Basic computational geometry algorithms

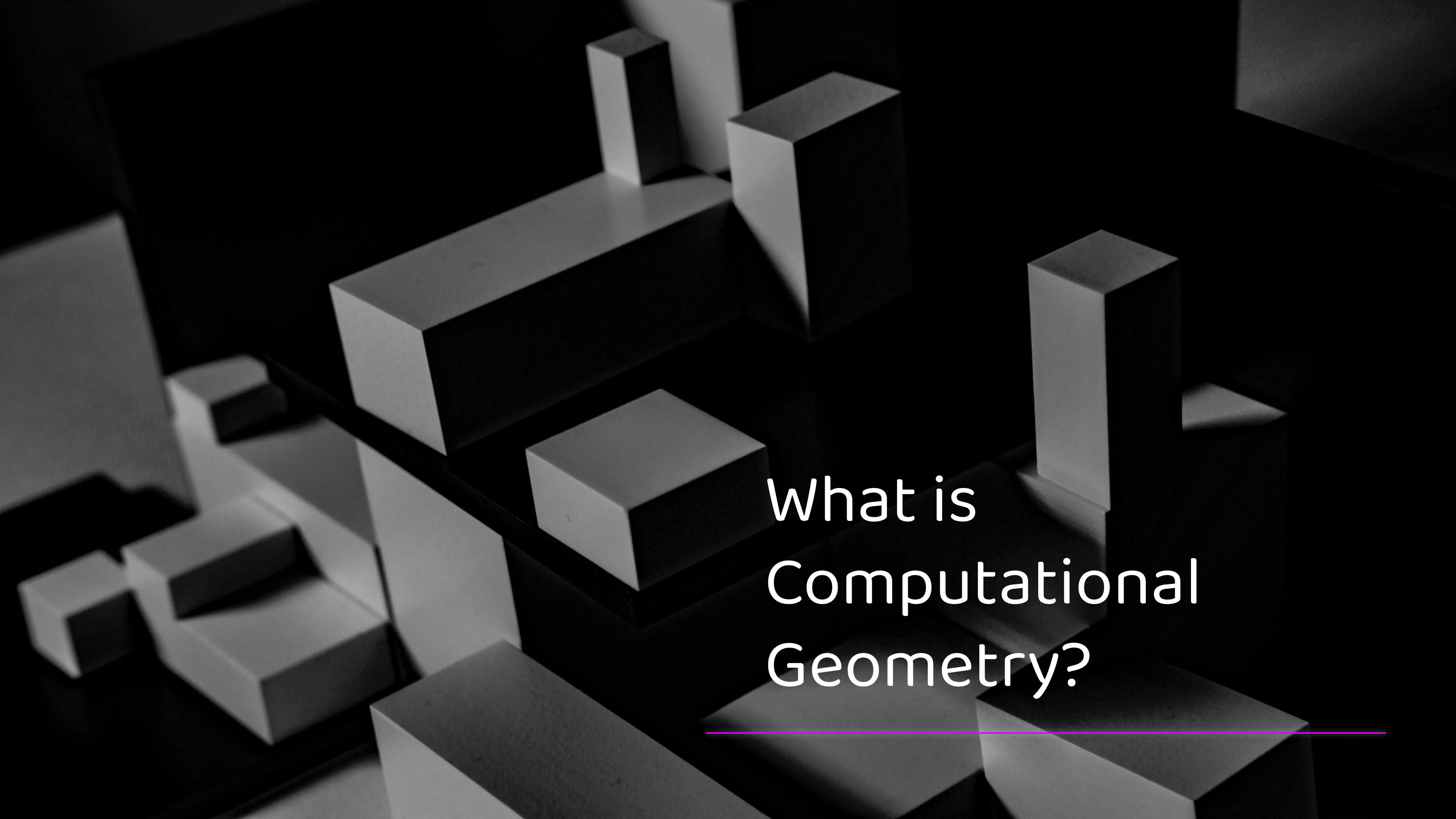
Index

About me

- Software Developer at Instituto Atlântico
- MSc student at Universidade Federal do Ceará - UFC
- Loves Computer Graphics and crochet

