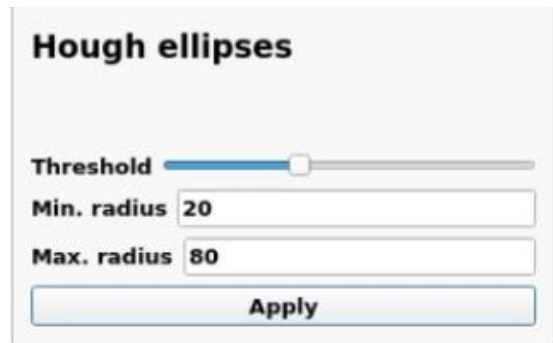
1. Detect boundaries using the `boundary_detection` function, storing the resulting contours.
2. For each contour in the list of detected contours:
 - If the contour contains at least the specified threshold number of points:
 - Fit an ellipse to the contour using OpenCV's `fitEllipse` function.
 - Extract parameters of the fitted ellipse: center (x, y), major and minor axes lengths, angle.
 - Add the ellipse parameters to the list of ellipses.
3. Draw the detected ellipses on the input image using the `drawEllipses` function.

Boundary detection function:

1. Clear the existing contours.
2. Apply Canny edge detection on the input image with thresholds of 100 and 600, storing the result in `edges`.
3. Traverse through each pixel of the edge-detected image:
 - If the pixel value is not zero (i.e., it's an edge pixel):
 - Create a new contour.
 - Add the current pixel to the contour.
 - Mark the current pixel as visited.
 - While there are neighboring edge pixels:
 - Check neighboring pixels.
 - If a neighboring pixel is an edge pixel:
 - Update the current pixel.
 - Add the updated pixel to the contour.
 - Mark the updated pixel as visited.
 - Continue until no more neighboring edge pixels are found.
 - Add the contour to the list of contours.
4. Output the list of detected contours.

Our results:

Examples: the same settings are applied



3. Active contour:

Active contours, also known as snakes, are powerful tools in image processing and computer vision used for contour detection, object segmentation, and boundary extraction. They were introduced by Michael Kass, Andrew Witkin, and Demetri Terzopoulos in the late 1980s. Active contours are widely employed in various applications such as medical image analysis, object tracking, and shape modeling.

The concept behind active contours is inspired by the behavior of snakes in nature, which can move and deform their bodies to adapt to their surroundings. Similarly, active contours are flexible, deformable curves or surfaces that can iteratively adjust their shape to match the features of interest in an image.

Active contours are driven by energy minimization principles, seeking to minimize an energy functional that is defined based on image features such as gradient information, edge intensity, and smoothness constraints. By iteratively minimizing this energy functional, active contours evolve towards the boundaries of objects or regions of interest in the image.

first we initialize contour.

The input parameters for this function are:

- center: The center point of the circle.
- radius: The radius of the circle.

The function initializes an empty vector curve to store the points of the circular contour.

It then iterates over angles from 0 to 360 degrees (inclusive) with a step size of 1 degree. For each angle:

1. It converts the angle from degrees to radians ($\text{angle} = i * \text{CV_PI} / 180$).
2. It calculates the x and y coordinates of a point on the circle using trigonometric functions:

5.
$$\begin{aligned} x &= \text{center.x} + \text{radius} \times \cos(\text{angle}) \\ y &= \text{center.y} + \text{radius} \times \sin(\text{angle}) \end{aligned}$$

3. It creates a Point object with the calculated coordinates and adds it to the curve vector.

After iterating over all angles, the function returns the vector curve, containing the points representing the circular contour.

In summary, the `initial_contour` function generates an initial circular contour by calculating points along the circumference of a circle with the given center and radius and returns these points as a vector of `Point` objects.

We calculate three types of energy:

- External Energy
- Internal Energy
- Balloon Energy

3.1 External Energy

refers to the energy associated with features or characteristics external to the object or region of interest in an image. In the context of edge detection, such as with the Canny edge detector, external energy can be calculated based on the intensity values of pixels along the detected edges.

