



SHAPE DETECTION AND CONTOURING

Assignment 2

TEAM 1

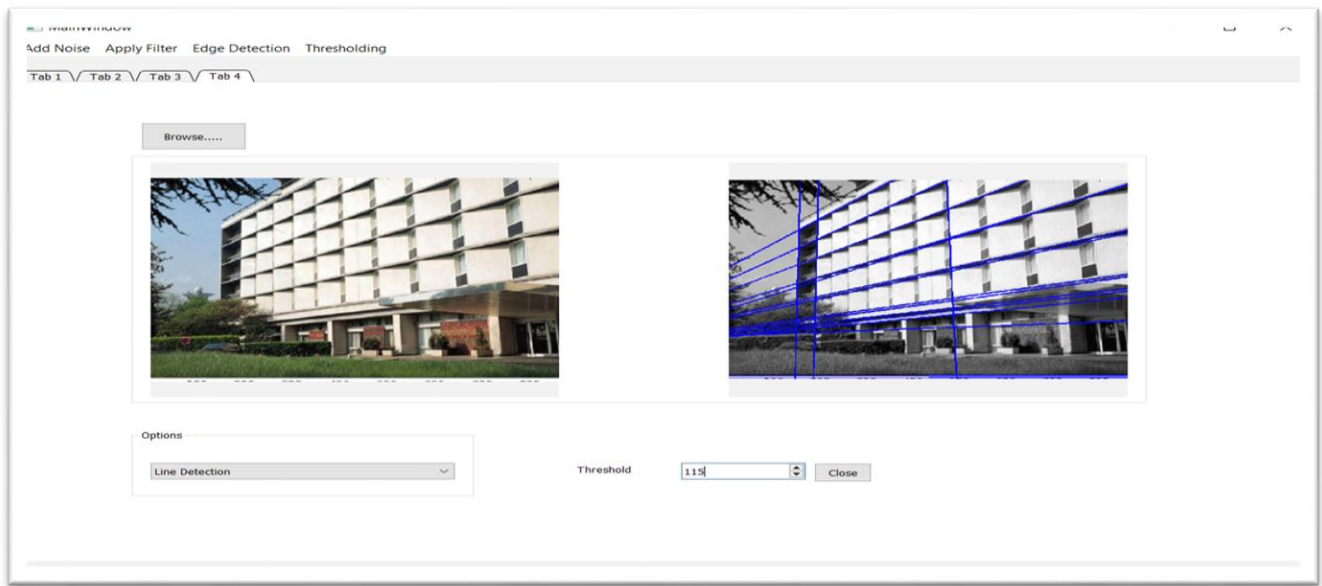
- HABIBA FATAHALLA
 - RAWAN MOHAMMED FEKRY
 - SARA AYMAN EL-WATANY
 - MARIAM WAEL
-
- Eng. Peter Emad
 - Eng. Laila Abbaas

Introduction:

To implement the algorithms in the assignment from scratch we used C++ and Qt for gui, to be able to obtain the same results as the algorithms in Open cv built in library.

To read and show the images we used open cv methods, and we also used it to test our output to see if it's accurate or not compared to the built in algorithms.

Line Detection:



To detect lines in image we use [Hough line Transform](#).

Before applying our technique we first need to change the input image into an edge image by using [Canny Edge Detection](#).

The line equation in the edge image is represented by the equation $y = ax + b$ where **a** represents the slope and **b** represents the intercept of the line on the edge image.

Hough Space is a 2D plane with vertical axis represents the intercept of line on edge image **b** and horizontal axis represents the slope **a**.

One line on the edge image can produce a point on the Hough Space since it's characterized by its slope **a** and intercept **b**.

Respectively an edge point (x_i, y_i) in the image has infinite number of lines passing through it, So an edge point produces a line in the Hough Space which it's equation is $b = ax_i + y_i$

So in [Hough Transform Algorithm](#) we use Hough Space to determine whether a line exists in the edge image or not.

To make the algorithm able to detect vertical lines ($a = \text{infinity /undefined}$) we need to represent the line differently by using the form of the [Normal line](#) which is $\rho = x \cos(\theta) + y \sin(\theta)$ where ρ is the length of the normal line and θ is the angle between the x-axis and the normal line

so the Hough Space is now represented by ρ in the vertical axis and θ on the horizontal axis

Our Algorithm:

- Determine the range of values for both ρ and θ , the range of θ is $[0, 180]$ and the range of ρ is $[-d, d]$ where d is the length of the edge image diagonal
- First we apply canny edge detection technique to the input image
- We initialize the accumulator matrix with zeros as a 2D array based on the image dimensions
- Loop over each pixel in the edge image, and for each non zero pixel value we iterate over the range of θ values to compute d using $d = x \cos(\theta) + y \sin(\theta)$ where x and y are the pixel coordinates in the edge image, and the update the values in the accumulator matrix which will be used to detect the lines
- Let the user decide an **threshold value** by which we will decide the min. number of votes to consider a line is a truly a line, we iterate over the accumulator matrix and try to find the local max. to convert its polar coordinates(ρ, θ) to cartesian coordinates(x, y) and draw these lines

Note:

By changing the threshold value the detect lines may change .