Code

Available online on Github and as a formatted notebook



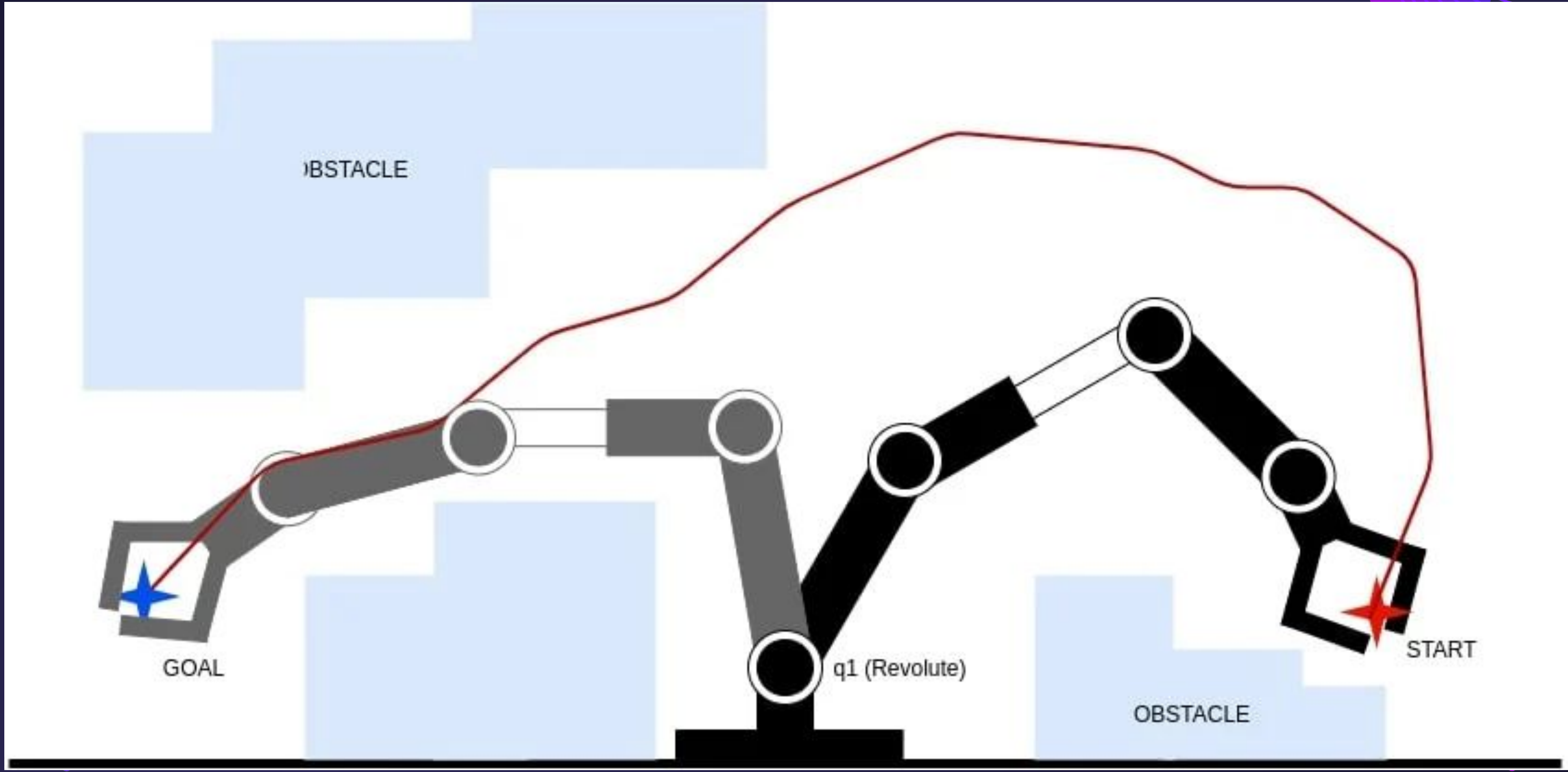
What is Computational Geometry?

From Wikipedia, “Computational geometry is a branch of computer science devoted to the study of algorithms which can be stated in terms of geometry.”

