Task 2

Hough Transform & Contouring

Team members:

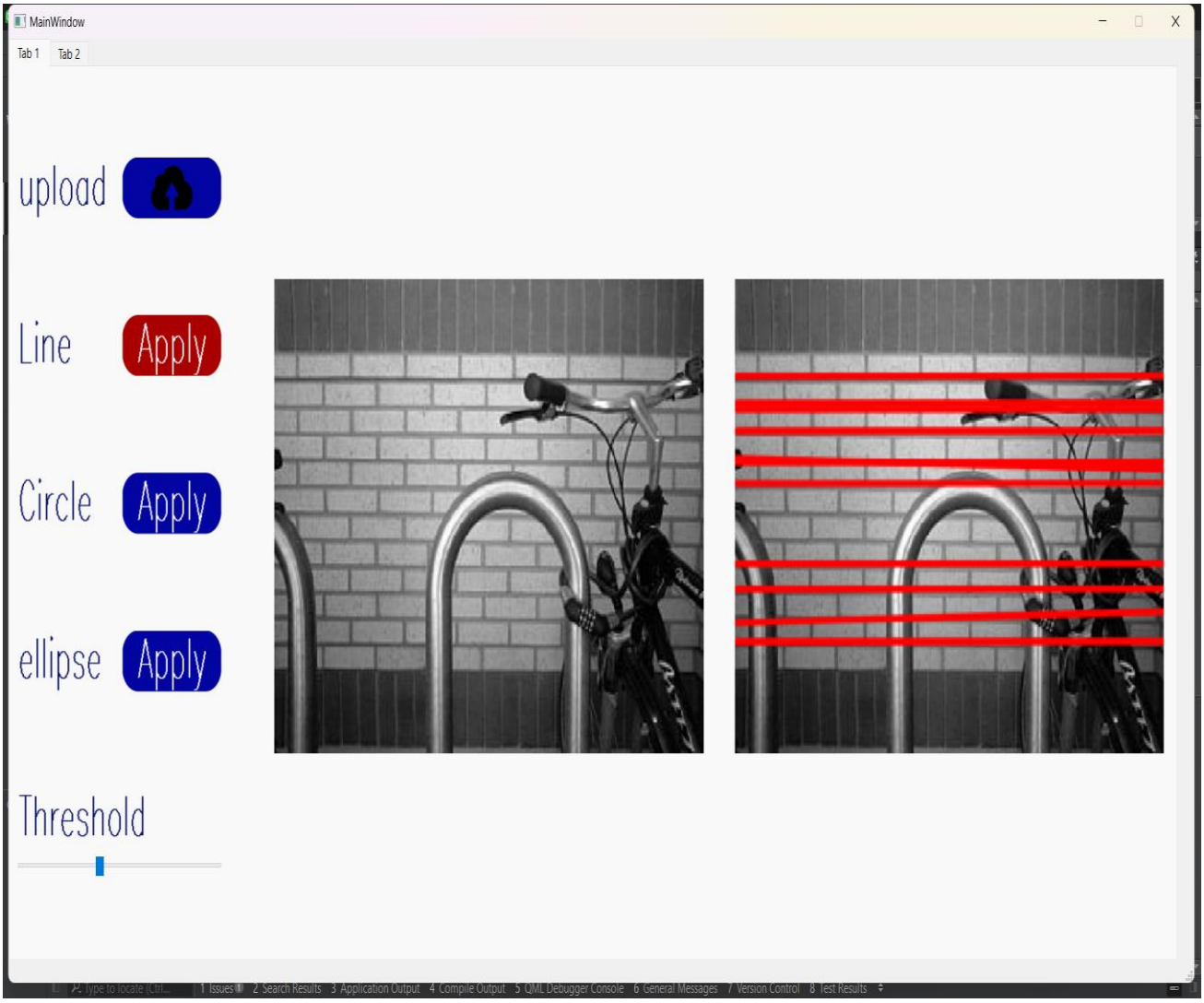
NAME	SECTION	BN
Abram Gad	1	1
Mena Safwat	2	44
Mahmoud Hamdy	2	24
Mariam Hossam Eldin	2	31
Mariam Mohammed Mahmoud	2	34

Contents:

GUI:

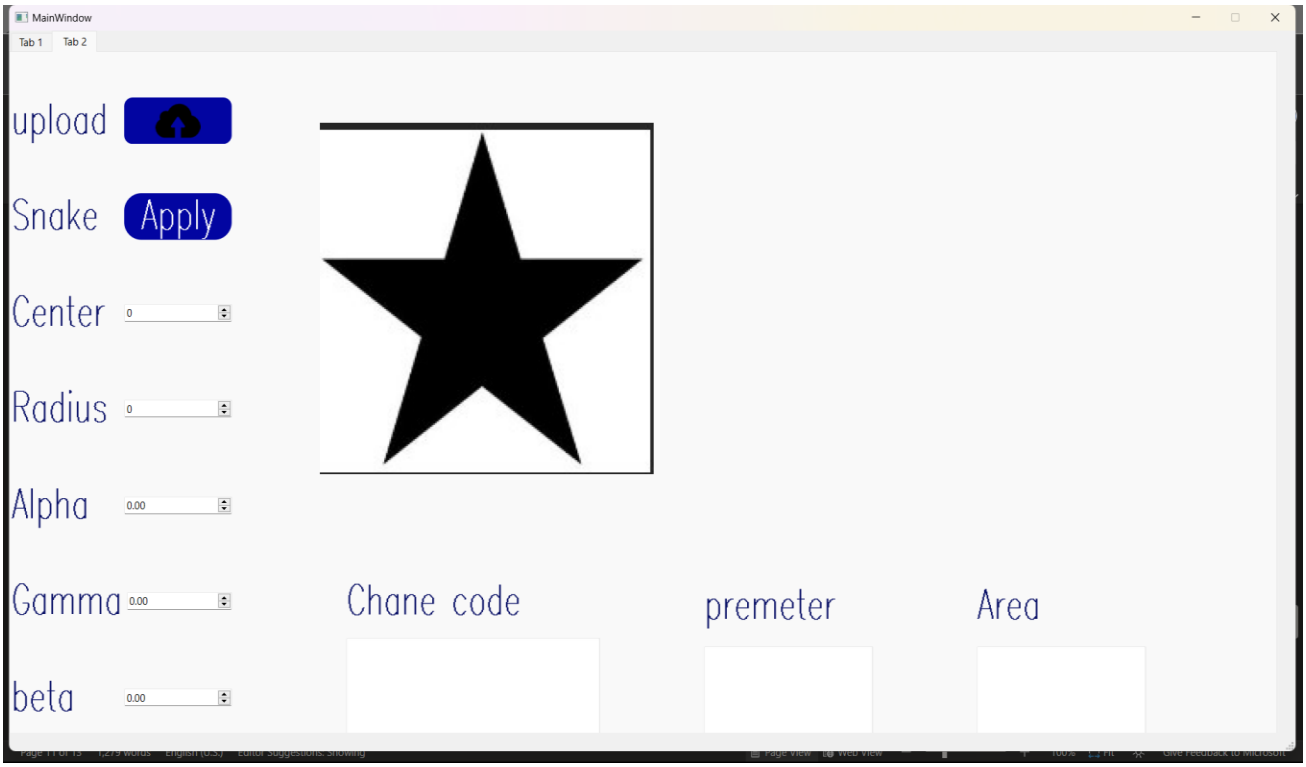
Tab1:

The user can browse for image and choose between three options, first one is applying Hough line, second is apply Hough circle, final option is applying Hough ellipse



Tab2:

The user can browse for image and choose the parameter to apply the snake contouring



Code Functionalities:

1)Hough line.

2)Hough Circle.

3)Hough Ellipse.

2)Snake Contouring.

Hough Transform tap:

After choosing image you need to choose the operation

First operation: Hough line

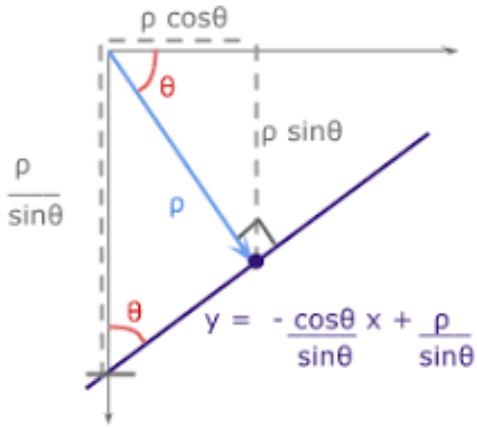
Hough line function

Parameters:

The edges of the image (Using canny edge detection), Threshold value

Algorithm:

The algorithm gets steps of theta and rho as it transforms from image space into polar space to calculate the slope and the intercept and builds the accumulator with dimensions of the length of theta and Rohs as the following image shows:



So, using this equation to calculate the rho and theta after that we use voting technique as we test each point if it satisfies a line then we iterate through the accumulator and choose the rhos and thetas that has votes larger than threshold

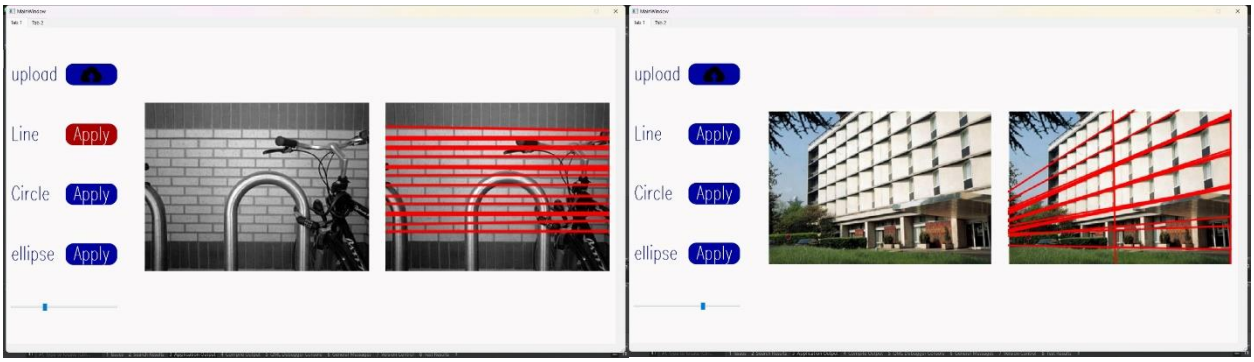
```
vector<pair<float, float>> hough_line_transform(Mat& image, int angle_step, int threshold) {
    int height = image.rows;
    int width = image.cols;
    double diagonal = sqrt(pow(height, 2) + pow(width, 2));
    vector<float> rhos;
    for (double i = -diagonal; i <= diagonal; i += 1) {
        rhos.push_back(i);
    }
    vector<float> thetas;
    for (int i = -90; i < 90; i += angle_step) {
        thetas.push_back(i * CV_PI / 180.0);
    }
    int num_thetas = thetas.size();
    vector<float> cos_t(num_thetas);
    vector<float> sin_t(num_thetas);
    for (int i = 0; i < num_thetas; i += 1) {
        cos_t[i] = cos(thetas[i]);
        sin_t[i] = sin(thetas[i]);
    }
    int accumulator_rows = 2 * diagonal;
    int accumulator_cols = num_thetas;
    Mat accumulator(accumulator_rows, accumulator_cols, CV_8UC1, Scalar(0));
    vector<int> y_idxs;
    vector<int> x_idxs;
    findNonZero(image, x_idxs, y_idxs);
    for (int i = 0; i < x_idxs.size(); i += 1) {
        int x = x_idxs[i];
        int y = y_idxs[i];
        for (int t_idx = 0; t_idx < num_thetas; t_idx += 1) {
            int rho = round(x * cos_t[t_idx] + y * sin_t[t_idx] + diagonal);
            accumulator.at<uchar>(rho, t_idx) += 1;
        }
    }
```

In this code we define the ranges for theta and rho then we build the accumulator, and we select the edges (Nonzero values) and the last 2 for loops where the voting occurs

```
vector<pair<float, float>> lines;
    for (int y = 0; y < accumulator_rows; y += 1) {
        for (int x = 0; x < accumulator_cols; x += 1) {
            if (accumulator.at<uchar>(y, x) > threshold) {
                float rho = rhos[y];
                float theta = thetas[x];
                lines.push_back(make_pair(rho, theta));
            }
        }
    }
```

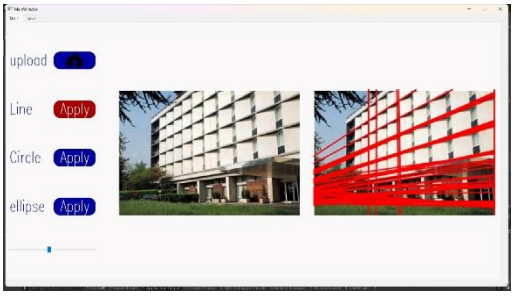
Here where checking for values greater than threshold occur and we store those values in new vector and draw and transform the equation back to image space equation (aka: y = mx+b)

