GB21802 - Programming Challenges

Week 7 - Computational Geometry

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-06-9,12

Last updated June 8, 2017

Results for the Previous Week

Here are the results for last week:

Week 6: Mathematics
Deadline: 2017/6/9 23:59:59 (1 day, 14:08 hours from now)
Problems Solved -- 0P:29, 1P:9, 2P:7, 3P:8, 4P:1, 8P:1,

#	Name	Sol/Sub/Total	My Status
1	How Many Trees?	19/19/55	
2	Dice Throwing	20/21/55	
3	Self-describing Sequence	5/7/55	
4	Triangle Counting	5/5/55	
5	Summation of Four Primes	3/3/55	
6	Divisibility of Factors	5/6/55	
7	Marbles	1/1/55	
8	Winning Streak	1/1/55	

This Week: Computational Geometry

Computational Geometry problems involve answering questions about lines, points and angles;

- What is the area of the smallest poligon that covers points S_1, S_2, S_3?
- If we have N rectangles, $x_1, y_1, w_1, h_1; \ldots; x_N, y_N, w_N, h_N$, what is the smallest length of lines that connect all of them?
- How many triangles fit in this area?
- How many lines do you need to divide a polygon, so this set of points are in separate regions?

However, it is very easy to take WE (wrong answer) in geometric problems.

Geometry problems are easy for mistakes

It is very easy to receive a WE in a geometry problem.

Problem 1 – Special Cases

Many special cases: Two points in the same place; Collinear points; Vertical lines; Parallel Lines; Intersection at extremes; etc, etc.

Problem 2 – Precision Errors

Computational Geometry functions involve many multiplications and divisions. Floating point errors propagate very easily and can affect your final result.

You need to deal with these problems carefully!

Geometry problems are easy for mistakes – solution

Solving Special Cases

- Learn the common special cases for each geometric structure;
- Think about the possible special cases before programming;
- Prepare basic functions that deal with special cases;

Solving Precision Errors

- Whenever possible, convert all values to integers
- When it is not possible, use an EPSILON constant to make comparisons:

```
if (float.1 == float.2) then                 // Don't do this
if (fabs(float.1 - float.2) < EPS) then // Do this!
```

In both cases, the solution for not getting WE in geometry problems is Build a good library of basic cases!

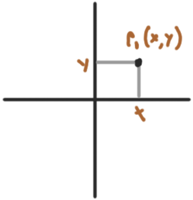
Class Outline

- Implementation of basic Geometric Functions
- Implementation of circles and triangles
- Algorithms on polygons

And of course some examples.

Basic Library – Points 1

You will use these functions many times when solving geometric problems. Make sure to write (and test!) them carefully.



Point Representation

```
struct point_i { int x, y;  // Using int coordinates.
  point_i() { x = y = 0; }
  point_i(int _x, int _y) : x(_x), y(_y) {}};

struct point { double x, y; // Using floats
  point() { x = y = 0.0;}
  point(double _x, double _y) : x(_x), y(_y) {}};
```

Basic Library – Points 2

Comparing (and sorting) points using overloaded operators:

Point Comparison

```
struct point { double x, y;
   point() { x = y = 0.0;
   point(double _x, double _y) : x(_x), y(_y) {}

   // Sorting by coordinate  -- sorting by angle also used
   bool operator < (point other) const {
      if (fabs(x - other.x) > EPS)
         return x < other.x;
      return y < other.y; }

   // Equality testing -- Note the use of EPS
   bool operator == (point other) const {
      return (fabs(x - other.x) < EPS &&
              (fabs(y - other.y) < EPS)); }
   }
```

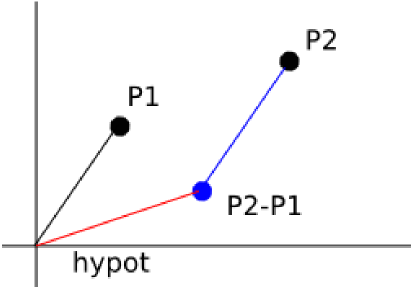
Basic Library – Points 3

Most common distance measure: Euclidean distance. Sometimes Manhattan distance (Taxicab distance) is also used.

```
#define hypot(dx,dy) sqrt(dx*dx + dy*dy)

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y); }

double taxicab(point p1, point p2) {
  return fabs(p1.x - p2.x) + fabs(p1.y - p2.y); }
```