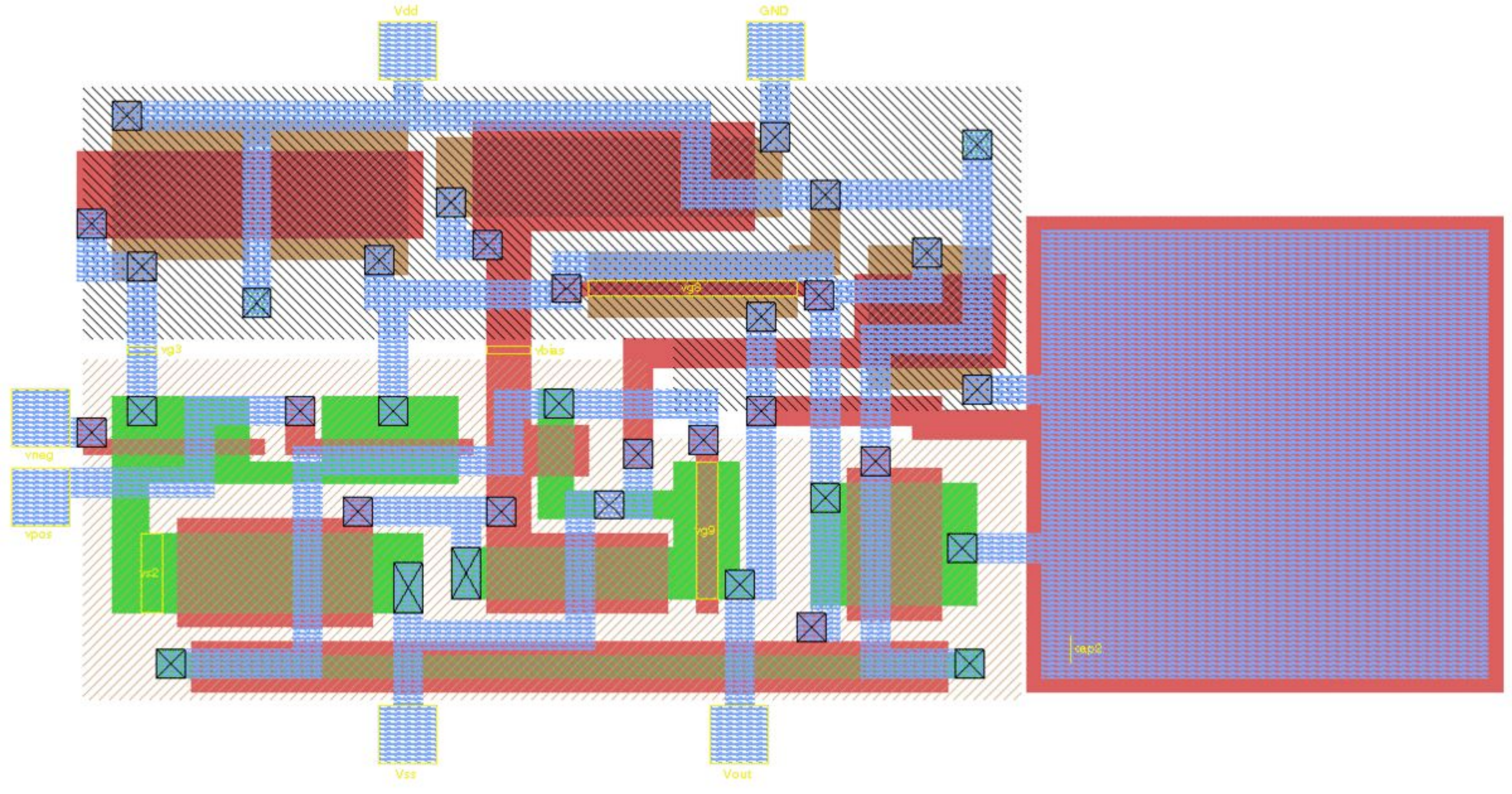
It started as a discipline with the progress of computer graphics and display technology, however, it has grown to encompass many different applications, like **robotics** (motion planning), **integrated-circuit design** (IC geometry design and verification) and **computer vision** (3D reconstruction).

Computational complexity is central to computational geometry, owning to the common sizes of input data, which can be millions of points. In this context, the difference between a $O(n^2)$ and a $O(n\log n)$ algorithm might mean days of computation.

