Documentation for Neocis Assessment Project

Akram Sbaih – Oct 26, 2019

I used Python for this project for its clear syntax, powerful libraries (numpy and scipy), and to challenge myself to learn pygame as a GUI library in limited time.

Program Structure

Since multiple parts of the program use the same GUI structure, I created a class that uses pygame to view a flexible GUI whenever an instance is made. In each instance, I also set the specialty of the GUI by passing a callback function that runs the algorithms the assessment asks for. By checking through which parameter the function is passed, the GUI knows whether or not to include a 'Generate/Reset' button, or to allow point toggling.

Part 1

The user clicks down on the mouse somewhere in the grid and drags to draw a circle with radius equivalent to the line traced by the drag. The GUI then calls the function specified by 'drag' in part1.py with the coordinates of the starting and ending positions of the drag. That function then determines which points lie on the circle by checking if their distance from the circumference lies within a chosen threshold. It finishes by drawing two other red circles at the minimum and maximum radius of these chosen points, highlighting them. The user can then do a new drag and the program repeats the same.

$$return\ \{\ p\ |\ |\|p - c\| - r| < K\}$$

Where p is any point. C is the center of the circle. R is its radius. K is the threshold.

Part 2

The user toggles any points they want before they hit 'Generate/Reset.' This calls the function that checks if there are enough toggled points to find a matching circle, or not, to reset the window. The vanilla version of this algorithm finds the average position of the toggled points and makes it the center of the matching circle. Then it calculates the average distance of these points from this center and makes it the radius of the matching circle, before drawing it. This solution works in common cases and is cheaper computationally, so it's worth keeping.

$$c = \sum_i \frac{p_i}{n} \qquad r = \sum_i \frac{\|p_i - c\|}{n}$$

Part 2 Optimized

This implementation builds on the last one. However, it goes further by defining a loss function given a center location and a radius. It then uses Scipy to find the optimal circle that minimizes this loss. The loss here is the sum of the squared distance of each point from the circumference of the circle. Since the first implementation provides a reasonable approximation, it can be used as a starter point for the optimizer to be more efficient.

$$loss(c, r) = \sum_i (\|p_i - c\| - r)^2$$

Part 2 Ellipses

This works in a very similar way to the last algorithm. However, we are now trying to find the center of the ellipse (x, y) and its axes (a, b). The loss function here is the sum of the distance squared of each point from the circumference of the ellipse. The vanilla implementation of part 2 helps give a good initial guess of the ellipse as a circle before scipy optimizes on it.

In some cases, scipy fails to find an optimal solution. Therefore, I keep track of the last configuration that scipy tries in 'lparams' to recover at least the closest to optimal solution when failure happens.

This function finds the nearest point on the ellipse:

$$nearest_{a,b}(x, y) = \left(\frac{x}{y}m, m\right) \; for \; m = \frac{a^2 b^2}{\frac{x^2}{y^2}b^2 + a^2}$$

For a and b, the axes of the ellipse centered at (0, 0). The signs of the result are the signs of the input because they are on the same direction from the origin. This formula is derived from the fact that the closest point on the ellipse satisfies its equation and lies on the straight line connecting its center to the point in question.

$$loss\left(a, b, c_{(x,y)}\right) = \sum_i \left\|(p_i - c) - nearest_{a,b}(p_i - c)\right\|^2$$

Notes

It is worth mentioning that the points don't need to be on a grid. Since I use concrete math all the way through. There's a constant on the top of interface.py that allows you (when turned to True) to scatter points randomly on the screen while all the features still work perfectly.

It is also interesting to see how incomplete shapes and edge cases work perfectly.