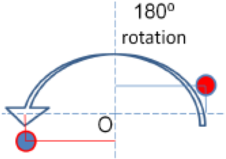


Basic Library – Points 4

Rotating a point around the origin

```
#define PI          3.14159265358979323846 // Pi const
double PI = 2 * acos(0.0)                  // Better Pi
#define DEG_to_RAD(X) (X*PI)/180.0

// theta is in degrees
point rotate(point p, double theta) {
   double rad = DEG_to_RAD(theta);
   return point(p.x * cos(rad) - p.y * sin(rad),
                p.x * sin(rad) + p.y * cos(rad));}
```



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Quiz: What do you do if you want to rotate a point around x_0, y_0?

Basic Library – Lines 1

There are many ways to specify a line:

- $ax + by + c = 0$ – useful for most cases.
- $y = mx + c$ – useful for angle manipulation, but special cases
- x_0, y_0, x_1, y_1 – two points, not very useful for programming.

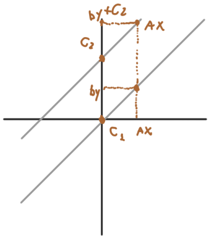
Point to Line

```
struct line { double a,b,c; };

void pointsToLine(point p1, point p2, line &l) {
   if (fabs(p1.x - p2.x) < EPS {
      l.a = 1.0; l.b = 0.0; l.c = -p1.x; }
   else {
      l.a = -(double) (p1.y - p2.y)/(p1.x - p2.x);
      l.b = 1.0; l.c = -(double) (l.a*p1.x) - p1.y;}
}
```

Basic Library – Line 2

- Two lines are parallel if their coefficients (a, b) are the same;
- Two lines are identical if all coefficients (a, b, c) are the same;
- Remember that we force b to be 0 or 1;



Parallel and identical lines

```
bool areParallel(line l1, line l2) {
   return (fabs(l1.a-l2.a) < EPS) &&
          (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {
   return areParallel(l1,l2) &&
          (fabs(l1.c - l2.c) < EPS); }
```

Basic Library – Line 3

The **intersection** point x_I, y_I is where two lines meet. We can find this point by solving the following system of linear equations:

$$a_1x + b_1y + c_1 = 0$$
$$a_2x + b_2y + c_2 = 0$$

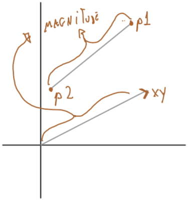
Computing the intersection

```
bool areIntersect(line l1, line l2, point &p) {
   if (areParallel(l1,l2)) return False;

   p.x = (l2.b * l1.c - l1.b * l2.c) /
         (l2.a * l1.b - l1.a * l2.b);
   if (fabs(l1.b) > EPS) // Testing for vertical case
      p.y = -(l1.a * p.x + l1.c);
   else
      p.y = -(l2.a * p.x + l2.c);
   return true; }}
```

Basic Library – Vectors

- A Vector indicates direction and length;
- Represented as a x, y point in relation to the origin;
- Operations: Scale, Translation, Addition, Product;



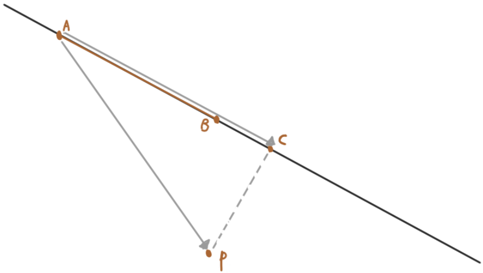

```
struct vec { double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {
    return vec(b.x - a.x, b.y - a.y); }
vec scale(vec v, double s) {
    return vec(v.x * s, v.y * s); }
point translate(point p, vec v) {
    return point(p.x + v.x , p.y + v.y); }
```

Distance between point and line

Given a point p and a line l, the distance between the point and the line is the distance between p and the c, the closest point in l to p.

