CS 370: Introduction to Computational Geometry

Advisor: Chandrajit Bajaj
Brady Zhou

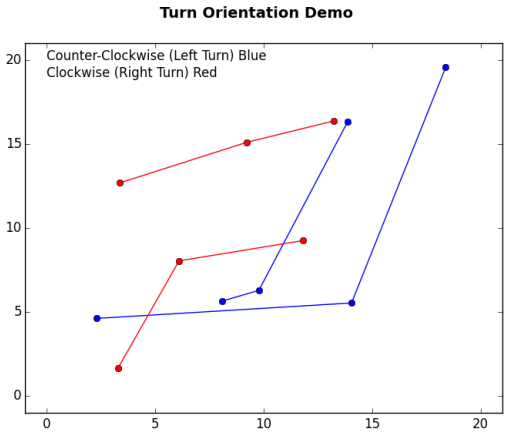
Abstract

The goal of this independent study course is to explore the fundamentals of computational geometry. In this semester long course, the topics covered will include turn orientation, convex hull, minkowski sum, triangulations, delanauy triangulations, voronoi diagrams, and more topics not decided yet. Professor Bajaj will advise by suggesting topics and papers to explore. The tools used will include **CGAL**, a computational geometry library in C++, and **matplotlib**, a Python library used for charting and visualizations of algorithms. The source code for this course can be found at github.com/bradyz/sandbox/tree/master/geometry.

0.1 Turn Orientation

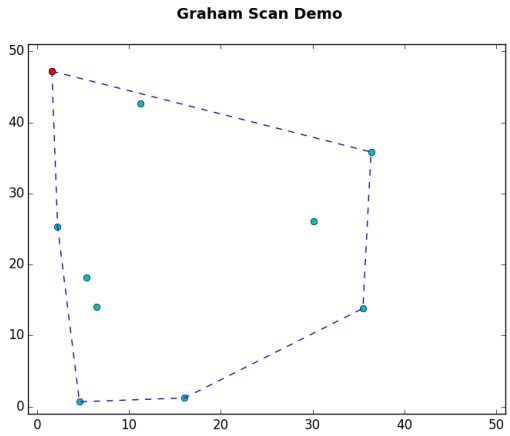
A common tool in the geometry toolbox is the idea of **turn orientation**, that is, whether a set of three points makes a clockwise (right), or counter-clockwise (left) turn. In a naive implementation of determining orientation of a turn, the angle of the line formed is calculated using some form of division and arcos. This can lead to loss of precision due to floating point numbers.

A better way to deal with line orientation is to take the determinant of the three points and if the determinant is larger than some ϵ, the points make a clockwise turn. If the determinant falls within $-\epsilon$ to ϵ, we say the points are colinear, and if the determinant is smaller than ϵ, the points make a counter-clockwise turn. The benefits of using this implementation is that accuracy is more robust as it does not use division.



0.2 Convex Hull

With just a one tool in our geometry toolbox, we can solve a fundamental problem of computational geometry - computing the **convex hull** of a set of points X. A convex hull is defined as the minimal subset of points that envelop X, or more formally, the intersection of all convex sets containing all X. There are well many well-known algorithms, but the one of interest is the Graham Scan, which was published in 1972.



The algorithm works by splitting the problem into two problems - finding the upper hull and then the lower hull. On a high level, to find the upper hull, we iterate over the points from left to right and have a result set R of points in upper hull. We keep appending points onto our result set and then check to see if the last points make a right turn. If they do not, we know the middle point is not a part of our final result set and we delete the middle point from the result set. We continue this until there are less than 3 points to make a turn, or the last three points do make a right turn. When the algorithm reaches the rightmost point, it terminates and returns the upper hull. The lower hull is computed using the same concept, but all the points in the lower hull will make left turns.