Circle Detection:



To detect lines in image we use [Circle Hough Transform](#).

To be able to represent a circle we mainly need three parameters (a,b,r) which can be formed by the equation $(x - a)^2 + (y - b)^2 = r^2$ where **a** and **b** are the center of the circle so in this case we will need 3D accumulator array.

So our Hough Space axis are **a, b and R** which represent a 3d plane and for each point in the edge image we draw a corresponding circles in the Hough Space.

Our Algorithm:

- As in line detection we first deal with the input image as edge image by applying Canny detection algorithm.
- The range of the θ [0,360] and the user enters the min. and max radius for the circles he wants
- We first compute the Hough vote for each pixel in the edge image which represents the number of circles that pass through the pixel in the edge image and we loop over the edged image, and for each non zero pixel value we iterate over the θ values and compute $a = x - r * \cos(\theta)$ & $b = y - r * \sin(\theta)$ and compute the max. vote value and update it
- We search for the local max. in the Hough vote matrix which is above the threshold value.
- We then make sure that there's no duplicated or overlapping circles in our vector

Ellipse Detection:

We had 2 approaches for ellipse detection, one which relies on contour detection and then undergoing certain steps to confirm the ellipsoid shape, and the other using the Hough transform.

- First Approach (Refer to lines 299-644 in hough.cpp)

Mainly relies on canny edge detection & Hough transform, where accumulator matrix(vector) is used to store the votes for each assumption you have for your minor axis using some certain equations by getting the position corresponding to your maximum vote for each case forming your assumed ellipse given that the major axis, center, and orientation were previously assumed.

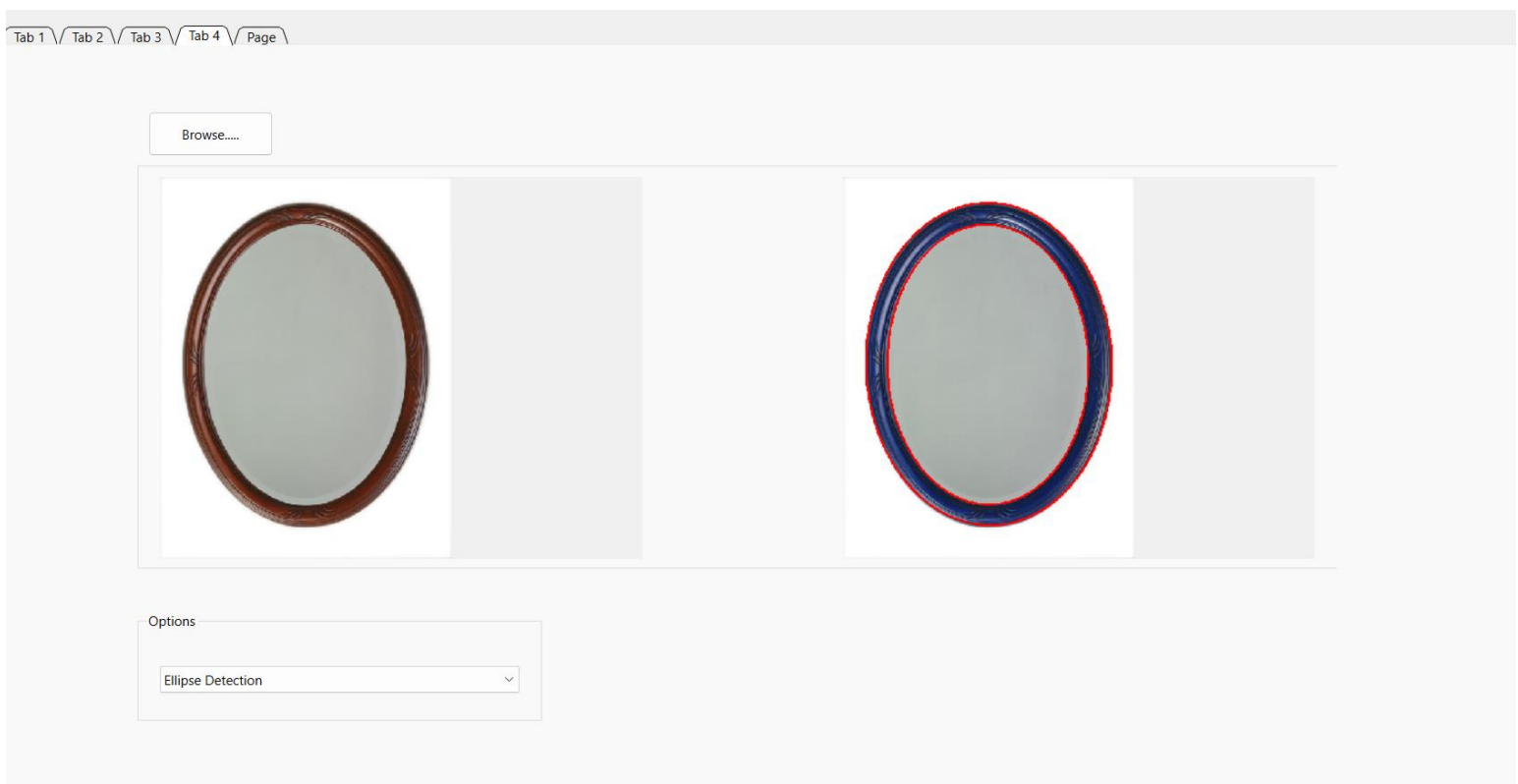
- 1- Any Ellipse can be defined by 5 parameters center(x_0, y_0), major axis(a), minor axis(b) and orientation.
- 2- Given this concept you can get the edges present in your image and store these non-zero intensity points in a vector.
- 3- Consider each 2 points of the edge pixels to be the major axis by checking if they are above some minimum value.
- 4- Get the center(x_0, y_0), major axis(a), and orientation for your assumed ellipse.
- 5- Loop through each edge pixel again and assume it is a random point on the ellipse.
- 6- Get the distance between the previous calculated center(x_0, y_0), and your points, and store votes in your accumulator array for each length.
- 7- Get the highest vote, which represents your possible minor axis length.
- 8- Draw your ellipse on your image after you've looped through all the edge pixels stored in your vector.