In function `calcExternalEnergy`, the input parameters are:

- `img`: The input image, likely already processed by a Canny edge detector to highlight edges.
- `pt`: The coordinates of a specific pixel in the image.
- `beta`: A factor used to control the weight of the external energy.

The function retrieves the intensity value of the pixel at the specified coordinates (`pt.x`, `pt.y`) in the input image using `img.at<uchar>(pt.y, pt.x)`. This intensity value represents the strength of the edge at that pixel location.

The function then multiplies this intensity value by `-beta`. The negative sign indicates that the energy is being calculated in a way that pulls the contour towards the edge rather than away from it. The factor `beta` allows adjusting the strength of this effect.

3.2 Internal Energy:

This internal energy is calculated based on the curvature at the point, with the intention of promoting smoothness in the contour.

The input parameters for this function are:

- ``pt`` : The current point in the contour for which internal energy is being computed.
- ``prevPt`` : The previous point in the contour.
- ``nextPt`` : The next point in the contour.
- ``alpha`` : A parameter used to control the influence of the curvature on the internal energy.

The function first computes the differences in x and y coordinates between ``pt`` and its neighboring points (``prevPt`` and ``nextPt``), which are denoted as ``dx1``, ``dy1``, ``dx2``, and ``dy2`` respectively.

Then, it calculates the curvature at ``pt`` using the formula for curvature estimation in 2D curves:

$$\text{curvature} = \frac{dx_1 \cdot dy_2 - dx_2 \cdot dy_1}{(dx_1^2 + dy_1^2)^{3/2}}$$

This formula computes the signed curvature at ``pt``. Positive curvature indicates the contour is bending to the left, while negative curvature indicates bending to the right. The magnitude of the curvature is inversely proportional to the square of the distance between ``pt`` and its neighbours, which encourages smoothness in the contour.

Finally, the function multiplies the computed curvature by the parameter ``alpha`` to control its influence on the internal energy.

3.3 Balloon Energy:

Balloon energy is a term commonly used in image segmentation algorithms, particularly active contour models, to model forces that push or pull the contour towards or away from object boundaries.

The input parameters for this function are:

- ``pt`` : The current point in the contour for which balloon energy is being computed.
- ``prevPt`` : The previous point in the contour.
- ``gamma`` : A parameter used to control the strength of the balloon force.

The function first computes the differences in x and y coordinates between ``pt`` and its previous point ``prevPt``, which are denoted as ``dx`` and ``dy`` respectively.

Then, it calculates the balloon energy using the formula:

$$\text{balloon energy} = \gamma \cdot (dx^2 + dy^2)$$

This formula computes the energy based on the squared distance between `pt` and its previous point `prevPt`, effectively measuring the local expansion or contraction of the contour. Larger values of `gamma` amplify this effect, making the contour more sensitive to local curvature changes.

Finally, the function returns the computed balloon energy, scaled by the parameter `gamma`.

We move the curve by this energy according to energy minimization principles by iteratively moving each point to a nearby location that minimizes the overall energy.

The input parameters for this function are:

- `img`: The input image.
- `curve`: A vector of points representing the curve to be moved.
- `alpha`: A parameter controlling the influence of internal energy.
- `beta`: A parameter controlling the influence of external energy.
- `gamma`: A parameter controlling the influence of balloon energy.

The function starts with creating a new curve to store the updated points. It then iterates through each point in the original curve.

It initializes `minEnergy` to a large value (`DBL_MAX`) and `newPt` to the current point (`pt`).

The function then iterates over nearby locations around the current point (including the point itself and its 8 adjacent points) and calculates the energy associated with moving the point to each of these locations using the `calcEnergy` function. It then updates `minEnergy` and `newPt` if a lower energy configuration is found.