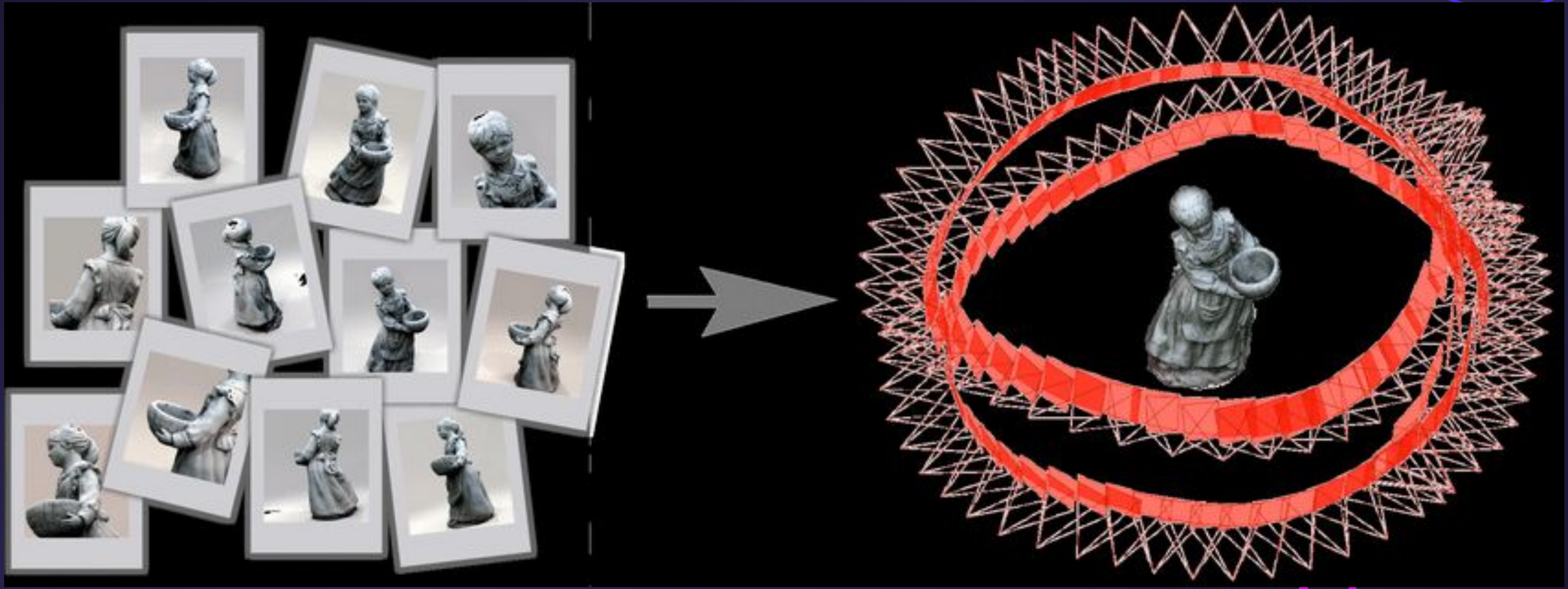
Motion planning



IC Design

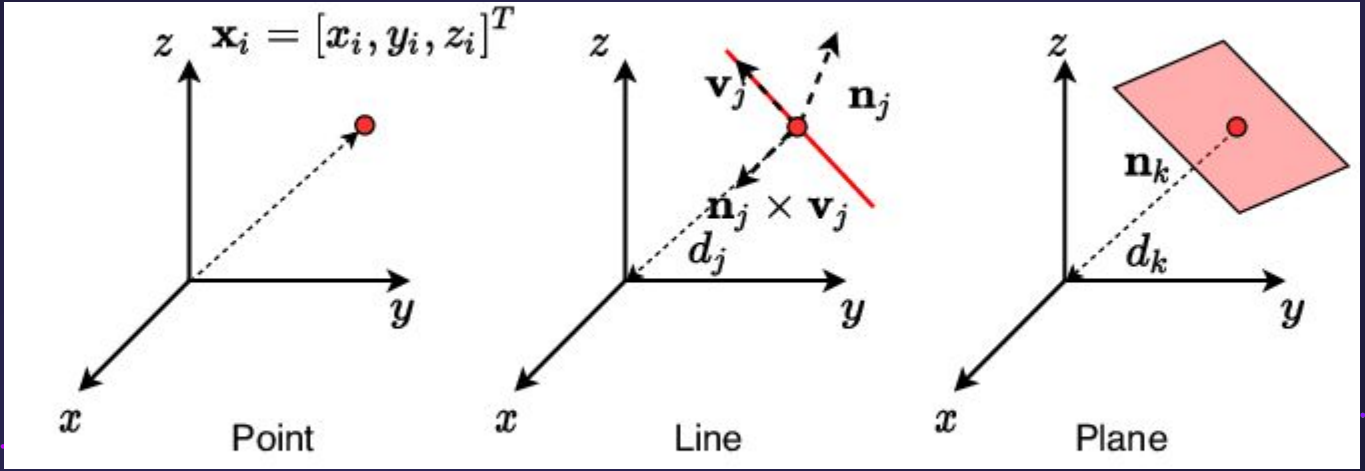


3D reconstruction

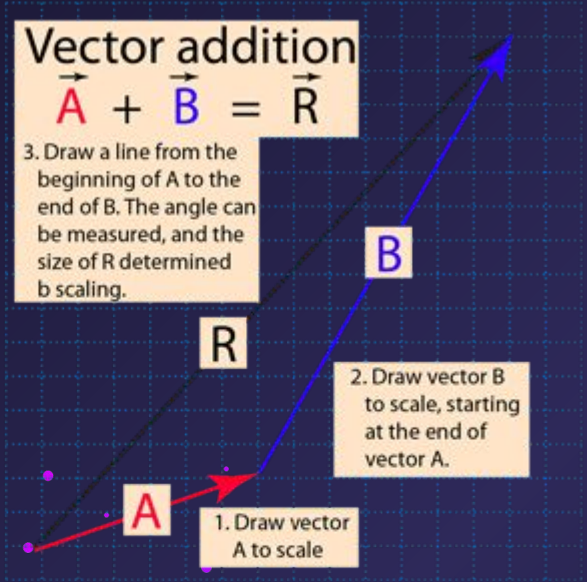


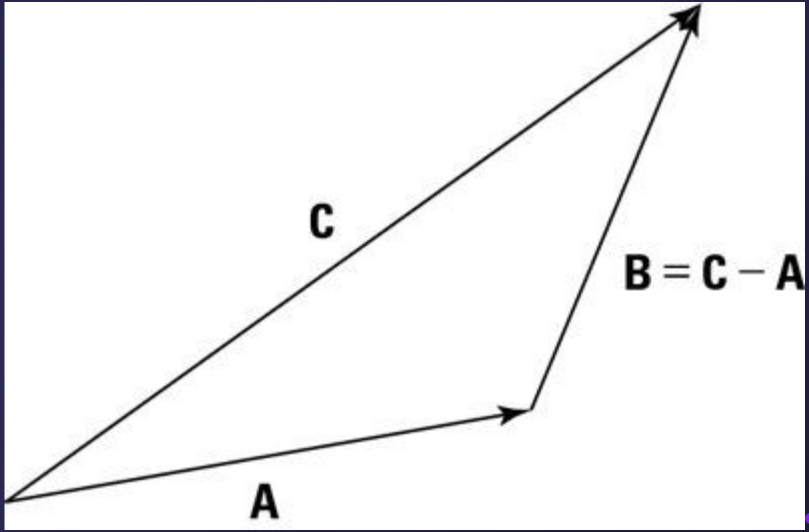
Basic entities and algorithms

Basic entities for Computational Geometry

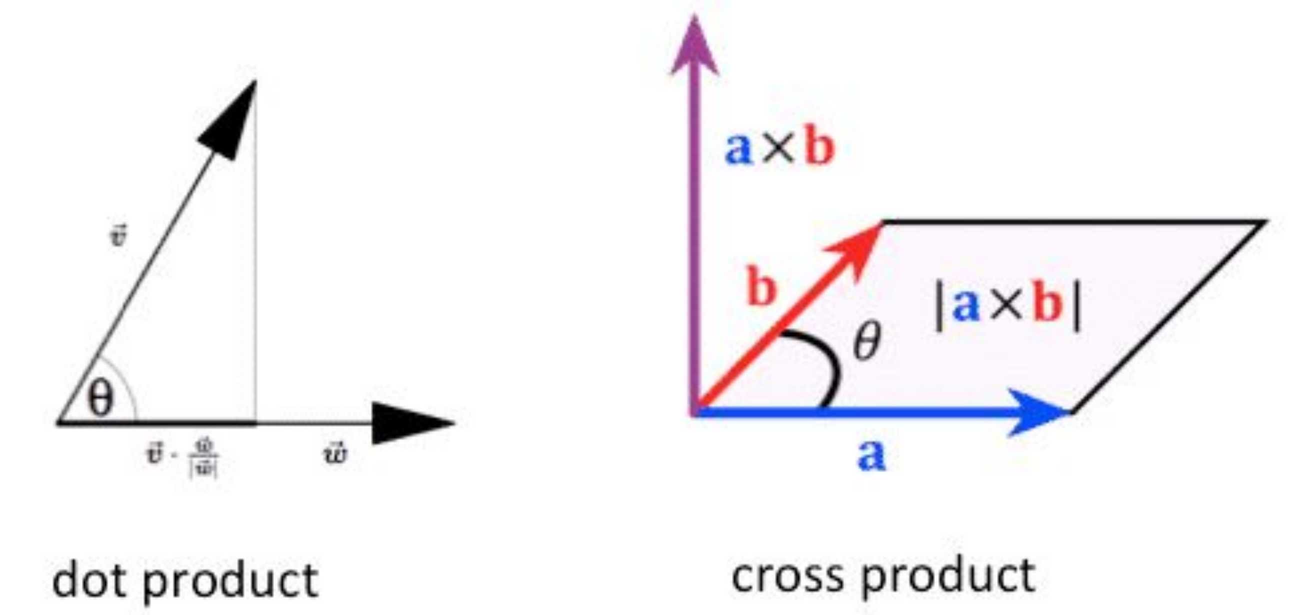


Basic operations





Basic operations