Credits go to this paper:

https://sites.ecse.rpi.edu/~qji/Papers/ellipse_det_icpr02.pdf

- Second Approach (Refer to lines 656-749 in hough.cpp)
 - 1- Get all the contours for the objects of a given image, you will end up getting a black image with contours representing each object.
 - 2- Draw bounding rectangles for each contour using the maximum and minimum points for horizontal and vertical axes.
 - 3- For each bounding box drawn you can get the width and height of it along with its orientation.
 - 4- Given the width and height you assume the major as well as the minor axis of your ellipse.
 - 5- Loop through each point (x) from left to right of your assumed ellipse for the upper and lower part by substituting in the ellipse equation to get (y).
 - 6- Store the y points of the previous step, compare each vector with the ones stored in the contour vectors.
 - 7- Get the amount of dissimilarity between the assumed ellipse and your contour, if it is below certain point then our assumption was true (set to **0.01** in our implementation), and you can proceed by drawing it on your image.



▪ Image 1



■ Image 2

Tab 1 / Tab 2 / Tab 3 / Tab 4 / Page

Browse.....

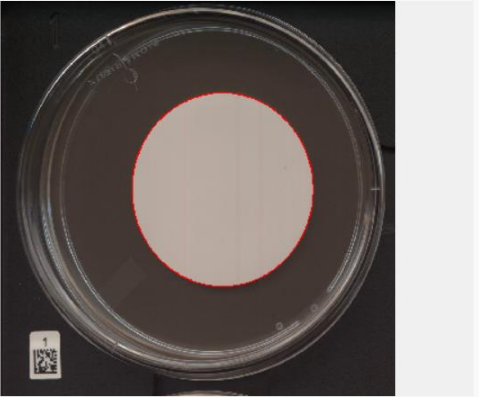
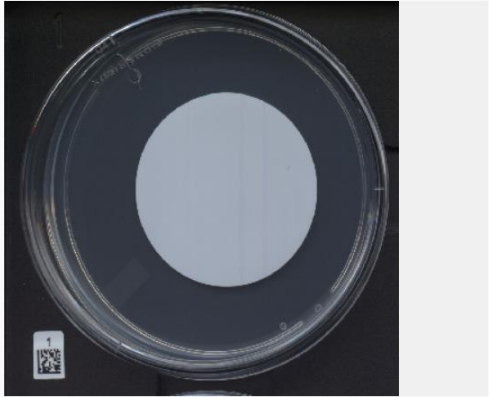


Options
Ellipse Detection

■ Image 3

Tab 1 / Tab 2 / Tab 3 / Tab 4 / Page

Browse.....





Options
Ellipse Detection

▪ Image 4

Tab 1 Tab 2 Tab 3 Tab 4 Page

Browse.....



Options

Ellipse Detection

Note

The intensity of output image slightly varied from the intensity of input image due to the conversion from QImage object to Mat object (QT related issue).

Active Contour (Snake):

Method of bounding objects by making an initial boundary and then continuously iterating to make the contour fit properly. (Refer to lines 9-30 in activecontour.cpp)

1. Calculate the internal energy of a contour point, which is a measure of how much the point deviates from the overall shape of the contour by calculating the differences in x and y coordinates between the current point and its neighboring points along with the curvature of the point using certain equation (represents the rate at which the direction of the contour changes at this point). (Refer to lines 33-46 in activecontour.cpp)

Parameters:

- pt: The current point on the contour.
- prevPt: The previous point on the contour.
- nextPt: The next point on the contour
- alpha: Changes the curvature of our contour.

Returns:

- The internal energy, where the result is returned as a double.

2. Calculate the external energy of a contour point, which is a measure of how much the point likes" or "dislikes" being in its current position based on the image content. (Refer to lines 49- 54 in activecontour.cpp)

Parameters:

- pt: The current point on the contour.
- img: The greyscale image.
- beta: Changes the curvature of our contour.