Finally, the function updates the `newCurve` with the location of the point (`newPt`) that minimizes the energy.

After processing all points in the original curve, the function replaces the original curve with the updated `newCurve`, effectively moving each point to a new position that minimizes the combined energy of the curve.

3.4 Contour perimeter:

This function calculates the perimeter of a contour defined by a sequence of points. It iterates over each pair of adjacent points in the contour and computes the distance between them using the `points_distance` function. The total distance is accumulated in the variable `distance_sum`, which represents the perimeter of the contour.

The input parameters for this function are:

- `points`: A vector containing the points of the contour.
- `n_points`: The number of points in the contour.

For each pair of adjacent points (`i` and `i+1`), it calculates the Euclidean distance between them using the `points_distance` function. If the current point is the last point in the contour, it considers the next point to be the first point to close the contour. Finally, it returns the total sum of distances, which represents the perimeter of the contour.

3.5 Contour area:

This function calculates the area of a polygonal contour defined by a sequence of points using the shoelace formula (also known as Gauss's area formula or the surveyor's formula). The formula computes the area of a polygon given the coordinates of its vertices.

The input parameters for this function are:

- `points`: A vector containing the points of the contour.
- `n_points`: The number of points in the contour.

The function initializes the variable `area` to 0.0 to accumulate the area of the contour. It also initializes the variable `j` to the index of the last point in the contour (`n_points - 1`).

The function then iterates over each point in the contour. For each iteration, it calculates the contribution to the area of the contour formed by the current point and the previous point using the formula:

$$\text{area} += (\text{points}[j].x + \text{points}[i].x) \times (\text{points}[j].y - \text{points}[i].y)$$

The variable `j` is updated to the current index `i` to prepare for the next iteration.

Finally, the function returns the absolute value of half the accumulated area, as the shoelace formula yields twice the actual area of the polygon. This ensures that the returned value represents the absolute area of the polygon.

3.6 Chain Code:

This function generates a chain code representing the contour of an object in a binary image.

The input parameters are:

- image: A 2D vector representing the binary image.
- startPoint: The starting point for tracing the contour.

Inside the function, it initializes an empty vector contour to store the chain code representing the contour.

It then iteratively traces the contour by moving from one pixel to the next, following the object boundary.

The function starts from the startPoint and follows the object boundary until it reaches the starting point again.

At each step, it determines the next direction to move using the findNextDirection function, and checks the neighboring pixels using the getNeighbor function.