We can calculate the position of c by taking the projection of $\bar{ac}$ into l (a, b are points in l).



Distance between point and line

```
double dot(vec a, vec b) {
   return (a.x * b.x + a.y * b.y); }
double norm_sq(vec v) {
   return v.x * v.x + v.y * v.y; }

// Calculates distance of p from line, given
// a,b different points in the line.
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
  // translate a to c
  return dist(p, c); }
```

Distance between point and segment

If we have a segment *ab* instead of a line, the procedure to calculate the distance is similar, but we need to test if the intersection point falls in the segment.

```
double distToLineSegment(point p, point a,
                         point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);

  if (u < 0.0) { c = point(a.x, a.y); // closer to a
                 return dist(p, a); }
  if (u > 1.0) { c = point(b.x, b.y); // closer to b
                 return dist(p, b); }

  return distToLine(p, a, b, c); }
```

Angles between segments

angle between two segments ao and ob

```
#import <cmath>

double angle(point a, point o, point b) { // in radians
vec oa = toVector(o, a), ob = toVector(o, b);
return acos(dot(oa, ob)/sqrt(norm_sq(oa)*norm_sq(ob)));}
```

Left/Right test: We can calculate the position of point p in relation to a line l using the cross product.

Take q, r points in l. Magnitude of the cross product pq x pr being positive/zero/negative means that $p \rightarrow q \rightarrow r$ is a left turn/collinear/right turn.

```
double cross(vec a, vec b) {
  return a.x * b.y - a.y * b.x; }
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }
collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
```

Problem Example: UVA – Intersection

Summary

You are given the following:

- Two points that define a segment:
  - $x_{p1}, y_{p1}, x_{p2}, y_{p2}$
- Four points that define a rectangle:
  - $x_{r1}, y_{r1}, x_{r2}, y_{r2}, x_{r3}, y_{r3}, x_{r4}, y_{r4}$

Calculate if the segment **intersects** the rectangle or not.

What are the steps necessary to calculate this?

Problem Example: UVA – Intersection

Summary

You are given the following:

- Two points that define a segment:
  - $x_{p1}, y_{p1}, x_{p2}, y_{p2}$
- Four points that define a rectangle:
  - $x_{r1}, y_{r1}, x_{r2}, y_{r2}, x_{r3}, y_{r3}, x_{r4}, y_{r4}$

Calculate if the segment **intersects** the rectangle or not.

What are the steps necessary to calculate this?

- Test if p_1 or p_2 are inside the rectangle (how?)
- Test if the segment intersects any side of the rectangle (how?)

Problem Example: UVA – Waterfalls

Summary

- A waterfall falls from the **source** straight down to $y = 0$.
- Barriers are diagonal segments represented by two points x_1, x_2
- What are the x position of the sources when they reach the ground?

What are the steps necessary to calculate this?

Problem Example: UVA – Waterfalls

Summary

- A waterfall falls from the **source** straight down to $y = 0$.
- Barriers are diagonal segments represented by two points x_1, x_2
- What are the x position of the sources when they reach the ground?

What are the steps necessary to calculate this?

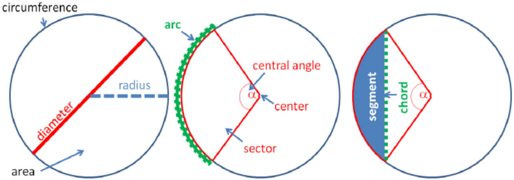
- A watersource is a vertical line.
- Each segment changes the position of the water source
- Scan the movement from top to bottom

Circles

- A circle is defined by its center (a, b) an its radius r
- The circle contains all points such (x, y) such as $(x-a)^2 + (y-b)^2 \leq r^2$

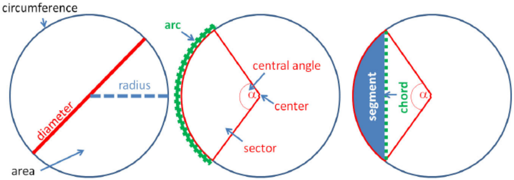
```
int insideCircle(point_i p, point_i c, int r) {
   int dx = p.x-c.x, dy = p.y-c.y;
   int Euc = dx*dx + dy*dy, rSq = r*r;
   return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
   // 0 - inside, 1 - border, 2- outside
}
```

