

Computational Geometry

Andre Kessler *

December 4, 2009

Computational geometry is a topic that comes up every so often on USACO competitions, and thus is relatively useful to know. Since storing equations of lines and working with them is rather annoying, we'll use vectors (most often, 2 or 3 dimensional) for our calculations.

1 Basic Vector Facts

Recall the basic facts about vectors:

$$\mathbf{v} = \vec{v} = \langle v_x, v_y, v_z \rangle = v_x \mathbf{i} + v_y \mathbf{j} + v_z \mathbf{k}$$

where \mathbf{i} , \mathbf{j} , \mathbf{k} are the unit vectors in the positive x , y , and z directions. We can also specify the vector in terms of magnitude and angle. We denote the magnitude of vector \mathbf{v} by $v = |\mathbf{v}|$ and this can be found to be equal $\sqrt{x^2 + y^2 + z^2}$ by the Pythagorean theorem. We can multiply a vector by a real number, or scalar. The resulting vector is $k\mathbf{v} = \langle kv_x, kv_y, kv_z \rangle$ and its magnitude is just $k|\mathbf{v}|$. The vector in the opposite direction of \mathbf{v} is just $-\mathbf{v}$. Addition or subtraction of vectors works componentwise:

$$\mathbf{u} + \mathbf{v} = \langle u_x + v_x, u_y + v_y, u_z + v_z \rangle$$

In general, there are two ways to multiply vectors, the first being the “dot product” and the second being the “cross product.” The dot product results in a scalar, and is written as

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = uv \cos \theta$$

The angle θ is the angle between \mathbf{u} and \mathbf{v} ($\theta \leq \pi$). This also means that if and only if two vectors are perpendicular (or one is $\vec{0}$), then their dot product is equal to 0. The second way to multiply vectors is the cross product, which results in a vector orthogonal to \mathbf{u} and \mathbf{v} with magnitude equal to the area of the quadrilateral with adjacent sides \mathbf{u} and \mathbf{v} , or $uv \sin \theta$. The easiest way to remember the cross product is to write the two vectors in a matrix:

$$\begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix}$$

The cross product is then simply the determinant of that matrix, or

$$\mathbf{u} \times \mathbf{v} = \langle (u_u v_z - u_z v_y), (u_x v_z - u_z v_x), (u_x v_y - u_y v_x) \rangle$$

The cross product is NOT commutative; in fact, $\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$.

*Geometric formulas section extensively stolen from Johnathan Wang's old computational geometry lecture.

2 Geometric Formulas

2.1 Area of Triangle

To calculate the area of a triangle $\triangle ABC$, pick a vertex (for example, A) and create a vector to the other two vertices (let $\mathbf{u} = \mathbf{b} - \mathbf{a}$, and $\mathbf{v} = \mathbf{c} - \mathbf{a}$). The area of the triangle then is one half the length of cross product: $K = \frac{1}{2}|\mathbf{u} \times \mathbf{v}|$. Alternatively, if we are given the side lengths a, b, c , we can use Heron's formula: $K = \sqrt{s(s-a)(s-b)(s-c)}$, where s is the semiperimeter of length $\frac{a+b+c}{2}$.

2.2 Check if two lines are parallel

Make two vectors that represent the two lines and take their cross product. If the magnitude of this is 0, the lines are parallel. Since we're using doubles, check if $\text{abs}(\text{cross}(\mathbf{u}, \mathbf{v})) < \text{EPS}$.

2.3 Distance from point to a line

With a bit of thought and trigonometry, one can prove that the distance from a point P to line AB is

$$d(P, AB) = \frac{|(\mathbf{P} - \mathbf{A}) \times (\mathbf{B} - \mathbf{A})|}{|\mathbf{B} - \mathbf{A}|}$$

2.4 Distance from point to a line segment

Check if the triangle $\triangle PAB$ is obtuse; if so, take the minimum of $d(A), d(B)$; otherwise, just use the distance from a point to a line formula.

2.5 Check if a point is on a line

A point is on a line if the distance from the point to the line is 0.

2.6 Check if a point is on a line segment

Same as for a line.

2.7 Check if points are on the same side of a line

Note that this only works for two dimensions. If we want to check if points C and D are on the same side of line AB , calculate the z component z of $(\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A})$, and z' of $(\mathbf{B} - \mathbf{A}) \times (\mathbf{D} - \mathbf{A})$. If $zz' > 0$, then C and D are on the same side of line AB .

2.8 Check if a point is inside a triangle

A triangle is bounded by three lines. We know that the average of the vertices of the triangle is inside the triangle. So we check if this average point and the point in question are on the same side of each of the three sides of the triangle. If they are, the point we are concerned with is inside the triangle. If you like, you can pick some other point that is inside the triangle aside from the average point as well.

2.9 Check if a point is inside a convex polygon

You can do the same thing you did with the triangle, but now with n sides.