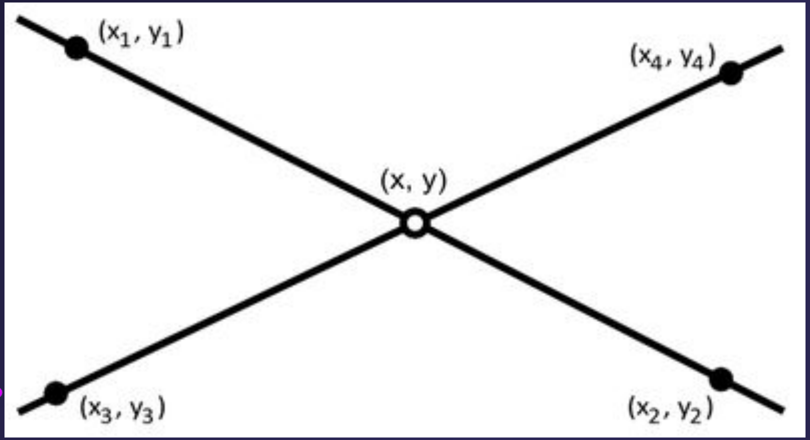


Basic operations

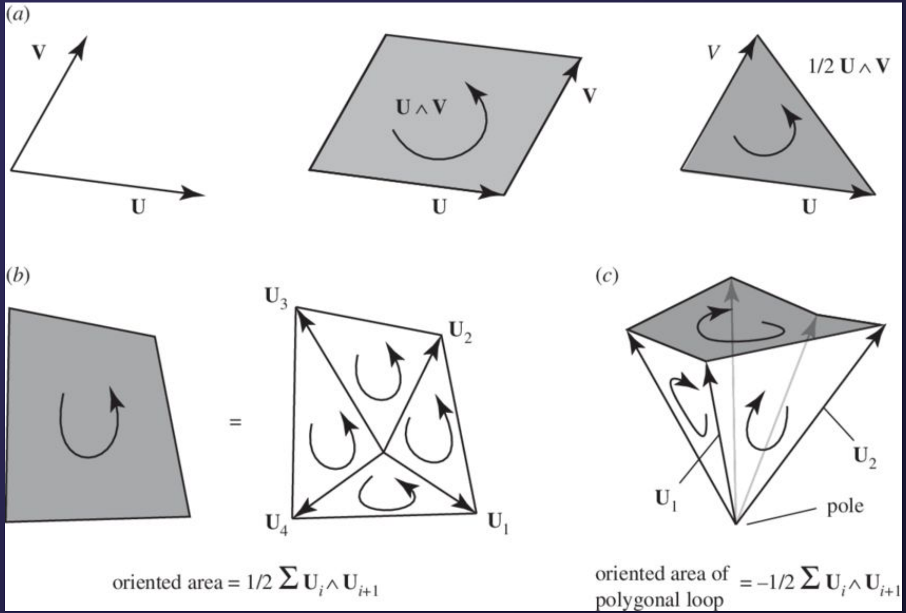
```
def vec_sum(a,b):
    return np.array([a+b for [a,b] in zip(a,b)])

def vec_subtract(a,b):
    return np.array([a-b for [a,b] in zip(a,b)])

def vec_dot(a,b):
    return sum([a*b for [a,b] in zip(a,b)])

def cross(a,b):
    return a[0]*b[1] - a[1]*b[0]
```

