1 Winding Number Algorithm

The winding number algorithm was the algorithm that we first built to test insideness. Let C represent a polygon with n vertices. Let V_i be any vertex of C where $0 \leq i \leq n$. We define the winding number as a function W.

$$W(P, C) = \frac{1}{2\pi} \sum_{i=1}^{n-1} \arccos \frac{(V_i - P) \cdot (V_{i+1} - P)}{|V_i - P| \cdot |V_{i+1} - P|}$$

A point $P = (x, y)$ is inside the polygon C, if $W(P, C) = a$ and if $a \, ! = 0$ then P is inside the polygon C. The winding number has a simple implementation from a coding perspective and is very accurate for bounded polygons. When a polygon is unbounded, the winding number will not work without bounding the polygon in some artificial way.

2 Ray Casting Algorithm

The ray casting algorithm at its core works by taking a given point Q and extending that point in the positive x direction. Let us refer to that ray $\overrightarrow{\text{Q}}$. If $Q = (a, b)$, then $\overrightarrow{\text{Q}} = \{a + x | x \geq 0, x \in \mathbb{R}\}$ where $x \in \mathbb{R}$. Let C be a polygon with n verticies, each each line segement formed by connecting vertices we consider to be "walls". If Q lies outside the polygon, the $\overrightarrow{\text{Q}}$ will either not intersect the polygon at all, or there will be at least two intersections which cancel each other out. To cancel each other out means that an intersection will occur when the ray enters the shape and there will be an exiting intersection. Where as, a point that is inside the shape could intersect the shape in multiple places, although there will only be one exiting intersection and no entry intersection. Thus, the algorithm thinks about this as an index. Let us call this index I, I is an integer which will incremented by one on an entry and decremented when exiting the shape. If a point lies outside the shape I will be zero. A point is inside the shape when I is not zero.

2.1 Crossing Number Implementation of Ray-Casting

The crossing number algorithm and all other ray-casting inspired algoirthms are considered applicable to all arbitrary polygons [1]. The winding number depends on floating point numbers to preform the calculations [1]. This could be considered a weakness as the variable typings are create more memory than they should. I will discuss this weakness of the winding number in the optimization section. For now, let's define the crossing number algorithm more mathematically.

The first thing we are going to define is the mathematical significance of the crossing number. To begin, let us discuss the similarities between the Ray Casting Algorithm and the Crossing Number algorithm. They both use this idea of extending a point into a ray. Let $P = (a, b)$ and let $\overrightarrow{\text{Q}} = \{a + x | x \geq 0, x \in \mathbb{R}\}$

where $x \in \mathbb{R}$. Now let C be a polygon and let V_i be any vertex of C where $0 \leq i \leq n$. Let us now define a function CN that takes a vertex of the curve V_i and P as inputs.

$$CN(V_i, P) = \begin{cases} -1 & if\ P\ left\ of\ V_i \\ 1 & if\ P\ right\ of\ V_i \end{cases}$$

For each vertex of the arbitrary polygon C the crossing number is computed for each vertex. If the polygon C has n edges, we can now talk about the intersection index determined by the crossing number. The insideness of the point is determined by the intersection index. The intersection index I is also slightly different to the deffinition that ray-casting uses.

$$I = \sum_{i=1}^{n} CN(P, V_i) + CN(P, V_{i+1})$$

P is inside the polygon if I is an odd number and is outside the polygon if I is even. The idea of ray-casting stays the same amongst the crossing number. What I mean is, crossing number and ray-casting "cancel" out edges based on how $\overrightarrow{\mathrm{Q}}$ collides with C. We choose to calculate the collisons $\overrightarrow{\mathrm{Q}}$ makes with the polygon by incorporating a sense of direction of the ray. There are other implementations that sum I differently depending on the circumstances of the polygon.

Although, crossing number is not perfect. Unbounded polygons still do not work with the crossing number directly. Both with the crossing number and the winding number case, we need to be creative to implement these algorithms for unbounded polygons.