And then draw these lines on the image and show it using imshow

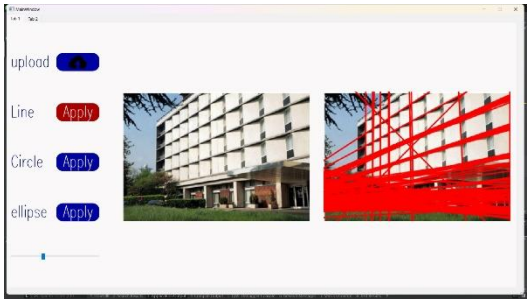


This is the output with threshold about 170

And if we decrease the threshold more lines will be drawn



Second operation: Hough Circle



Hough Circle function:

Parameters:

The edges of the image (Using canny edge detection), Threshold value and Max Radius, Min Radius are Optional.

Algorithm:

Here we search for each point the possible circle it can pass through this point and according to certain threshold

So first, we build the accumulator 3D array with the following dimensions width, height and radius and then we iterate over the edges to get the values of a & b and then we vote according to their values and the radius value after that we iterate again and get the parameters having vote > threshold value

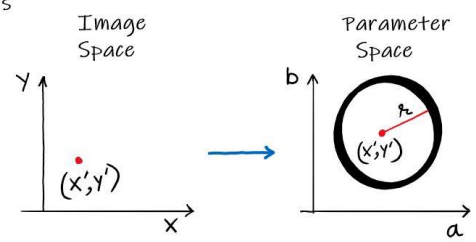
In parametric form, circle can be expressed as

$x = a + r\cos\theta$
$y = b + r\sin\theta$

Similarly, in ab space we can write

$a = x' - r\cos\theta$
$b = y' - r\sin\theta$

This is equivalent to circle with center(x',y')



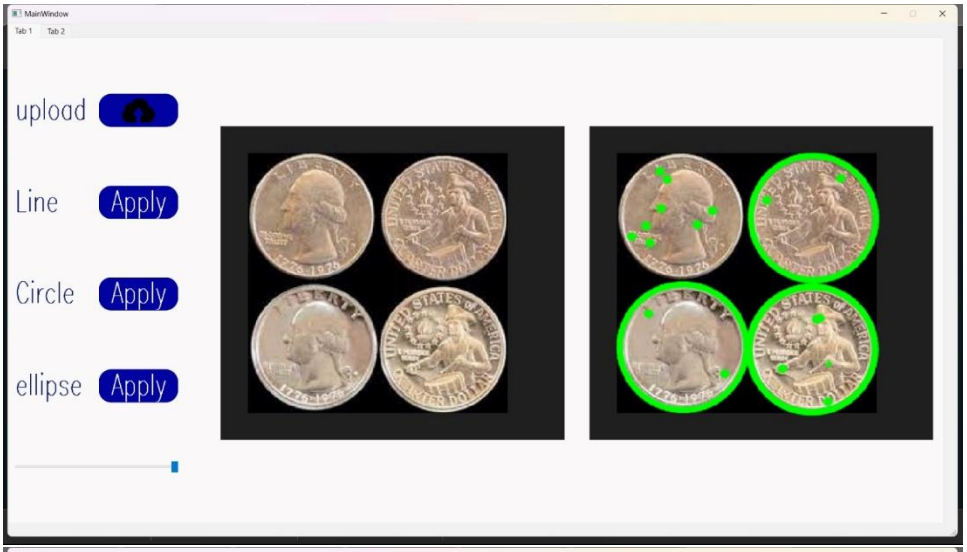
The following code shows the voting process

```
Mat hough_circle(Mat& image, int threshold, int minRad = 10, int maxRad = 50){
    cvtColor(image, image, COLOR_BGR2GRAY);
    Mat edges;
    Canny(image, edges, 50, 150, 3);
    int height = edges.rows; //Get the Height of the image
    int width = edges.cols; //Get the Width of the image
    //Theta
    int theta_range = 360;
    float deg_to_radian = M_PI / 180.0;
    //Get the accumulator
    vector<vector<vector<int>>> accumulator(width, vector<vector<int>>(height, vector<int>((maxRad-minRad+1),0)));
    //Voting
    for(int y = 0; y<height; y++){
        for(int x = 0; x<width; x++){
            if(edges.at<uchar>(y, x) !=0){
               for(int radius = minRad; radius<maxRad; radius++){
                   for(int theta = 0 ; theta<theta_range; theta++){
                       double y_not = y - radius * sin(theta * deg_to_radian);
                       double x_not = x - radius * cos(theta * deg_to_radian);
                       if(x_not<width && x_not>=0 && y_not<height && y_not>=0){
                           accumulator[(int)x_not][(int)y_not][radius-minRad]++;
                       }
                   }
               }
            }
        }
    }
```

