

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 number of vertices. Let V_i be any vertex of C where $0 \leq i \leq n$. We define the winding number as a function W . A point $P = (x, y)$ is inside the polygon C , if for all V_i on C :

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

If a point is inside the polygon C then the winding number has to be a non-zero integer. We decided to use this algorithm as a way to approximate a circle by increasing the amount of vertices on the polygon.

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. Lets call that ray \vec{Q} . If $Q = (a, b)$ where a and b are constants, then $\vec{Q} = \langle x, b \rangle$ where $x \in R$. Let C be a polygon with n vertices, each each line segment formed by connecting vertices we consider to be "walls". If Q lies outside the polygon, the \vec{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 a index. Let's call this index I , I is an integer which will incremented by one on an entry and decremented when exiting the shape. I if a point lies outside the shape I will be zero. A point is inside the shape when I is not zero.

2.1 Bounding Box Implementation

The pure ray casting algorithm's issues arise when the amount of vertices becomes large. A large number of vertices slow down the algorithm too much. The Ray casting algorithm was designed for polygons of smaller amounts of vertices. We needed a scalable algorithm like the winding number to compute insideness. Thus, we needed to adjust the ray-casting algorithm. Rather than extending the a point Q only in the positive x direction, we extend the point in both directions. An intersection from an extended point in the negative x direction will subtract one from the intersection index. Where as an intersection from the positive x direction adds one to the intersection index. Thus, a point lies within a polygon if the intersection index is zero. We also throw out all points that are greater than the maximums of the shape or lower than the minimums of the polygons. Throwing out those points creates a box around the shape. Points within that box will be the only points tested.

2.2 Optimized Bounding Box Implementation (Crossing Number)

We discussed earlier the bounding box implementation. To summarize, 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 given intersects the polygon. The bounding box and crossing number in general, the same. So what are the differences? The main differences stem from their implementations. Firstly let's look at 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_tolerance = .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 inaccuracies of the algorithm. I believe that the main cause was the handling of the numbers. In the bounding box algorithm we also define a lot of variables, these definitions could have impacted the accuracy of the numbers.

Before we can discuss the inaccuracies 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 need. 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 written 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 handles 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 absolutely 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