Cross Product algorithms

Line-line intersection



Oriented area



Cross Product algorithms

```
def intersect(a,b,c,d):
    ab = np.subtract(b, a)
    ac = np.subtract(c, a)
    ad = np.subtract(d, a)

    cd = np.subtract(d, c)
    ca = np.subtract(a, c)
    cb = np.subtract(b, c)

    p1 = cross(ab, ac) * cross(ab, ad)
    p2 = cross(cd, ca) * cross(cd, cb)

    return p1 < 0 and p2 < 0
```

```
def oriented_area(a,b,c):
    o = np.array([0,0])

    oa = np.subtract(a,o)
    ob = np.subtract(b,o)
    oc = np.subtract(c,o)

    return 0.5 * (cross(oa, ob) + cross(ob, oc) + cross(oc, oa))
```

Pseudo-angles

A pseudo-angle is a function defined over vectors that monotonically increases according to that vector's angle.
We use pseudo-angles to compare between vectors without having to compute the angle itself (which can be expensive).
Some possible pseudo-angles are the square pseudo-angle and the 1 minus cosine pseudo-angle.

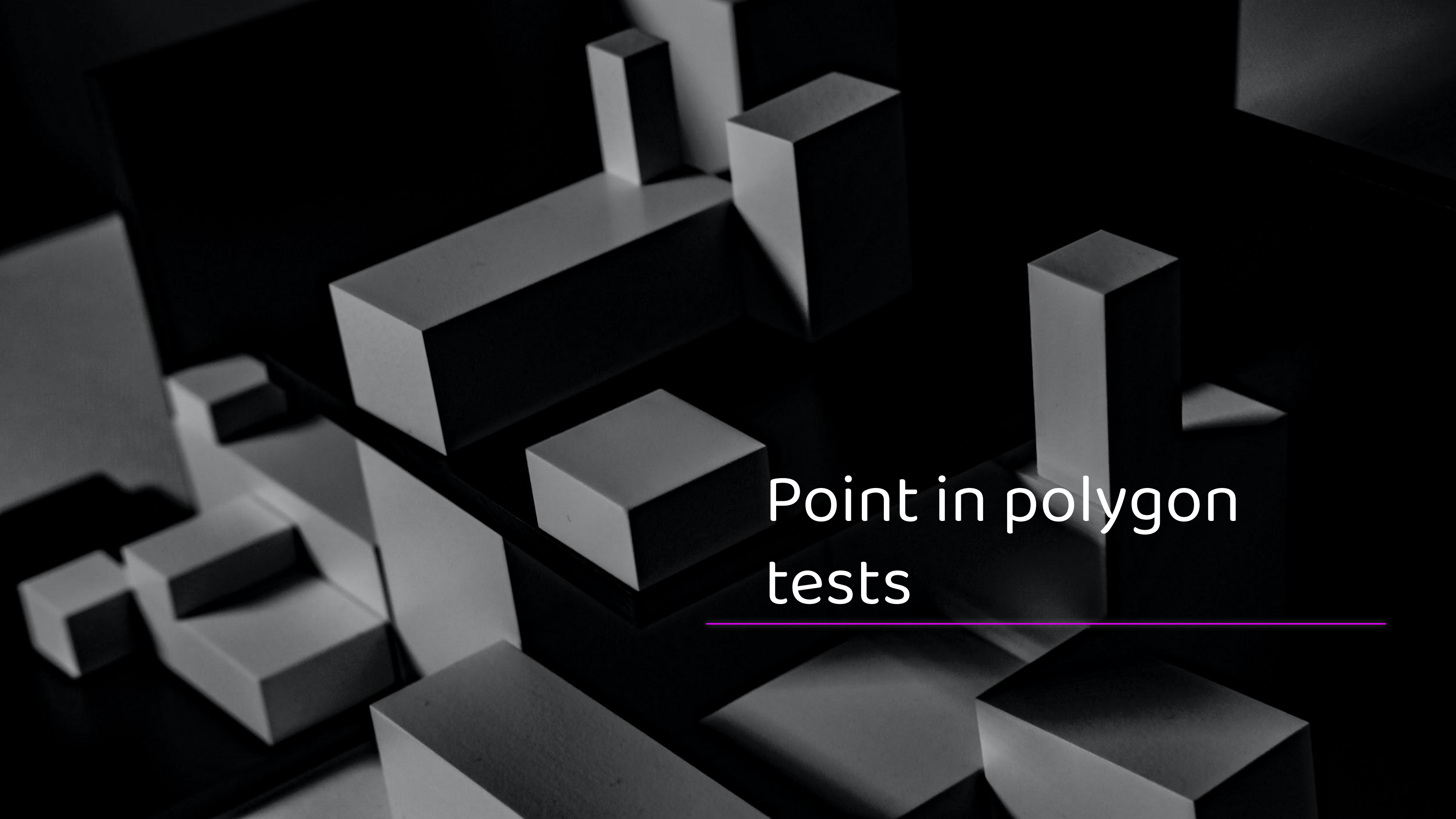
Square pseudo-angle

```
def square_pseudo_angle(v):
    if v[1] > 0:
        if v[0] > 0:
            a = v[1]/v[0]
            if a ⩾ 1:
                return v[1]+1-(1/a)
            else:
                return a
        else:
            return 4 - square_pseudo_angle([-v[0], v[1]])
    else:
        return 8 - square_pseudo_angle([v[0], -v[1]])
```