2.10 Check if four (or more) points are coplanar

To determine if a collection of points are coplanar, we take three points, A, B , and C . If for some other point D , $(\mathbf{B} - \mathbf{A})(\mathbf{C} - \mathbf{A})(\mathbf{D} - \mathbf{A}) = 0$, then the collection of points resides in the same plane.

2.11 Line Intersection

In 2D, two lines intersect if they are not parallel. In 3D, two lines intersect if they are not parallel and the four endpoints of the lines are coplanar (or just the lines are coplanar).

2.12 Line Segment Intersection

In 2D, two line segments AB and CD intersect if and only if A and B are on opposite sides of line CD and C and D are on opposite sides of line AB .

2.13 Check convexity of 2-D polygon

To check the convexity of a 2-dimensional polygon, traverse the vertices in clockwise order. For every three vertices A, B, C , calculate the cross product $(\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A})$. If the z component of all of the cross products are positive, the polygon is convex.

2.14 Check if point is in non-convex polygon

To determine if a point is inside a non-convex polygon, make a ray in a random direction from the point and count how many times it intersects the polygon. If the ray intersects at a vertex or along an edge, pick a new direction. Otherwise, the point is inside the polygon if and only if the ray intersects the polygon an odd number of times. You can do this for convex polygons as well.

3 Coding

3.1 Debugging

Make sure to cover the special cases, especially ones that might cause your program to crash (such as division by zero). Computational geometry problems often have an irritating number of special cases. Also, remember to use epsilons in your calculations with doubles - never check if two doubles are equal. Instead, check if $(\text{abs}(A - B) < \text{EPS})$, with $\text{EPS} = 1\text{e-}9$.

3.2 C++ <complex>

There are multiple ways of representing the vector we will use in our geometric calculations. You could create a struct of pairs of doubles, or use C++ pairs, but then you need to code $\text{dist}(A, B)$, yourself, remember how to calculate angles, rotate points, remember the formula for the cross product, and so on. There's lots of room for making tiny errors here. But there's a way out (in C++, at least)! Complex numbers will make coding long computational geometry algorithms much nicer because you don't need to code any of the basic functions. A few useful things you can do with complex numbers in C++ are listed below.

```
#include <complex>

// To save some typing later
typedef complex <double> pt;

// Given x, y coordinates, make a point (x, y) -- really the complex number x + yi
pt A = pt (x, y);
```

```

// X coordinate, Y coordinate
double x = A.real (), y = A.imag ();

// Angle point A makes with x-axis [0, 2 * Pi), angle between A and B
double angle = arg (A), angle_A_B = arg (A - B);

// Point A reflected across the x-axis (conjugate of a)
pt a_reflect = conj (A);

// Point A rotated by angle theta
pt a_rotated = A * exp (pt (0, theta));

// Distance between two points A, B
double dist = abs (A - B);

// Centroid of triangle ABC
pt centroid = (A + B + C) / 3;

// Dot product of vector A and vector B
double dot (pt A, pt B) {
    return real (conj (A) * B);
}

// Signed magnitude of A cross B
double cross (pt A, pt B) {
    return imag (conj (A) * B);
}

// Area of a triangle ABC
double triarea (pt A, pt B, pt C) {
    return 0.5 * abs (cross (B - A, C - A));
}

// Are two triangles A1, B1, C1, and A2, B2, C2 similar?
// First line checks if they're similar with same orientation,
// second line checks if they're similar with a different orientation
bool similar (pt A1, pt B1, pt C1, pt A2, pt B2, pt C2) {
    return ( (B2 - A2) * (C1 - A1) == (B1 - A1) * (C2 - A2)
            || (B2 - A2) * (conj (C1) - conj (A1)) == (conj (B1) - conj (A1)) * (C2 - A2));
}

```

4 Convex Hull

Given a collection of points in the plane, we want to find the convex polygon with smallest area such that each point is contained within (or on the boundary of) the polygon. This polygon is known as the convex hull of the set of points. If each point is a fixed peg on a board, the problem is the same as finding the polygon formed when we stretch a rubber band around that set of points. We will describe the Graham Scan algorithm below, which has complexity $O(N \log N)$. We find the average of all the points as the center point that we are certain is inside the convex hull. Then, we set this point as the “origin” and find the angles all the potential vertices make to the positive x axis. This lends itself to using the `atan2` function or `arg (z)` if we’re using complex numbers. We sort the angles to get the points in counterclockwise order. Now we traverse the points one by one. For every new point, we check the previous two points and see if the angle from $P_i P_{i-1} P_{i-2}$ is concave, using cross products. If it isn’t concave, we delete point P_{i-1} and check until

