ME 570: Robot Motion Planning

Homework 1 Report

By Cameron Cipriano

09/14/2021

Question 1.1 code : Polygon.plot()

It has been a while since I have coded in python using NumPy and I don't have too much experience with matplotlib, so this question forced me to get back into these packages and the language itself.

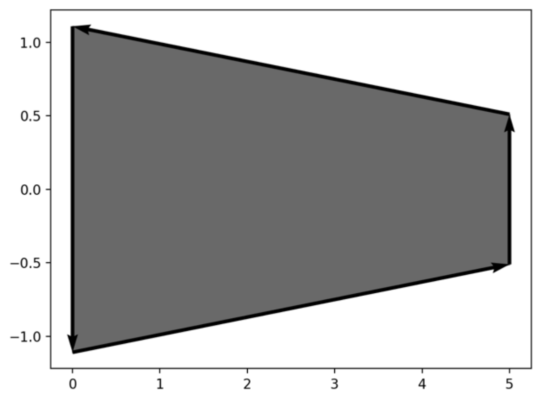

Figure 1.1: Filled Polygon

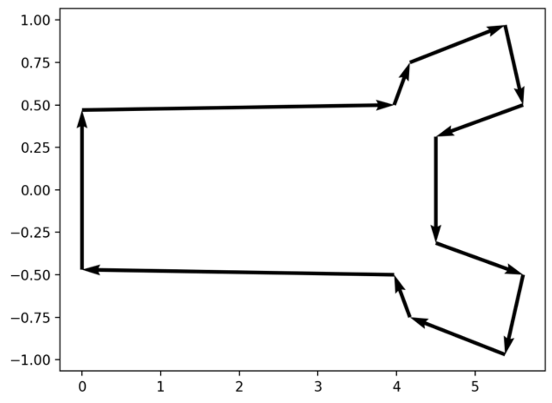

Figure 1.2: Hollow Polygon

Question 1.1 optional : Polygon.is_filled()

For this method, I found a particular formula that roughly calculates the area underneath a line segment, which equates to its average height $(y_2 + y_1)/2$ times its horizontal length $x_2 - x_1$. Summing each of these areas for each edge, we will either obtain a positive or negative value corresponding to the orientation of the polygon. Negative values indicate counter-clockwise orientation while positive indicate clockwise.

$$orientation = \sum_{i=1}^{nb_{vertices}-1} \left[\left(x_{(i+1\,\%\,nb_{vertices})} - x_i \right) * \left(y_{(i+1\,\%\,nb_{vertices})} + y_i \right) \right]$$

$$orientation \begin{cases} if > 0: & clockwise \\ else: & counterclockwise \end{cases}$$

Question 1.1 report : edge_angle description

The variable c_angle is directly computing the cosine angle between the two vectors and s_angle is directly computing the sine of the angle between the two vectors. The first step of the function normalizes the two vectors and divides them by this normal, meaning the two vectors now have unit length 1. With this, the two formulas:

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{|a||b|} \text{ and } \sin\theta = \frac{\vec{a}\times\vec{b}}{|a||b|}$$

We now have $\cos\theta = \vec{a}\cdot\vec{b}$ and $\sin\theta = \vec{a}\times\vec{b}$. Being that we now have the sine and the cosine of the angle, by dividing them, we get the $\tan\theta$. Then making use of the property that $\tan^{-1}\tan\theta = \theta$, we are able to obtain the actual angle.

Question 1.2 code : Edge.is_collision()

My first attempt at this method made use of the orientation of a 3-point line segment to determine whether the two line segments being compared are intersecting.

I had defined in me570_geometry.py is a function called 'orientation' that determined if a line segment defined by 3 points is completely straight (collinear) or curling in the clockwise/counterclockwise direction using the midpoint as the joint. The formula I used to calculate this is derived below:

$$Given\ 3\ points, p1, p2, p3:$$

$$Slopes = \frac{p2_y - p1_y}{p2_x - p1_x}\ \ and\ \frac{p3_y - p2_y}{p3_x - p2_x}$$

Set equal to determine relationship:

$$\frac{p2_y - p1_y}{p2_x - p1_x} = \frac{p3_y - p2_y}{p3_x - p2_x}$$

$$= (p2_y - p1_y)(p3_x - p2_x) = (p3_y - p2_y)(p2_x - p1_x)$$
$$(p2_y - p1_y)(p3_x - p2_x) - (p3_y - p2_y)(p2_x - p1_x) = 0$$

From this, we do not need them to necessarily be equal, rather let the subtraction become $>, <, or = 0$ to determine the result

$$\begin{cases} > 0: & clockwise \\ < 0: & counterclockwise \\ 0: & collinear \end{cases}$$

I soon realized that this method would not entirely work for the edge cases we had to test for, because it would always return the intersection. I then decided to switch to the more accepted version of parametric curves. This is now the version that is implemented in my code.