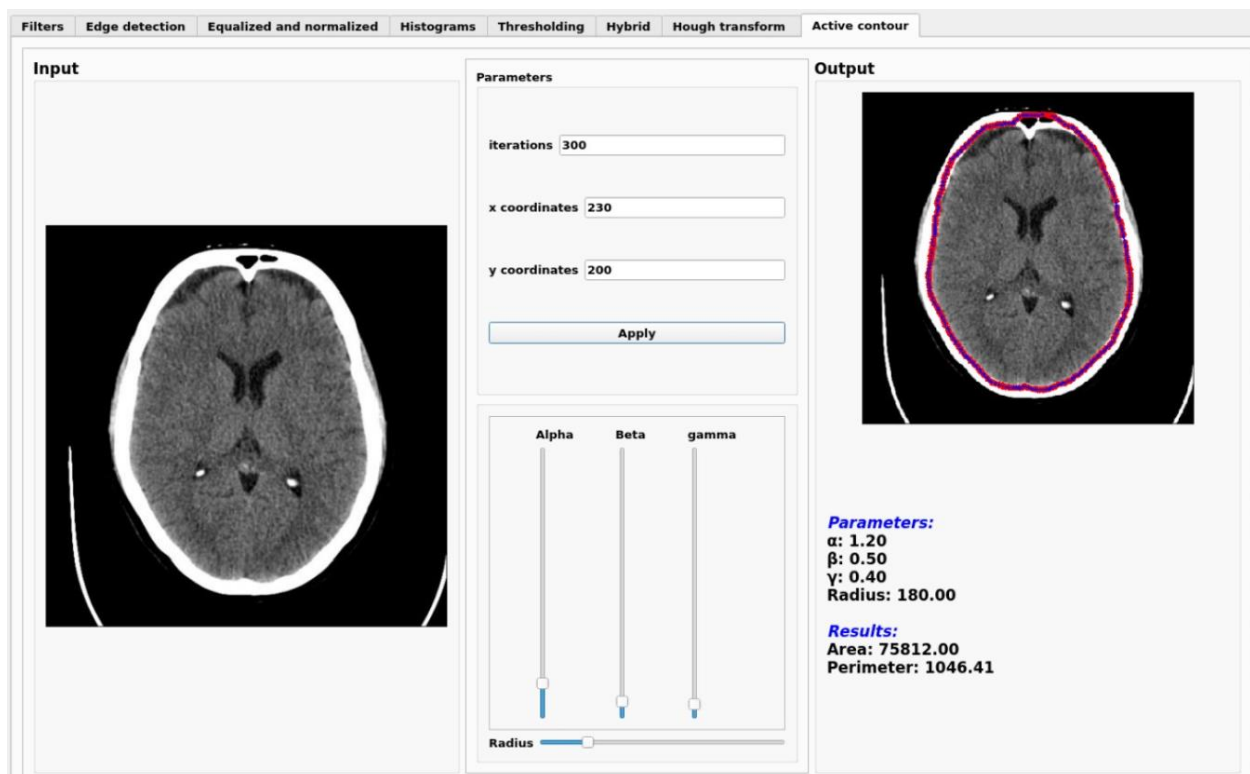
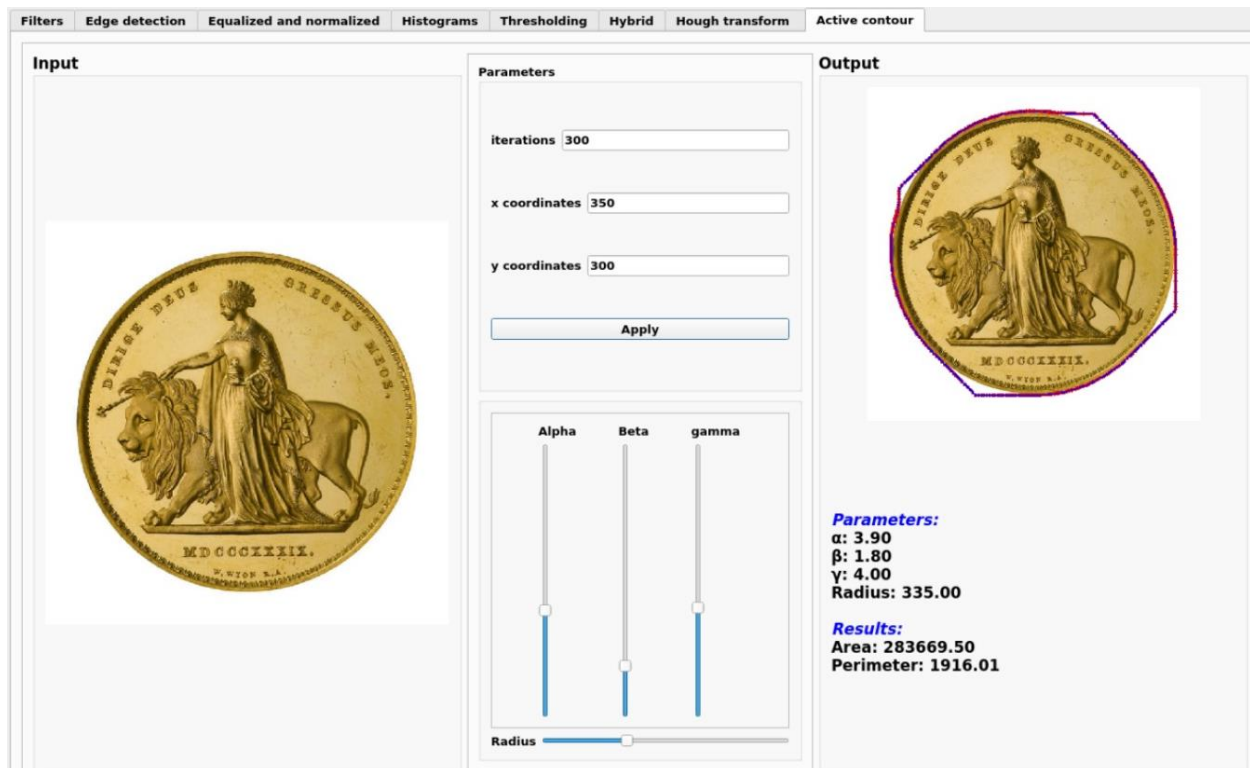
The function terminates when it either completes a full loop around the object or when the maximum number of iterations (maxIterations) is reached to prevent infinite loops.