And the code for Circle selection according to threshold value

```
//Loop over the accumulator to get the circle based on their threshold value
Mat circles = image.clone();
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        for  (int r = minRad; r < maxRad; r++)
        {
            if (accumulator[x][y][r-minRad] >= threshold)
            {
                circle(circles, Point(x , y ), r - minRad, Scalar(0, 0, 255), 1, LINE_AA);
            }
        }
    }
}
return circles;
```

These are some examples



Third operation: Hough Ellipse

Hough Ellipse function:

Parameters:

The edges of the image (Using canny edge detection), Threshold value

Algorithm:

A. The basic algorithm:

It requires a five-dimensional parameter space as it's clear in the equation:

$$\frac{(x\cos\alpha + y\sin\alpha)^2}{a^2} + \frac{(x\sin\alpha - y\cos\alpha)^2}{b^2} = 1$$

The algorithm takes two different points belonging to the ellipse. It assumes that it is the main axis. A loop on all the other points determines how much an ellipse passes to them. A good match corresponds to high accumulator values. And the following steps are the steps to ellipse detection:

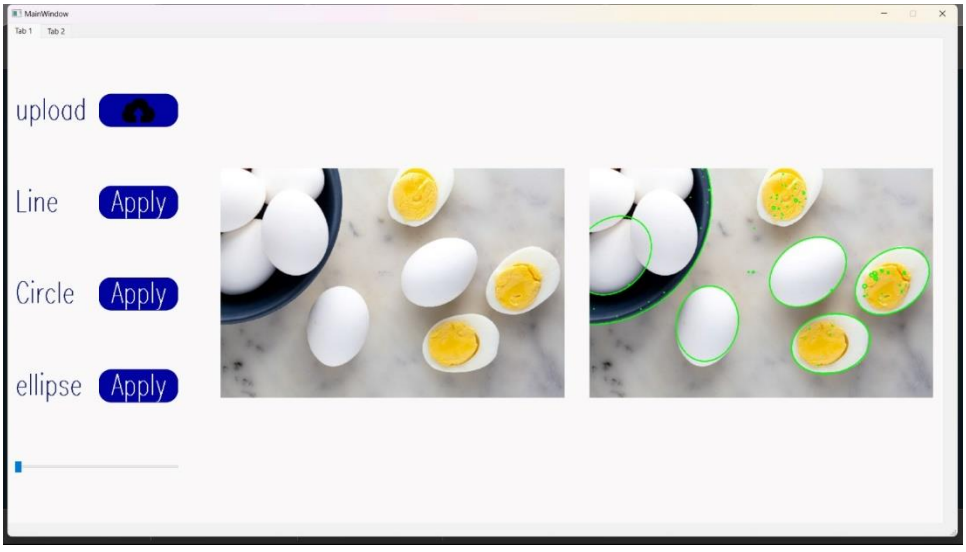
1. **Preprocessing:** The input image is first preprocessed to convert it to grayscale and apply edge detection to obtain a binary edge image. The binary edge image contains all the edge points in the image that correspond to the edges of the ellipse to be detected.
2. **Parameter space:** The Hough ellipse transform represents ellipses in a parameter space, where each point in the parameter space corresponds to a possible ellipse. The parameter space is a 3D array with dimensions `(a, b, theta)`, where a and b are the semi-major and semi-minor axes of the ellipse, and `theta` is the angle of rotation of the ellipse. Each point in the parameter space represents an ellipse with the corresponding values of a, b, and `theta`.
3. **Voting:** For each edge point in the binary edge image, the Hough ellipse transform finds the ellipse parameters that pass through the point. This is done by iterating over all possible values of a, b, and `theta` and computing the corresponding values of x and y for the ellipse equation. If the `(x, y)` coordinates of the point lie on the ellipse, the corresponding element of the accumulator array at the index `(a, b, theta)` is incremented.
4. **Thresholding:** After all edge points have been voted, a threshold is applied to the accumulator array to select the ellipses with the most votes. This is done by iterating over all elements of the accumulator array and selecting the ones with values above

the threshold. The threshold is set based on the maximum number of votes expected for an ellipse in the image.