Returns:

- The external energy.

3. Calculate the energy associated with a balloon term using the distance between the current and previous points on the contour, where the balloon term is used to control the contour's shape by attracting or repelling it from a given point. (Refer to lines 57-66 in activecontour.cpp)

Parameters:

- pt: The current point on the contour.
- prevPt: The previous point on the contour.
- gamma: The strength of the balloon term.

Returns:

- The energy associated with the balloon term.

4. Update the initial contour according to the energy functions calculated previously by iterating through certain number of loops that the user passes it through the UI. (Refer to lines 68-124 in activecontour.cpp)

Parameters:

- inputImg: The input image.
- outputImg: The output image to be displayed.
- center: The center of your contour.
- radius: radius of your contour (taken by the user).
- numOfIterations: The total number of iterations (taken by the user).
- alpha: Taken by the user.
- gamma: Taken by the user.

Returns:

- Contour points.

5. Calculate a vector that contains the direction of movement to each point by taking the final contour as a parameter. (Refer to lines 126-165 in activecontour.cpp)

Parameters:

- contour: The final contour.

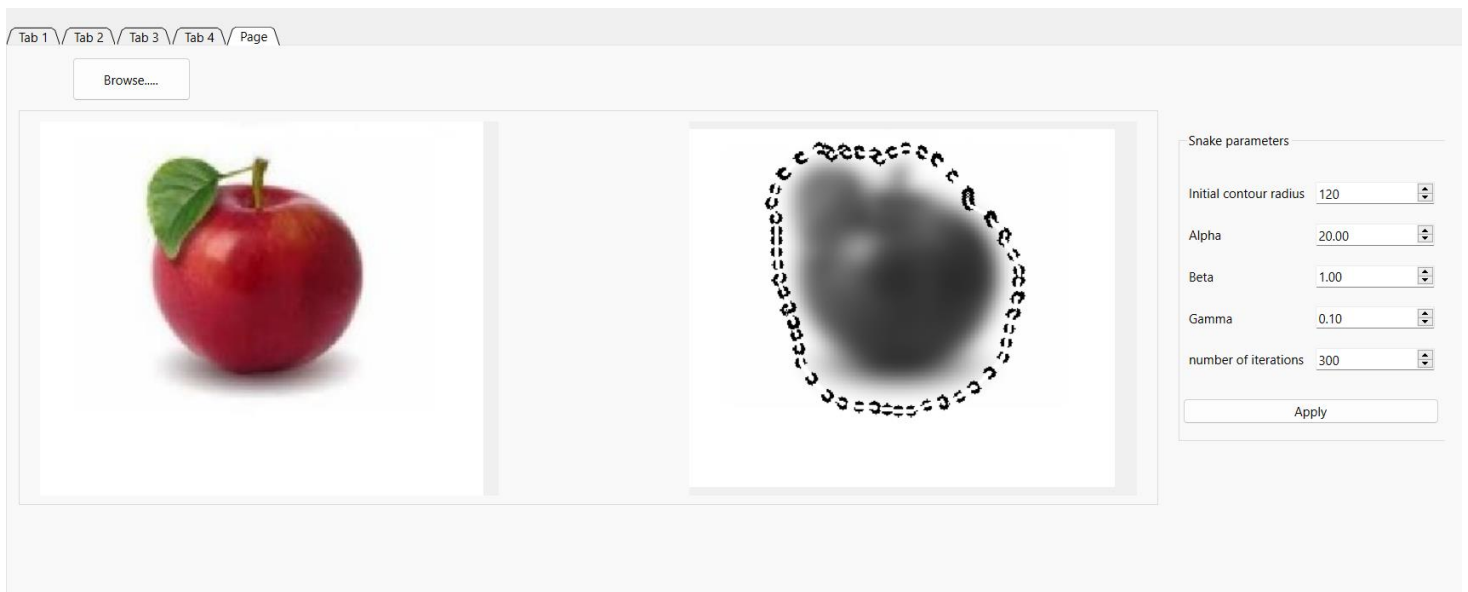
Returns:

- Chain code.

First Image:

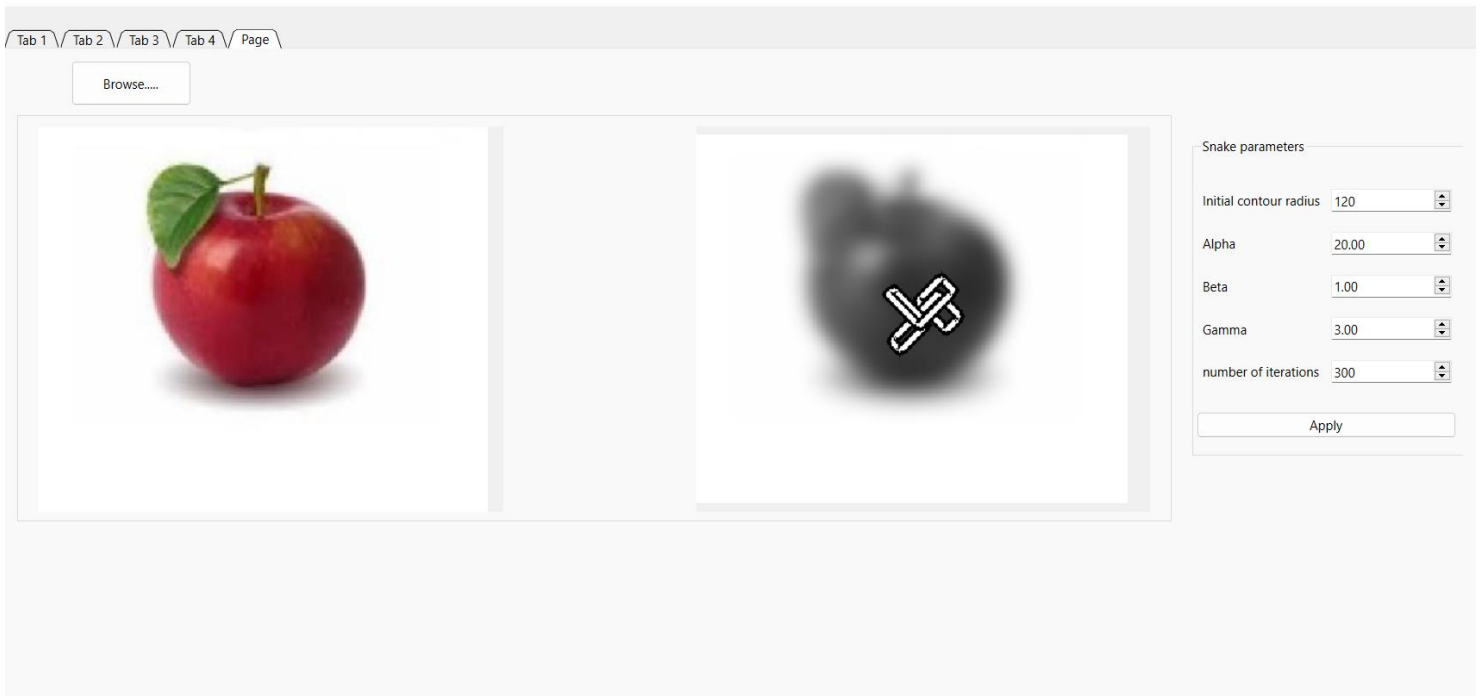
Matching Variables:

- Initial Contour Radius = 120
- Alpha = 20
- Beta = 1
- Gamma = 0.1
- No. of iterations = 300



Mismatching Variables:

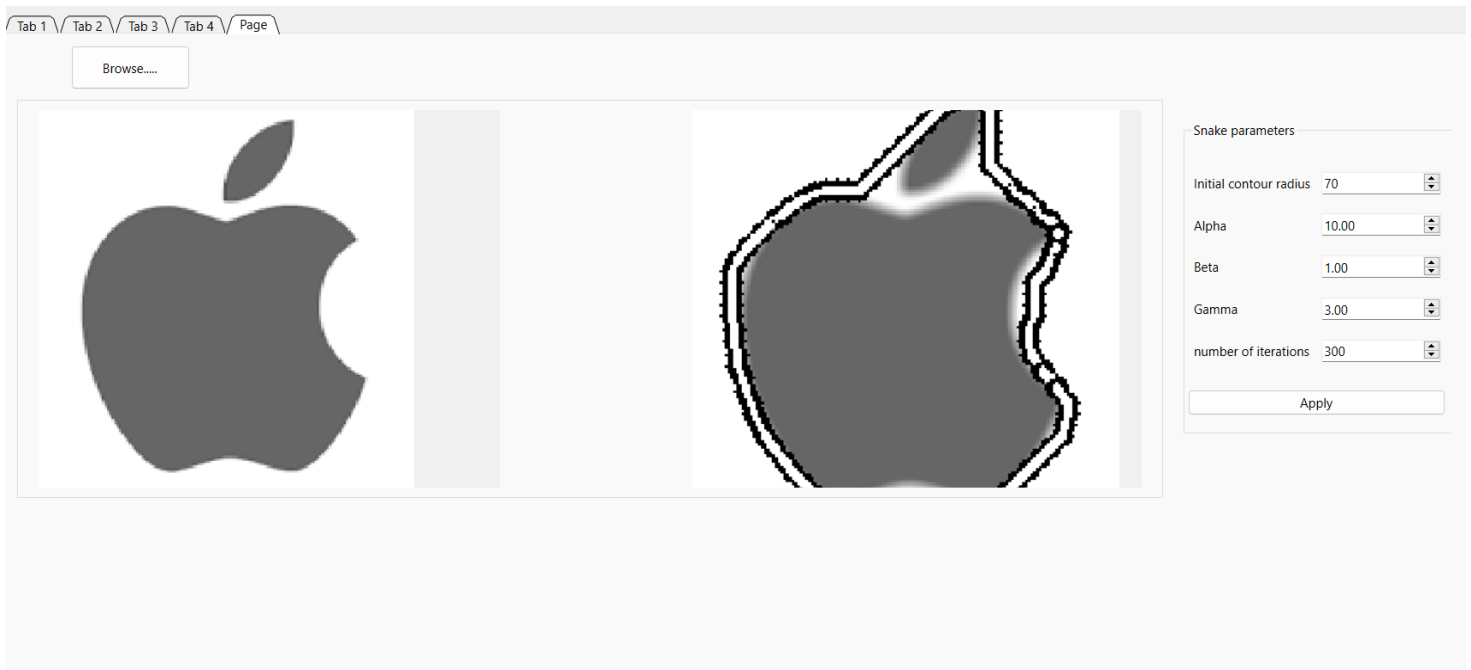
- Initial Contour Radius = 120
- Alpha = 20
- Beta = 1
- Gamma = 3
- No. of iterations = 300