Finally, it returns the vector contour containing the chain code representing the contour of the object in the image.

```
23:53:53: Starting /home/megzz/build-task2-Desktop_Qt_6_5_3_GCC_64bit-Debug/task2...
Error: Unable to convert image to QImage.
0.
20.30.40.50.60.70.80.911.11.21.31.41.51.61.71.81.90.28.30.40.50.60.78.89.911.11.21.31.4
.61.71.81.922.12.22.32.42.52.62.72.82.70.20.30.40.50.60.70.8591317202428323640444852555
77175796387909498102106110114118122125129133137141145149153156160164168172ChainCode = 8
3 0 0 5 1 6 2 7 3 0 0 5 1 6 2 7 3 0 0 5 1 6 2 7 3 0 0 5 1 6 7 4 0 5 1 6 6 3 7 4 0 5 1 6
7 4 0 5 1 6 6 3 7 4 0 5 1 6 6 3 7 4 0 5 1 6 2 7 3 0 4 1 5 2 2 7 3 0 4 1
2 7 3 0 4 1 5 2 2 7 3 0 4 1 5 2 2 7 3 8 4 1 5 2 2 7 3 0 4 1 5 2 3 0 4 1 5 2 6 3 7 4 4 1
6 3 7 4 4 1 5 2 6 3 7 4 4 1 5 2 6 3 7 4 4 1 5 2 6 3 7 4 6
ChainCode = 0 4
2.1ChainCode = 0 4
```

Our results and UI page:

The user can set all the parameters by himself.



Filters

Edge detection

Equalized and normalized

Histograms


Thresholding

Hybrid

Hough transform

Active contour

Input



Parameters

iterations

300

x coordinates

140

y coordinates

95

Apply


Alpha

Beta

gamma

Radius

Output



Parameters:

α : 9.30
 β : 1.30
 γ : 4.30
Radius: 91.00

Results:

Area: 12418.00
Perimeter: 789.20

Filters

Edge detection

Equalized and normalized

Histograms

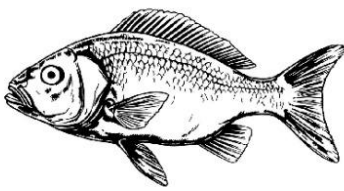
Thresholding

Hybrid

Hough transform

Active contour

Input



Parameters

iterations

300

x coordinates

400

y coordinates

295

Apply

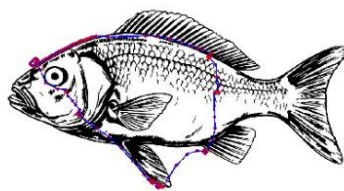
Alpha

Beta

gamma

Radius

Output



Parameters:

α : 7.00
 β : 0.60
 γ : 0.50
Radius: 220.00

Results:

Area: 105101.50
Perimeter: 1869.60