Circles (2)



- If you are not given π, use $pi = 2*\text{acos}(0.0)$;
- Diameter: $D = 2r$; Perimeter/Circumference: $C = 2\pi r$; Area: $A = \pi r^2$;
- To calculat the Arc of an angle α (in Degrees), $\frac{\alpha}{360} * C$;

Circles (3)



- A chord of a circle is a segment composed of two points in the circle's border. A circle with radius r and angle α degrees has a chord of length sqrt($2r^2(1 - \cos\alpha)$)
- A Sector is the area of the circle that is enclosed by two radius and and arc between them. Area is: $\frac{\alpha}{360}A$
- A Segment is the region enclosed by a chord and an arc.

Problem Example: Area

Summary

Given 4 circles, determine the proportion of points that fall in all four circles.

Triangle Basics

Any 2 dimensional polygon can be expressed as a combination of triangles. So triangles are important constructs in computational geometry.

Common Characteristics

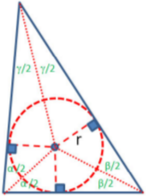
- Triangle Inequality: Sides a, b, c obey $a + b > c$
- Triangle Area: Be b one side of the triangle and h its height, $A = 0.5bh$
- Perimeter: $p = a + b + c$
- Semiperimeter: $s = 0.5p$

Heron's Formula

We can calculate the area of a triangle based on its sides:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

Incircle Triangle



Radius of the Incircle: $r = \text{area}(\Delta)/s$

```
def radiusInCircle(p1,p2,p3):
    ab, bc, cd = dist(p1,p2),dist(p2,p3),
                          dist(p3,p1)
    A = area(ab,bc,ca) % Heron's formula
    P = ab+bc+ca
    return A/(0.5*P)
```

Finding the center point of the Incircle

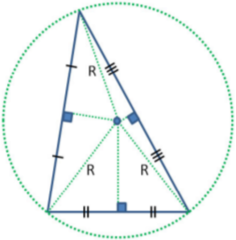
- Check that the three points are not colinear;
- Find the bisection *AP* of the *AB*-*AC* angle;
  - Calculate the point *P* in *BC* that bisects *A*
  - The proportion of *BP* is $(AB/AC)/(1 + AB/AC)$
- Find the bisection BP' of the *BA*-*BC* angle;
- Fint the intersection of AP-BP'

Incircle Triangle

Calculating the Center (Code)

```
int inCircle(point p1, point p2, point p3,
                point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0; // colinear points;
  line l1, l2; // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3),
                          ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);
  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3),
                   ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);
  areIntersect(l1, l2, ctr);
  return 1; }
```

Excircle Triangle



Radius of the excircle

A triangle with sides a, b, c and area A has an excircle with radius: $R = abc/4A$.

The center of the excircle is the intersection of the *perpendicular bisectors*.

Trigonometry

- Law of Cosines:
  $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$
  $\gamma = \text{acos}((a^2 + b^2 - c^2/2ab)$
- Law of Sines: (R is the radius of the excircle):
  $a/\sin(\alpha) = b/\sin(\beta) = c/\sin(\gamma) = R$

Example: UVA 11909 - Soya milk

Problem Description

Given the dimensions of a milk box and its inclination, calculate the amount of milk left in the box.

Example: UVA 10577 - Bounding Box

Given three vertices of a regular polygon, calculate the minimal square necessary to cover the polygon.

Hint: You don't actually need to calculate any polygons

Polygons

Definition

A polygon is a plane figure bounded by a finite sequence of line segments.