it is concave. After we go through all the points, we have to check points P_n , P_0 , and P_1 , to make the polygon closed and still ensure concavity. (Pseudocode below taken from the USACO training pages)

```

# x(i), y(i) is the x,y position
#   of the i-th point
# zcrossprod(v1,v2) -> z component
#       of the vectors v1, v2
# if zcrossprod(v1,v2) < 0,
#   then v2 is "right" of v1
# since we add counter-clockwise
#   <0 -> angle > 180 deg
1  (midx, midy) = (0, 0)
2  For all points i
3    (midx, midy) = (midx, midy) +
      (x(i)/npoints, y(i)/npoints)
4  For all points i
5    angle(i) = atan2(y(i) - midy,
      x(i) - midx)
6    perm(i) = i

7  sort perm based on the angle() values
# i.e., angle(perm(0)) <=
      angle(perm(i)) for all i

# start making hull
8  hull(0) = perm(0)
9  hull(1) = perm(1)
10 hullpos = 2
11 for all points p, perm() order,
      except perm(npoints - 1)
12   while (hullpos > 1 and
      zcrossprod(hull(hullpos-2) -
13     hull(hullpos-1),
      hull(hullpos-1) - p) < 0)
14     hullpos = hullpos - 1
15     hull(hullpos) = p
16     hullpos = hullpos + 1

# add last point
17 p = perm(npoints - 1)
18 while (hullpos > 1 and
      zcrossprod(hull(hullpos-2) -
19     hull(hullpos-1),
      hull(hullpos-1) - p) < 0)
20   hullpos = hullpos - 1

21 hullstart = 0
22 do
23   flag = false
24   if (hullpos - hullstart >= 2 and
      zcrossprod(p -
25     hull(hullpos-1),

```

```

        hull(hullstart) - p) < 0)
26     p = hull(hullpos-1)
27     hullpos = hullpos - 1
28     flag = true
29     if (hullpos - hullstart >= 2 and
        zcrossprod(hull(hullstart) - p,
        hull(hullstart+1) -
        hull(hullstart)) < 0)
30         hullstart = hullstart + 1
31         flag = true
32     while flag
33         hull(hullpos) = p
34         hullpos = hullpos + 1
// Convex hull is now stored in hull [] from hull [hullstart]
// to hull [hullpos - 1]

```

One possibly nicer algorithm to code (although the one above is pretty easy) is Andrew’s algorithm. This algorithm splits the convex hull into two parts: the upper and lower hull. These usually meet at the ends, although it’s possible for them to be joined by a line segment. We sort the points by x-coordinate, breaking ties by taking the largest y-coordinate. Afterwards, points are added in order of x-coordinate. This will sometimes still make the hull concave instead of convex, so we need to delete the second to last, third to last, etc, points until a convex triangle is found. At first glance, it may appear that this algorithm (excluding the sort) is $O(N^2)$. However, excluding the sort, this algorithm is actually $O(N)$. Why?

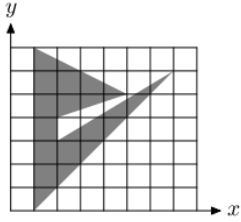
5 Line Sweep Algorithms

Farmer John’s N ($1 \leq N \leq 100,000$) cows are scattered across his pasture at unique (x, y) locations, grazing happily. However, Farmer John would like to know the location of the two closest grazing cows. Can you help him?

This is an example of a line sweep problem, where we consider a line “sweeping” across the plane, gathering the information that we need (although we can’t consider all locations, we must only consider special ones).

6 Problems

1. Given a set of N horizontal or vertical line segments in the plane, report all intersection points among the segments.
2. Given a set of N line segments in the plane, report all intersection points among the segments.
3. Farmer John’s cows eat grass in rectangular areas. Given a list of these rectangular areas, compute the total area of grass that cows have eaten from in $O(N \log(N))$. Note that parts of areas where the cows eat may intersect other cows’ grazing areas.
4. (IOI 98) Do the problem above, replacing “area” with ”perimeter.”
5. Given N ($1 \leq N \leq 100,000$) lines in the plane, find the x-coordinate of the leftmost intersection.
6. (ACM ICPC Nov06 NEERC) You are given lattice points in a $W \times H$ grid in an order that forms a closed polygon ($1 \leq W, H \leq 100$). Print out an ASCII representation of the polygon. Do this by printing a ‘.’ for each grid square filled in from 0% inclusive to 25% exclusive, a ‘+’ for 25% to 50%, an ‘o’ for 50% to 75%, a ‘\$’ for 75% to 100%, and a ‘#’ for exactly 100%. For example, given points (7,6), (1,0), (1,7), (5,5), (2,4), and (2,3), in that order, we get the picture



The correct output for these points would be:

```

.$+.....
.##$+...
.##$oo+..
.##$o...
.##o.....
.o.....
.o.....

```

7. (USACO DEC08) Given N ($1 \leq N \leq 250$) fence posts in the plane, what is the largest number of posts that Farmer John can select to form a convex fence?