1 minus cosine pseudo-angle

```
def pseudo_angle_dot(a,b):
    return 1 - (np.dot(a,b) / (np.linalg.norm(a)*np.linalg.norm(b)))
```

Point in polygon tests



Crossings test

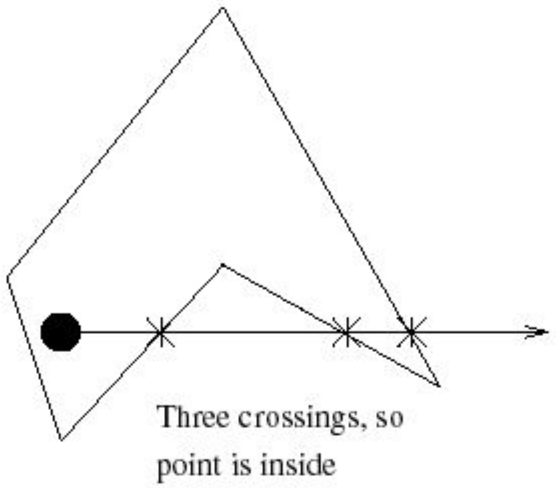


Figure 1 - Crossings Test

Winding Number test

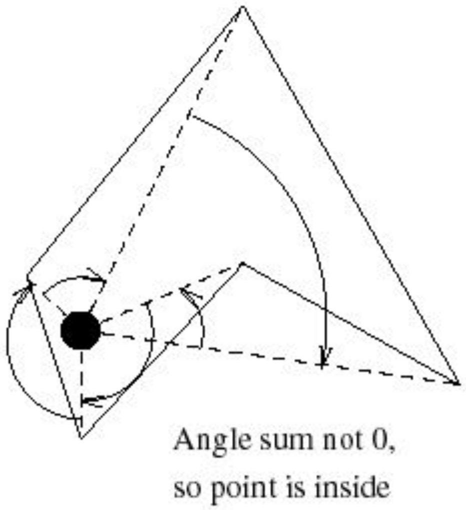
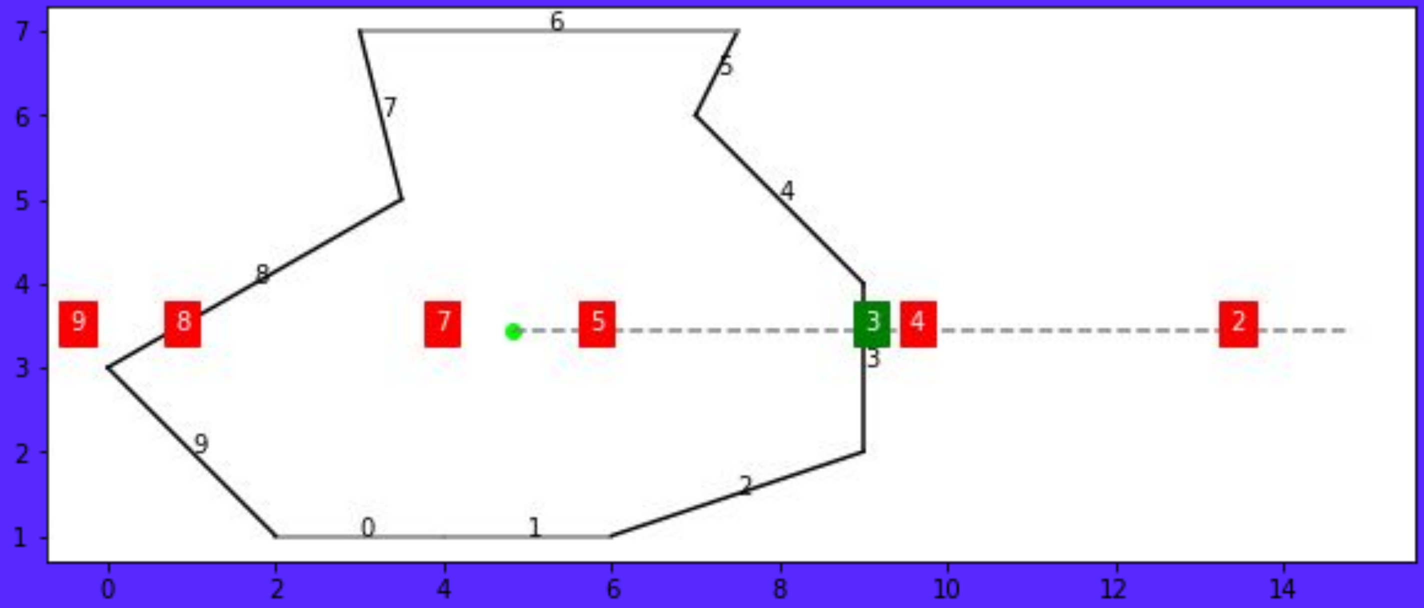