Question 1.3 code : Polygon.is_self_occcluded()

The most difficult part of this was understanding how to make use of the angle() function to determine what was inside the occlusion zone and what was not and also be orientation agnostic. Involved a lot of test cases.

Question 1.4 code : Polygon.is_visible()

This function was not as challenging as the self-occlusion check as it was mostly about putting existing code together. The check made use of the self_occluded function and a custom function I made in order to test if the vertex-point segment was intersecting among any of the polygon's edges.

Question 1.2 report : polygon_is_visible_test

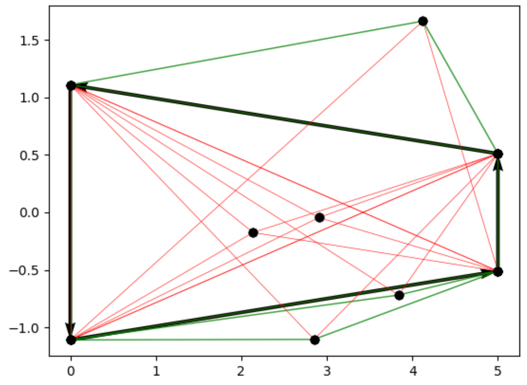

Figure 1: Solid polygon1

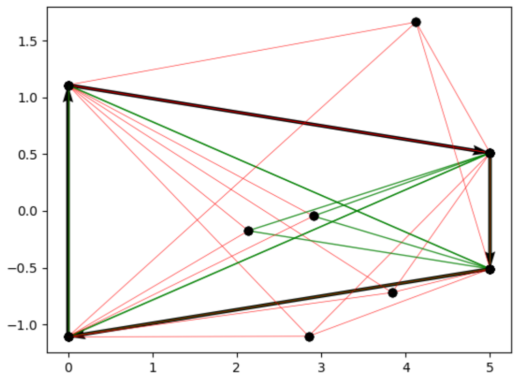

Figure 2: Hollow polygon1

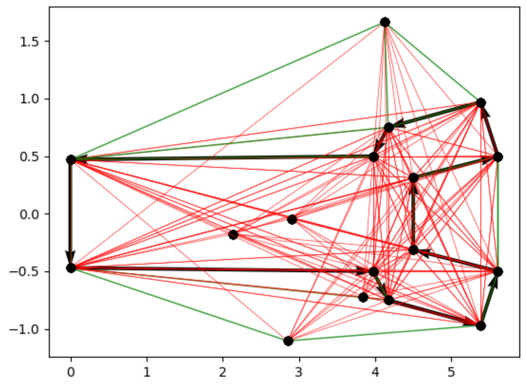

Figure 3: Solid polygon2

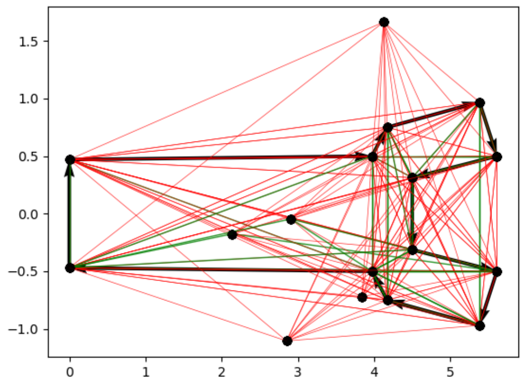

Figure 4: Hollow polygon2

Question 1.5 code : Polygon.is_collision

This method gave me a lot of trouble in trying to figure out how to represent whether a vertex was considered in collision. I had tried many different things, and I think I finally was able to get something that works properly. However, I realized that through the is_visible_test that something may be incorrect with my self_occluded function or is_visible function. I suspect it's the is_visible but I was not able to figure out why some points were considered not-visible/self-occluded.