Second Image:

Matching Variables:

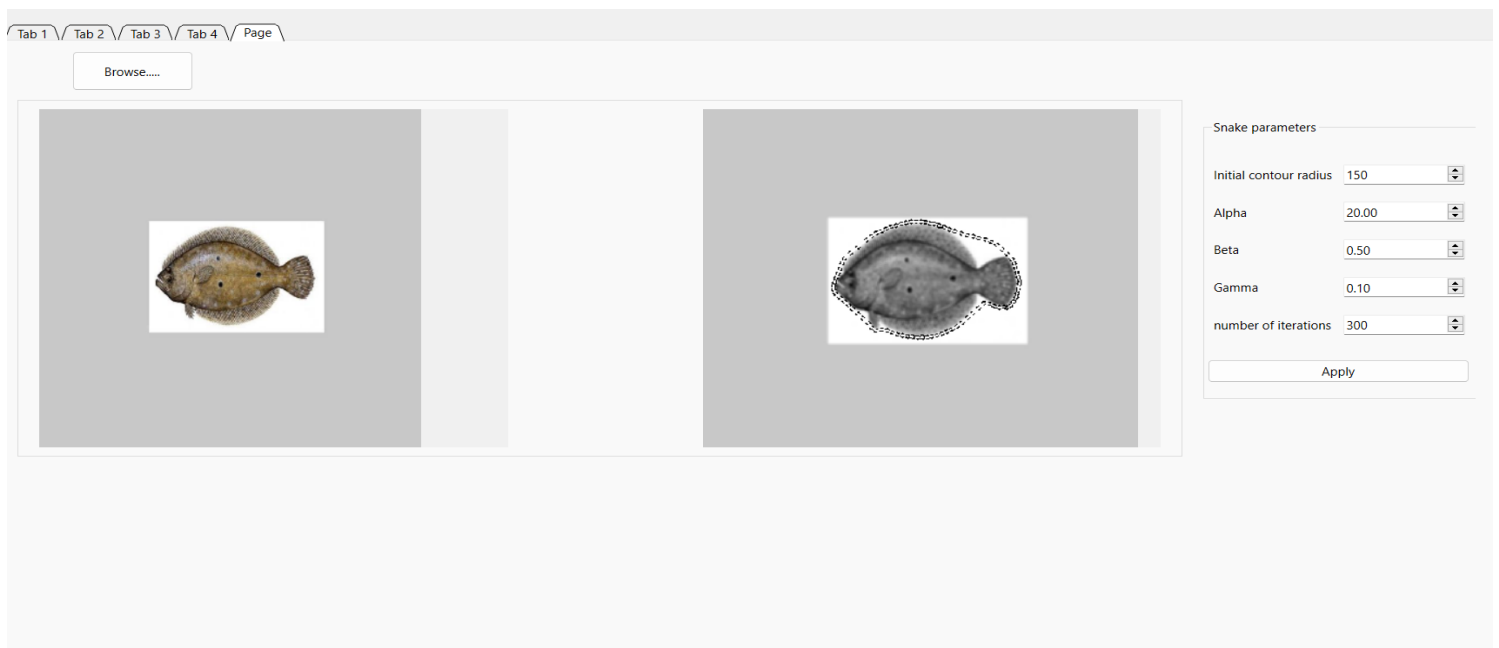
- Initial Contour Radius = 70
- Alpha = 10
- Beta = 1
- Gamma = 3
- No. of iterations = 300



Third Image:

Matching Variables:

- Initial Contour Radius = 70
- Alpha = 10
- Beta = 1
- Gamma = 3
- No. of iterations = 300