Figure 2 - Angle Summation Test

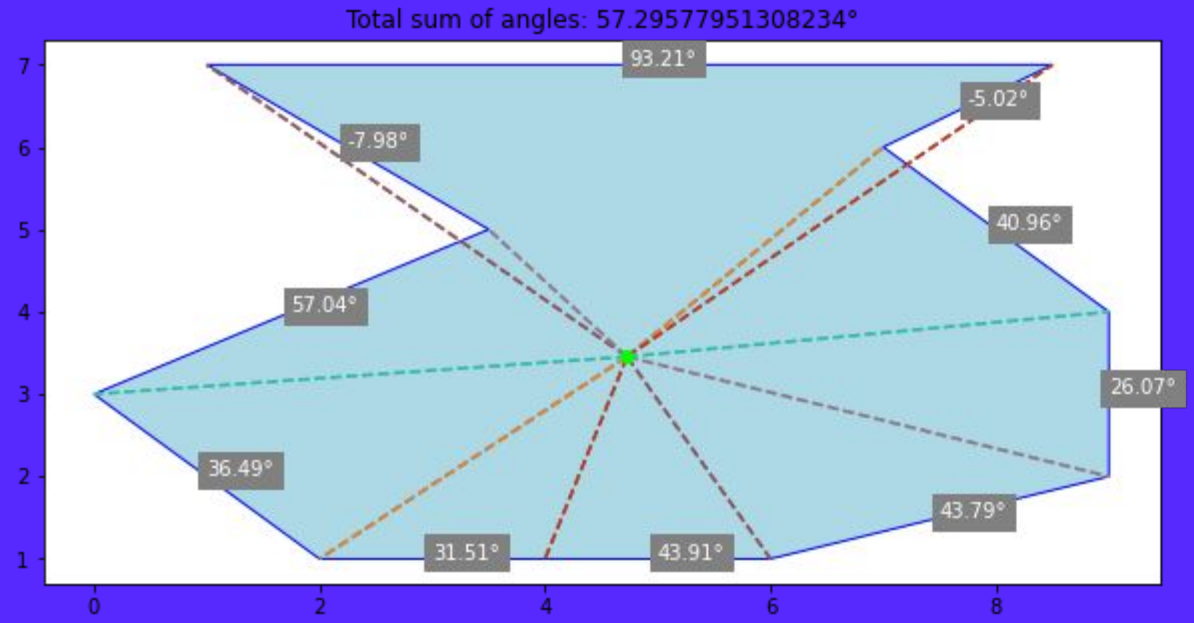
```python
# Receives a point P and a sequence of points p = [p_0, p_1, ... , p_n, p_(n+1)], that forms a closed polygon,
# where p_(n+1) == p_1. Returns -1 if outside, 0 if in frontier, 1 if inside
def point_in_polygon_intersection(P, p):
    n = len(p)-1
    N = 0 # Number of intersections
    [x0, y0] = [P[0], P[1]]
    Pn = np.add(P, [1,0]) # We will test the horizontal line that passes by P
    for i in range(0,n):
        xi = p[i,0]
        yi = p[i,1]
        xip1 = p[i+1,0]
        yip1 = p[i+1,1]

        if not math.isclose(yi, yip1): # Is not an horizontal line
            [x, y] = line_intersection(p[i], p[i+1], P, Pn) # Check the intersection between test line and one line of the poly
            if math.isclose(x, x0): # If the inter point is the same as the test point, itself lies on the polygon frontier
                return 0
            elif x > x0 and point_in_line([x,y], p[i],p[i+1]):
                N += 1
        elif point_in_line(P, p[i], p[i+1]):
            return 0
    odd = N % 2 == 1
    return 1 if odd else -1
```



9
8
7
5
3
4
2
0
1
2
3
4
5
6
7
8
9
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7

```python
def point_in_polygon_rotation(P, p):
    # Compute rotation index
    k = 0
    n = len(p)-1
    for i in range(0,n):
        Ppi = np.subtract(p[i], P)
        Ppip1 = np.subtract(p[i+1], P)
        or_angl = oriented_angle(Ppi, Ppip1)
        k += or_angl
    k *= 1/(2*math.pi)

    # Point is inside polygon if the rotation index is not zero
    return not math.isclose(k, 0, abs_tol=1e-5)
```

Important: When comparing floats, **never** use the equality sign, always use an epsilon comparison.


Total sum of angles: 57.29577951308234°
93.21°
-5.02°
-7.98°
40.96°
57.04°
26.07°
36.49°
43.79°
31.51°
43.91°
7
6
5
4
3
2
1
0
2
4
6
8