If we assume the list of points P is sorted by increasing x coordinates, the runtime algorithm is $O(n)$ where $n = |P|$, as the algorithm will iterate over the list once for each hull - upper and lower. Otherwise, if the list of points is not sorted, we will have to do so and that will be the dominating time complexity of $O(n\ log\ n)$. The psuedocode for the algorithm is as follows.

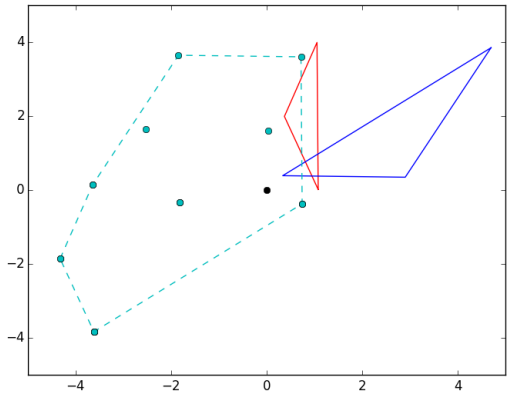
Algorithm UpperConvexHull(P)
Input: A set P of points to enclose.
Output: A set R, which is a subset of P.
Sort P by x-coordinate.
Add p_1, p_2 to R.
for $i \leftarrow 3$ **to** n
    Add p_i to R.
    while r_{n-2}, r_{n-1}, r_n do not make a right turn
        Remove r_{n-1} from R.
return R.

With this algorithm, we have to be careful in certain cases. In the case we have a triple of col-

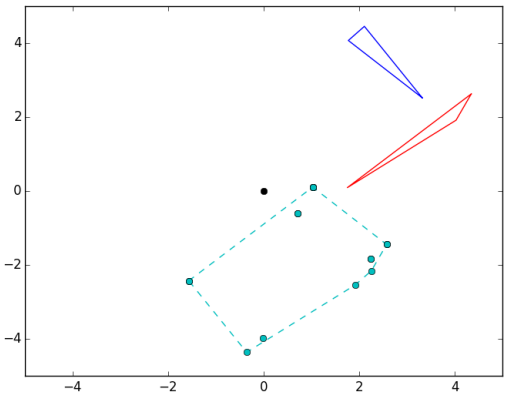
inear points on the hull, our algorithm should not include the middle point as the convex hull is defined as the minimal set of points. However, we can fix this relatively easy by ensuring our function that checks right turns returns false if the points are colinear, that is, if the determinant is zero or less.

0.3 Minkowski Sum

Once we have understood the concept of a convex hull, we can move onto to an application of the convex hull. Once we have the convex hull of two polygons, we can see if they collide. The **minkowski sum** of two vectors $S_1 \subset \mathbb{R}$ and $S_2 \subset \mathbb{R}$ is defined as $S_1 \oplus S_2 = \{p_x + q_x,\ p_y + q_y : p \in S_1,\ q \in S_2\}$. Similarly, the **minkowski difference** is defined as $S_1 \ominus S_2 = \{p_x - q_x,\ p_y - q_y : p \in S_1,\ q \in S_2\}$. Two polygons intersect in $\mathbb{R}^2$ if and only if $(0,\ 0) \in S_1 \ominus S_2$. With Minkowski differences, we can extend this collision detection to an arbitrary amount of dimensions, checking to see if the zero vector lies within the Minkowski difference.



Computing the Minkowski sum of two sets S_1, S_2 has a naive implementation of $O(n^2) + O(n\ log\ n)$, the addition of every point with another, followed by the calculating of the convex hull, but we can do better. If we consider an edge of $S_1 \oplus S_2$, we say the edge is **extreme**, that is, it has to be generated by points of S_1, S_2 that are extreme in this same direction. This way, each edge in S_1, S_2 is **charged** at most once.



The major application of Minkowski sum is in collision detecting. For instance, in robot motion planning, the robot and obstacles are represented by convex set of points and we can use the Minkowski difference to plan the motion of the robot.

TODO: Minkowskisum algorithm

TODO: Proof of runtime $O(n + m)$.

0.4 Triangulations

blah blah