Polygon Representation

- In general we want to sort the points in CW or CCW order
- Adding the first point at the end of the array helps avoid special cases;

```
// 6 points, entered in counter clockwise order;
vector<point> P;
P.push_back(point(1, 1)); // P0
P.push_back(point(3, 3)); // P1
P.push_back(point(9, 1)); // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7)); // P4
P.push_back(point(1, 7)); // P5
P.push_back(P[0]); // important: loop back
```

Polygon Algorithms

Perimeter of a Poligon – sum of distances

```
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)
     // remember: P[0] = P[P.size()-1]
     result += dist(P[i], P[i+1]);
  return result; }
```

Area of a Poligon – half the determinant of the XY matrix

```
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1); }
  return fabs(result) / 2.0; }
```

Polygon – Concave and Convex check

Convex Polygons

Has NO line segment with ends inside itself that intersects its edges.

Another definition is that all inside angles “turn” the same way.

Testing for a convex polygon

```
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false; // Not a polygon
  bool isLeft = ccw(P[0], P[1], P[2]); //described earlier
  for (int i = 1; i < sz-1; i++)
    if (ccw(P[i],P[i+1],P[(i+2)==sz? 1 : i+2])!=isLeft)
      return false; // works for both left and right
      // different sign -> this polygon is concave
  return true; }
```

Polygon – Testing Inside or outside

There are many ways to test if a point P is in a polygon.

- Winding Algorithm: Sum the angles of all angles APB (A, B) are points in the polygon. If the sum is 2π. Point is in polygon.
- Ray Casting Algorithm: Draw an segment from P to infinity, and count the number of polygon edges crossed. Odds: Inside. Even: Outside.

Winding Algorithm Code

```
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]); //left turn/ccw
      else sum -= angle(P[i], pt, P[i+1]); } //right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }
```

Polygon – Cutting

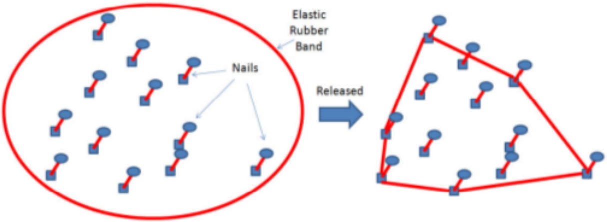
To cut *P* along a line *AB*, we separate the points in *P* to the left and right of the line.

```
point lineIntersectSeg(point p, point q, point A, point B) {
  double a=B.y-A.y; double b=A.x-B.x; double c=B.x*A.y-A.x*B.y;
  double u=fabs(a*p.x+b*p.y+c); double v=fabs(a*q.x+b*q.y+c);
  return point((p.x*v + q.x*u)/(u+v),
               (p.y*v + q.y*u)/(u+v)); }

vector<point> cutPolygon(point a, point b, const vector<point> &Q){
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1)
      left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS)
      P.push_back(Q[i]); //Q[i] is on the left of ab
    if (left1*left2 < -EPS) //edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b)); }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front()); // make P's first point = P's last point
  return P; }
```

Polygon – Convex Hull

Given a set of points S, the convex hull is the polygon P composed of a subset of S so that every point of S is either part of P, or inside it.



The main algorithm for calculating the convex hull is *Graham's Scan*.

It's idea is to test each point angle order, to see if the point belongs to the hull.

Polygon – Graham's Scan (1)

```
point pivot(0, 0);

bool angleCmp(point a, point b) { // angle-sorting
  if (collinear(pivot, a, b)) // special case
    return dist(pivot, a) < dist(pivot, b);
  // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }

vector<point> CH(vector<point> P) {
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0]==P[n-1])) P.push_back(P[0]); // special case
    return P; }
  // first, find P0 = point with lowest Y and, if tied, righmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y ||
        (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];
  // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);
```

Polygon – Graham's Scan (2)

```
// third, the ccw tests
vector<point> S;
S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);
// initial S
i = 2;
// then, we check the rest
while (i < n) {
  // note: N must be >= 3 for this method to work
  j = (int)S.size()-1;
  if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);
  // left turn, accept
  else S.pop_back(); }
  // or pop the top of S until we have a left turn
return S; }
```

This Week's Problems

- Sunny Mountains
- Bright Lights
- Packing polygons
- Elevator
- Soya Milk
- Trash Removal
- The Sultan's Problem
- Board Wrapping