5. **Non-maximum suppression:** To remove duplicate detections, non-maximum suppression is applied to the selected ellipses. This is done by iterating over the selected ellipses and removing any ellipses that overlap with a larger ellipse. Overlap is determined by computing the intersection of the ellipses and comparing it to the area of the smaller ellipse.
6. **Output:** The final set of ellipses that pass the thresholding and non-maximum suppression steps are returned as the output of the Hough ellipse transform.

B. The Algorithm we used:

It uses the active contour to get the contours in the image so the result we get is a vector of vector of points, where each vector of points represents the contour of an object. Then we loop over the contour and for each contour, it fits an ellipse using the OpenCV function `fitEllipse()`. The result is a `RotatedRect` object that represents the bounding box of the fitted ellipse. Then it calculates the score of the fitted ellipse, which is the absolute difference between the ratio of the width and height of the bounding box and 1. A perfect ellipse would have a ratio of 1, so the score is a measure of how close the fitted ellipse is to a perfect ellipse. It checks if the score is smaller than a threshold value. If it is, it adds the `RotatedRect` object that represents the fitted ellipse to a vector of ellipses. And then returns the chosen ellipses.



Contouring tap:

Active contour algorithm:

A) We start with the initial contouring points that includes the figure to be contoured within it.

We can get those points by two ways:

1)define each point (Explicitly defined)

2)define points that fits on a shape

Here we generate 10 points on a circle.

```
for (int i=0;i<10;i++){
    init_cont[i][0]=(int)(485 + 400 * cos(init[i]));
    init_cont[i][1]=(int)(485 + 400 * sin(init[i]));
    cout<<"("<<init_cont[i][0]<<", "<<init_cont[i][1]<<")\n";
}
```

B) Those points keep on changing their position until it reaches a defined number of iterations or the stability desired.

Here we iterate 100 times.

C) During each iteration according to the energy function that calculates the energy of each point in its neighborhood and its new position is the point with lowest energy.

But how the energy function is calculated:

First it has three components :

–Continuity

–Curvature

–Image (Gradient)

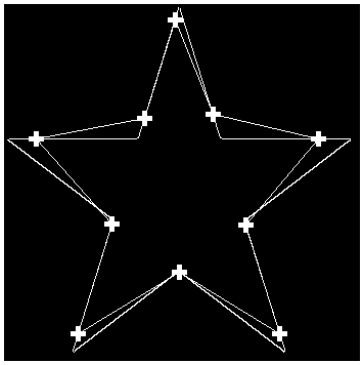
Each of them is weighted by specific parameter so that the total energy can be calculated by the following formula:

$$\text{Total energy} = \alpha \cdot \text{Continuity} + \beta \cdot \text{Curvature} + \gamma \cdot \text{Image}$$

Examples

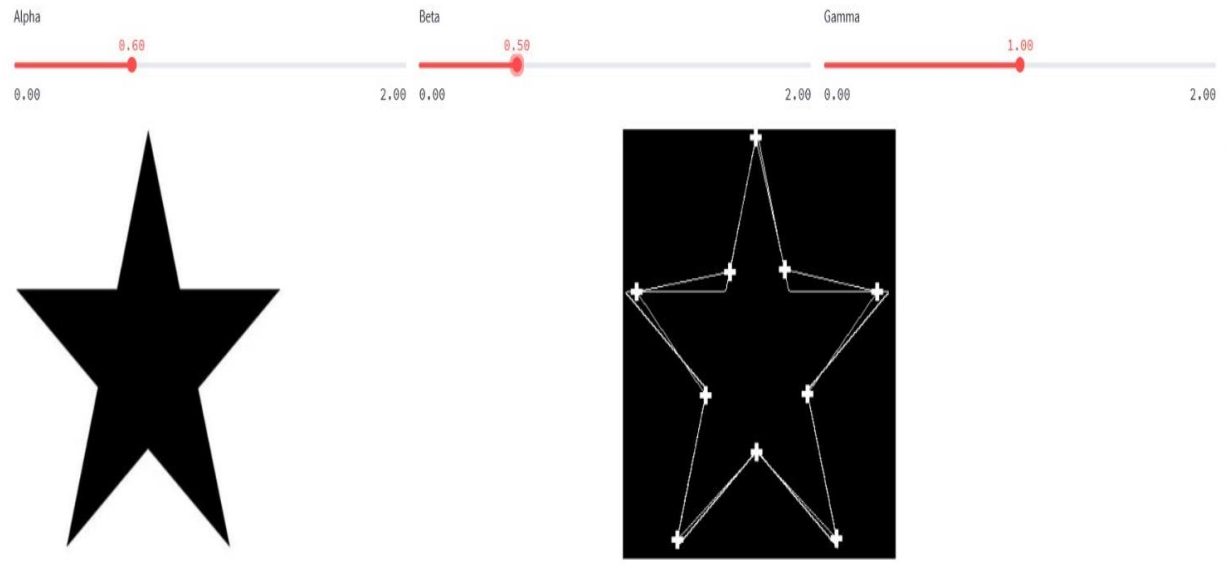
Input:

Output:

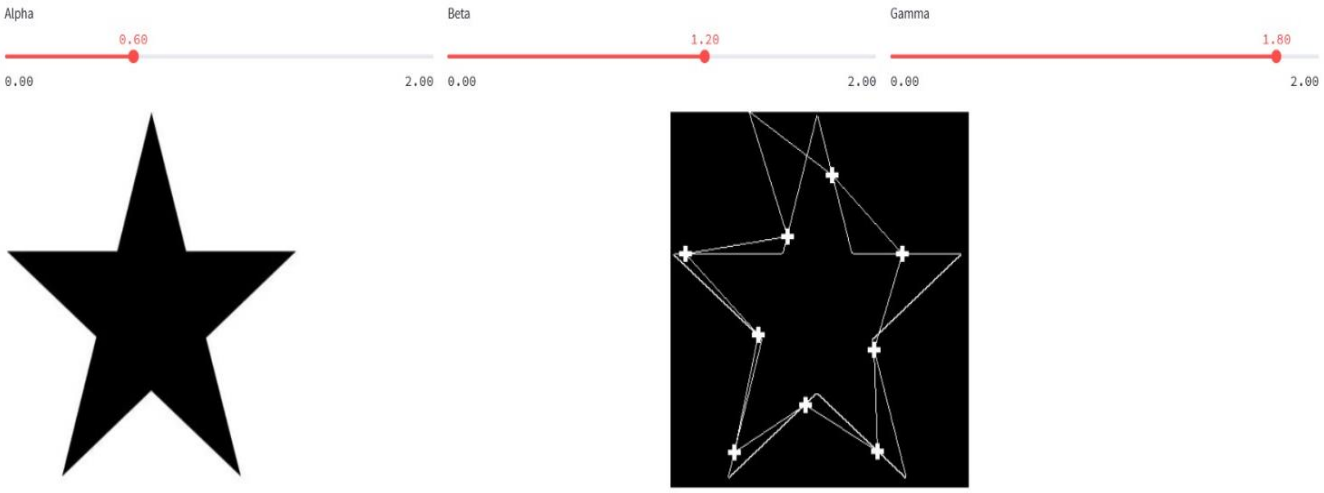


Examples at different alpha, beta and gamma:

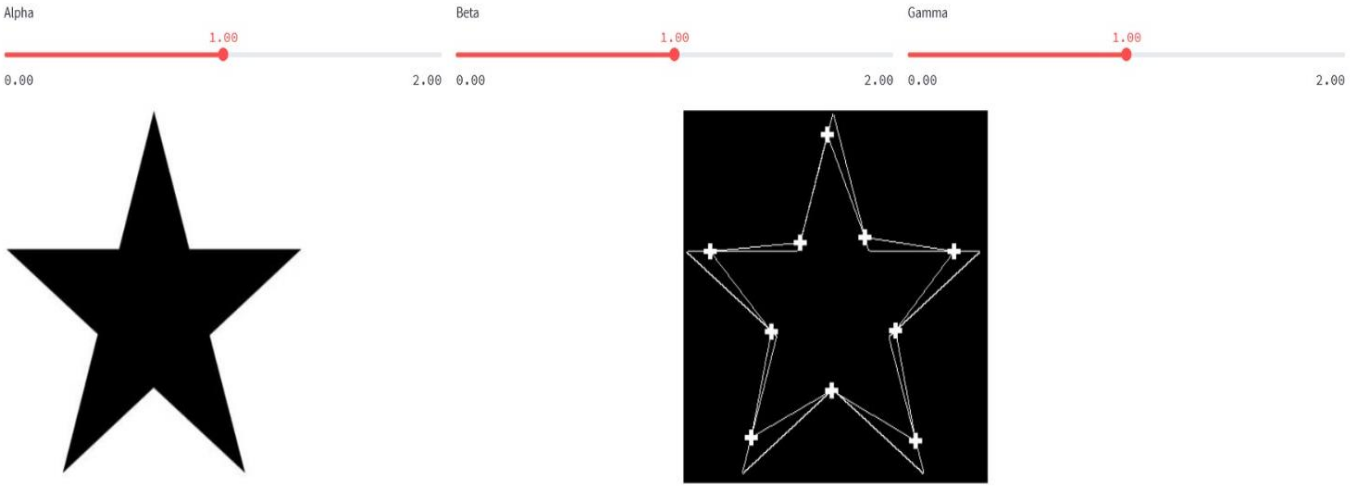
Example 1:



Example 2:



Example 3:

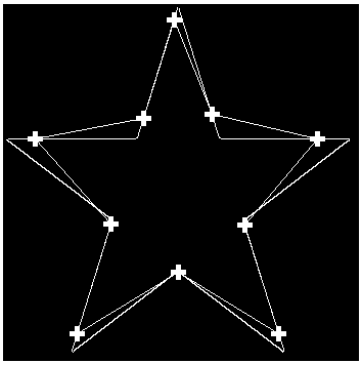


Perimeter and area calculation:

Example:

Input:

Output:



```
intial primeter:528.6761531829834
intial area:9421.0
final primeter:1088.2162322998047
final area:35473.5
```

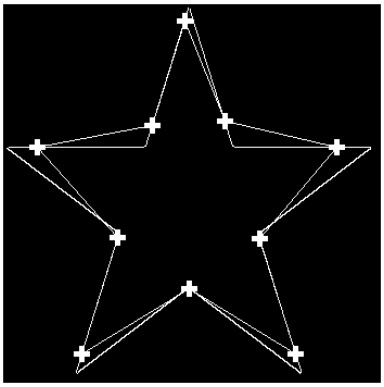
Chain code:

Here we are using 8 chain code representation

Example:

Input:

Output:



Detailed chain code output:

Chain code for the straight line from (109, 206) to (17, 169) is
4545445454454544545445454454544545445454454544545445454454544545445454454544545445454544545445454545445454

Chain code for the straight line from (17, 169) to (113, 139) is
0700700700070070070070070007007007007007000700700700700700070070070070070007007007007000700700700

Chain code for the straight line from (113, 139) to (133, 33) is
667666676666676666766667666667666676666676666766667666676666766667666676666766667666676666766667666676666766667666676666766

Chain code for the straight line from (133, 33) to (216, 107) is
1111011111111011111111101111111101111111101111111101111111101111111101111111101111

Chain code for the straight line from (216, 107) to (323, 74) is
0700700070070070070007007007007000700700700700070070070070007007007007000700700700700700700700700700700

Chain code for the straight line from (323, 74) to (263, 173) is
3232332323323233233232332323323323323323323323323323323323323323323323323323323323323323323323323323323

Chain code for the straight line from (263, 173) to (323, 271) is
1212112112121121121121121121121121121121121121121121121121121121121121121121121121121121121121121121121

Chain code for the straight line from (323, 271) to (217, 238) is
4544544454454454454454454454454454454454454454454454454454454454454454454454454454454454454454454454454

Chain code for the straight line from (217, 238) to (133, 309) is
3334333334333333433333343333334333333433333343333334333333433333343333334333333433333343333334333333433

Final output:

the chain code is

4545445454454544545445454454544545445454544545445454544545445454544545445454544545445454544545445454544

4545445454454540700700700070070070070007007007007000700700700700070070070070007007007007000700700700700

7007000700700700700700070070667666676666676666676666676666676666676666676666676666676666676666676666676

7666676666676666767666676666767666676666767666676666767666676666767666676666767666676666767666676666767

0111111101111111101111111101111111110111111110111070070007007007007000700700700700

0070070070070007007007007007000700700700070070070070070070070070070070070070070070

2332323323323233233232332323323323323323323323323323323323323323323323323323323323

3233232312121121121121121121121121121121121121121121121121121121121121121121121121

1212112121121121121214544544544544544544544544544544544544544544544544544544544544

4454454445445445445445444544544544544544333433333343333334333334333333433333343333

3333343333334333334333333433333433333343333343333334333334333333433333343333334333