Question 1.3 report : polygon_is_collision_test

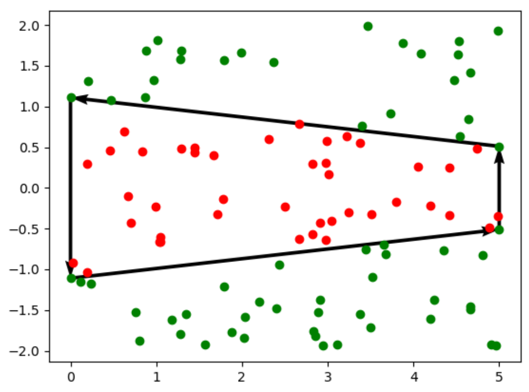

Figure 1: Solid Polygon1 Collisions

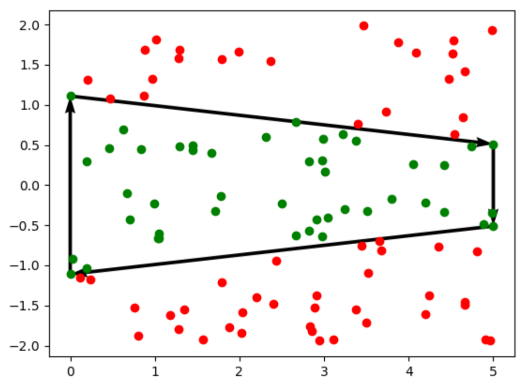

Figure 2: Hollow Polygon1 Collisions

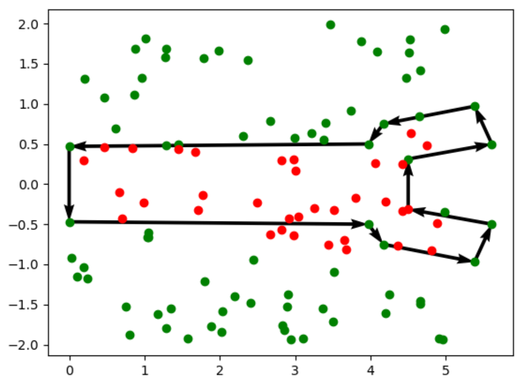

Figure 1: Solid Polygon2 Collisions

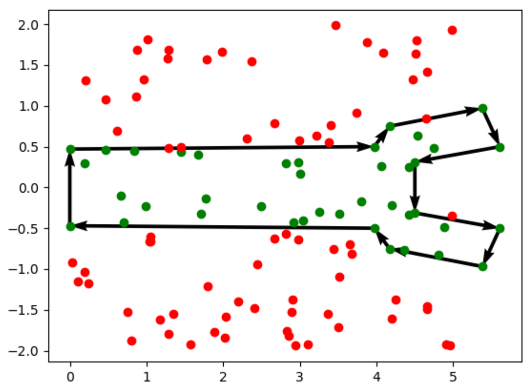

Figure 2: Hollow Polygon2 Collisions

Problem 2: Poor-man's Priority Queue

Question 2.1 code : Priority.insert()

This method was very straightforward

Question 2.2 code : Priority.min_extract()

This method was very straightforward when implemented with a linear search. More advanced methods would be using a min-heap.

Question 2.3 code : Priority.is_member()

This method was very straightforward when implemented with a linear search. Again, a more advanced method would be using a min-heap.

Question 2.1 report : priority_test

Code Output:

```
('Oranges', 4.5), ('Apples', 1), ('Bananas', 2.7)
(Apples, 1)
('Oranges', 4.5), ('Bananas', 2.7), ('Cantaloupe', 3)
Oranges is in my queue? --> True
Milk is in my queue? --> False
Removed: (Bananas, 2.7) --> remaining:  ('Oranges', 4.5), ('Cantaloupe', 3)
Removed: (Cantaloupe, 3) --> remaining:  ('Oranges', 4.5)
Removed: (Oranges, 4.5) --> remaining:
```

Question 2.2 report : Priority Queue

To display all the elements of a grid in descending cost order would be as simple as creating the priority queue through inserting every element of the grid, then finally extracting the minimum element (using min_extract()) until the Priority Queue is empty. This would give you the reversed order of cells that you'd be able to reverse and find the descending order. To make this more efficient, it would make more sense to have a max_extract() version.