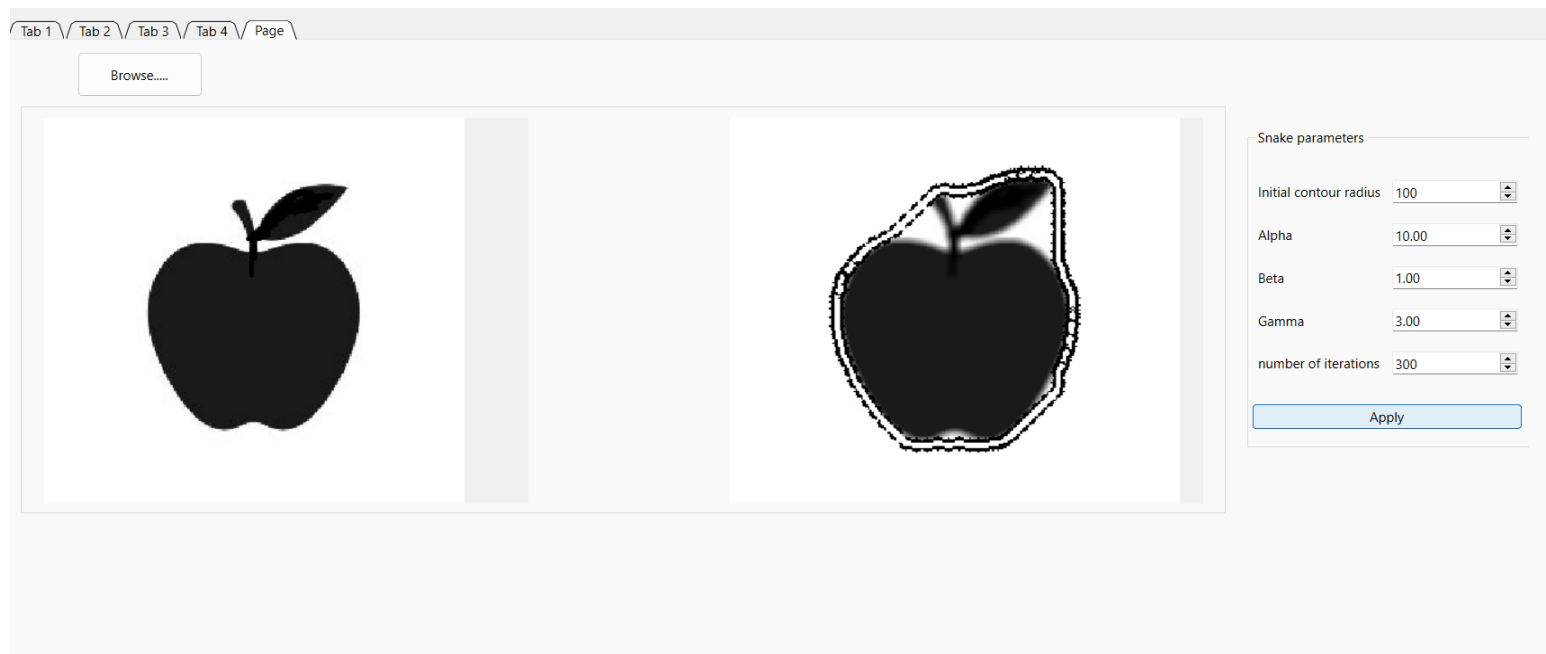


Fourth Image:

Matching Variables:

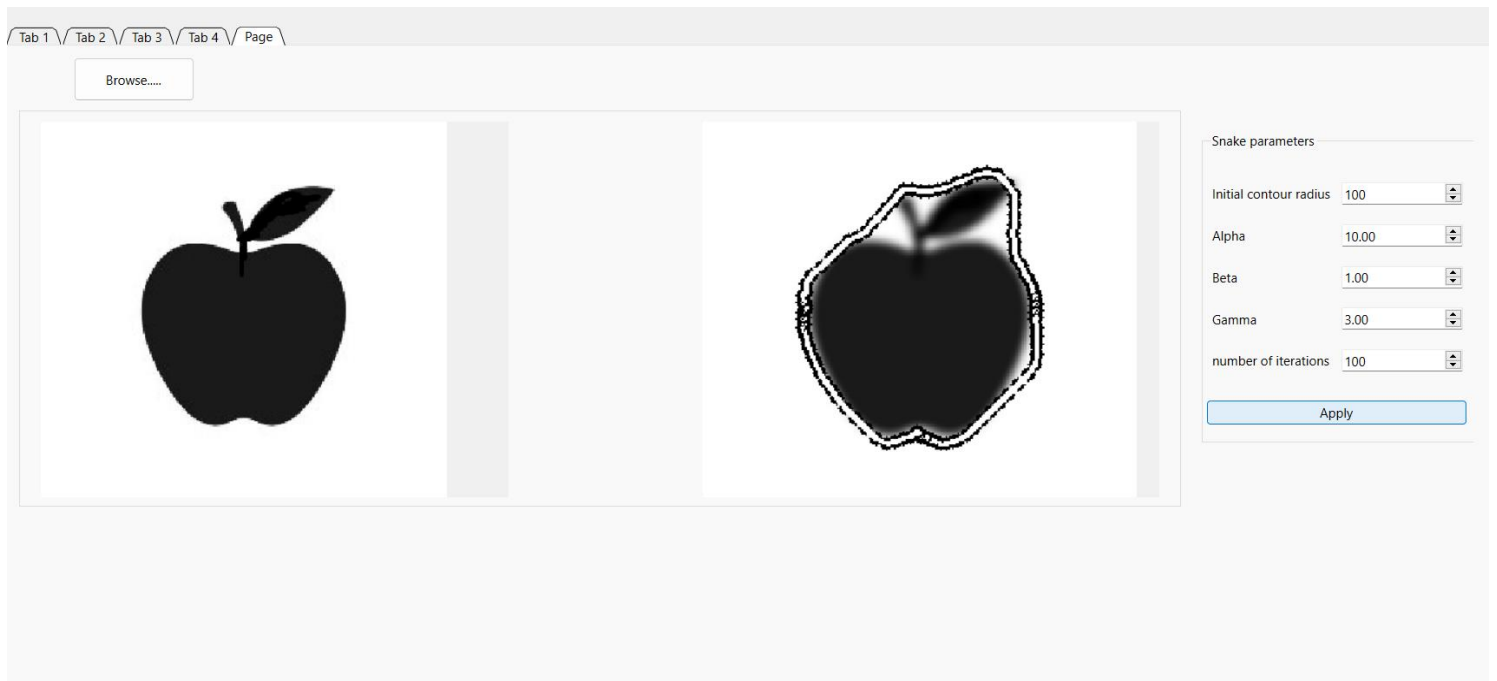
▪ Case 1

- Initial Contour Radius = 100
- Alpha = 10
- Beta = 1
- Gamma = 3
- No. of iterations = 300



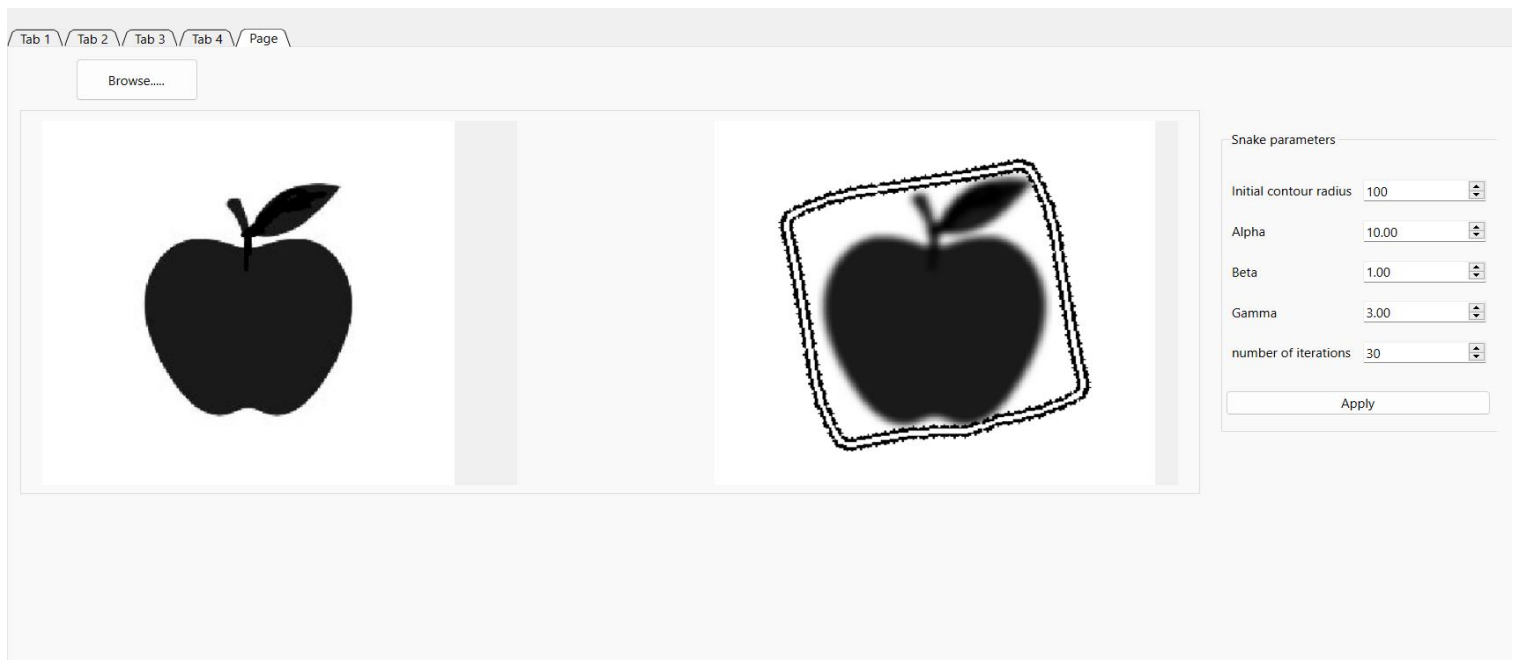
▪ Case 2

- Initial Contour Radius = 100
- Alpha = 10
- Beta = 1
- Gamma = 3
- No. of iterations = 100



■ Case 3

- Initial Contour Radius = 100
- Alpha = 10
- Beta = 1
- Gamma = 3
- No. of iterations = 30



Note

From the above 3 cases you can see that as the no of iterations increases the fit of the contour to our shape becomes much better, however this will reflect to your processing time along with your memory usage, you only get to decide your preferences.