Winding Number Algorithm

The winding number algorithm was the algorithm that we first built to test insideness. We define the winding number as a function f. A point $P = (x, y)$ where $x, y \in \mathbb{R}$ is inside the closed curve C, if for all V_i in C where $0 \leq i \leq n-1$:

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

Code

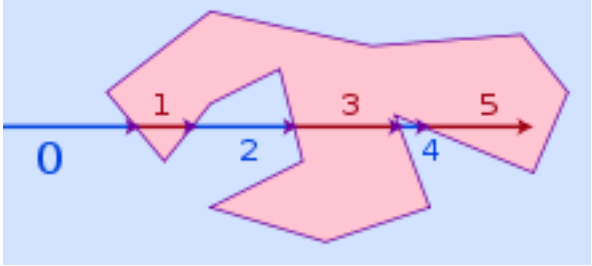
```
def winding_num(p,domain,total, min_max):

    x_max = min_max[0]
    y_max = min_max[1]
    x_min = min_max[2]
    y_min = min_max[3]

    if p[0] < x_min or p[1] < y_min:
        return 0

    if p[0] > x_max or p[1] > y_max:
        return 0

    for pos in range(0,len(domain)):
        if pos < len(domain)-1:
            point_1 = domain[pos]
            point_2 = domain[pos+1]
            vector_diff_1 = (p[0] - point_1[0], p[1]- point_1[1])
            vector_diff_2 = (p[0] - point_2[0], p[1]- point_2[1])
            dot_prod = vector_diff_1[0]*vector_diff_2[0] + vector_diff_1[1]*vector_diff_2[1]
            vector_length_1 = math.sqrt(vector_diff_1[0]**2 + vector_diff_1[1]**2)
            vector_length_2 = math.sqrt(vector_diff_2[0]**2 + vector_diff_2[1]**2)
            denom = vector_length_1*vector_length_2
            value = float(dot_prod/denom)
            calculation = np.arccos(value)
            total += (1/(2*np.pi))*calculation
    return total
```

If a point is inside the curve C then the winding number has to be greater than or equal to 1.

Ray Casting Algorithm

The ray casting algorithm works by extening a given point P to the right infinitly. Lets call that ray Q. The approach here is to think of one point as an infinite

ray. If the point lies outside the polygon, that implies that it should intersect with a polygon a even amount of times. If it is inside the figure the amount of intersections should be odd.



The figure illustrates the algorithm in pieces. The blue line shows once the vector is outside the shape it is even. The red vector illustrates the case where the points are inside and have an odd count of intersections.

Code

```
def ray_casting_alg(domain, p, prior_intersections, min_max):
    global _eps
    global _huge
    global _tiny

    intersect = prior_intersections
    x_max = min_max[0]
    y_max = min_max[1]
    x_min = min_max[2]
    y_min = min_max[3]

    if p[0] < x_min or p[1] < y_min:
        return 0

    if p[0] > x_max or p[1] > y_max:
        return 0

    for pos in range(0, len(domain)):
        if pos < len(domain)-1:
            p_1 = domain[pos]
            p_2 = domain[pos+1]
            if p_1[1] > p_2[1]:
                p_1 = p_2
                p_2 = p_1
            if p[1] == p_1[1] or p[1] == p_2[1]:
                p = (p[0], p[1] + _eps)
```

```
            if (p[1] > p_1[1] or p[1] <  p_2[1]) or (p[0] > max(p_1[0], p_2[0])):
                pass

            if p[0] < min(p_1[0], p_2[0]):
                intersect += 1
            else:
                if abs(p_1[0] - p_2[0]) > _tiny:
                    m_red = (p_2[1] - p_1[1]) / (float(p_2[0] - p_1[0]))
                else:
                    m_red = _huge
                if abs(p_1[0] - p[0]) > _tiny:
                    m_blue = (p[1] - p_1[1]) / (float(p[0] - p_1[0]))
                else:
                    m_blue = _huge
                if m_blue >= m_red:
                    intersect += 1
    return intersect
```