2.2 Optimization of the Crossing Number

We discussed earlier the bounding box implementation. To summerize, the bounding box implementation we discussed counts the amount of entries vs the amount exits and determines if the point is inside the space or not. If the number is an even number of crossings, the point is outside the polygon. If there is an odd number of crossings, the point is inside the figure. When we say "crossings", we are talking about the amount of times that a ray extends in the positive x-axis from the point give intersects the polygon. The bounding box amd crossing numberm in general, the same. So what are the differences? The main differences stem from their implementations. Firstly lets look both algorithms in the python code.

```
@njit(parallel=True)
def bounding_box_algorithm(domain, p, prior_intersections, min_max):
    intersections = prior_intersections
    left_int = False
    right_int = False
    last_intersection = 0
    x_max = min_max[0]
    y_max = min_max[1]
    x_min = min_max[2]
    y_min = min_max[3]
    y_tolerence = .00000001

    if p[0] < x_min or p[1] < y_min:
        return None

    if p[0] > x_max or p[1] > y_max:
        return None

    for pos in range(0,len(domain)):
        if pos < len(domain)-1:
            point_1 = domain[pos]
            point_2 = domain[pos+1]

            w = point_1
            v = point_2

            w_v = (w[0]-v[0], w[1]- v[1])
            p_v = (v[0]-p[0], v[1]-p[1])
            p_w = (w[0]-p[0], w[1]-p[1])

            dot_prod = float(w_v[0] * p_v[0] + w_v[1]*p_v[1])

            # if not_rad > 170 and dot_prod < 1 and dot_prod > -1:
            #     continue

            if v[1] <= p[1] and p[1] < w[1] and dot_prod > 0:
                intersections += 1

            elif w[1] <= p[1] and p[1] < v[1] and dot_prod <= 0:
                intersections -=1

    return intersections
```

```
@njit(parallel=True)
def crossing_number(domain, p, cn):

    for i in range(0,len(domain)-1):
        edge = (domain[i], domain[i+1])

        if (edge[0][1] <= p[1] and edge[1][1] > p[1]) or (edge[0][1] >
             p[1] and edge[1][1] <= p[1]):
            vt = float(p[1] - edge[0][1]) / float(edge[1][1] - edge[0][1])
            if p[0] < edge[0][0] + vt * (edge[1][0] - edge[0][0]):
                cn += 1

    return cn
```

The difference in length is staggering, especially because they are in a sense the same thing. What I want to focus on here is how in the bounding box implementation we use rays rather directly. Whereas, with the crossing number, we don't do as many operations. We just compare and divide floats given to the function. We also do not subtract intersections. The subtracting of intersections could have been one of the causes for the innaccuracies of the algorithm. I believe that the main cause was the handeling of the numbers. In the bounding box algorithm we also define a lot of variables, these deffinitions of could have impacted the accuracy of the numbers.

Before we can discuss the innaccuracies of the numbers we need to discuss what the Njit line above the functions mean. That line is called a decorator. A decorator tells the compiler to reinterpret the function based on what the decorator has in store. In our case, the Njit line dictates to the compiler that the functions are set to be compiled in a more C fashion. In other words, typings are now enforced. Typing is used to tell the compiler how many bits each variable will neeed. In C types are supposed to be explicitly defined before using the variables the user defines. Whereas, in python, that concept does not exist. Python defines the typing of variables based on the context of the code. It's a very impressive technique, many languages have adopted this methodology for compiling their variables. Without typing, code can be writen in a more abstract way. The extra layer of abstraction makes programming simpler and faster to do. As nice as it sounds, there are drawbacks. The biggest one being speed. If the compiler handels all of the typings of variables, they have to almost guess. The fact they have to pretty much guess, means there are a lot of checks as to what variable will work. To make sure they are absolutly sure, the compiler does a lot of tests at runtime before running the code. If the cannot figure out what the variable is, the user will get an error while running the code. This is suppose to act as a check to ensure we don't break the code. Typing variables is important to programers as it determines how many bits the variable will need to be represented. Having more control over variables puts more responsibility on the programmer to know what they are doing. Typing also allows for the code to run faster. What I mean is there are less safety checks at runtime, less checking allows for code to be executed much faster. So, Njit allows use to take advantage the speed of C while retaining the abstractness python inherently gives.

How does Njit impact the numbers within our algorithms? Njit doesn't know if a number is an integer or a double. This is important because doubles use more bits than integers. If you notice, the bounding box algorithm does many more operations with given values. This means there are more chances where the compiler misinterprets the numbers for being different things. If the compiler interprets the output of multiplication of an integer and a double as a integer, we lose a lot of data the double once held. Therefore, the bounding box algorithm has more of a change to lose data.

References

[1] Kai Hormann and Alexander Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131 – 144, 2001.