Computational Geometry

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2025

Table of Contents

1. Introduction
2. Cross Product
   - Segment-Segment Intersection
   - Polygon Area
   - Convex Hull
   - Intersection of Half-Planes
3. Example Problems

Computational Geometry

Introduction

Cross Product

Segment-Segment Intersection

Polygon Area

Convex Hull

Intersection of Half-Planes

Example Problems

Introduction

- Computational geometry is the most frustrating part of solving programming problems.
- Algorithms often involve an undesirable number of special cases.
- Or they will require the use of easy-to-get-wrong geometric primitives.
- But even when you handle these special cases, you can have precision issues.

Precision

- Keep things in integers as much as possible!
- Be cognisant of multiplications which may overflow or divisions that do anything.
- Try not to divide.
- Division is evil.

Precision

- Never use floats.
- Only use doubles when absolutely necessary.
- Even if you have decimals, if they are fixed precision, you can usually just multiply all the input and use integers instead.

Precision

- When comparing doubles, use an epsilon value so there is a tolerance for floating point error:

```
const double EPS = 1e-8;

bool zero(double x) {
  return fabs(x) <= EPS;
}
```

- The magnitude of the epsilon will differ depending on the problem, but usually anything between 10^{-6} and 10^{-9} is safe.

- Similar techniques should be used for $\leq$ and $\geq$.

Table of Contents

1. Introduction
2. Cross Product
   - Segment-Segment Intersection
   - Polygon Area
   - Convex Hull
   - Intersection of Half-Planes
3. Example Problems

Cross Product

- The most useful of the products.
- We mostly deal with the “2D” cross product, where we assume the third dimension is always zero.
- Recall that

$$\begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix} \times \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix} = \begin{pmatrix} y_A z_B - y_B z_A \\ z_A x_B - x_A z_B \\ x_A y_B - y_A x_B \end{pmatrix},$$

so in particular

$$\begin{pmatrix} x_A \\ y_A \\ 0 \end{pmatrix} \times \begin{pmatrix} x_B \\ y_B \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ x_A y_B - y_A x_B \end{pmatrix}.$$

CCW

- We can use the cross product to implement the CCW operation, which given three points A, B and C tells us if the path from A to C via B is a left turn, a right turn or straight, i.e. if the points are counterclockwise, clockwise, or neither.
  - $\text{ccw}(A, B, C)$ is left if $(B - A) \times (C - A) > 0$.
  - $\text{ccw}(A, B, C)$ is right if $(B - A) \times (C - A) < 0$.
  - $\text{ccw}(A, B, C)$ is straight if $(B - A) \times (C - A) = 0$.

Cross Product

- **Implementation**

```
const double EPS = 1e-8;
typedef pair<double, double> pt;
#define x first
#define y second

pt operator+(pt a, pt b) {
  return pt(a.x + b.x, a.y + b.y);
}

pt operator-(pt a, pt b) {
  return pt(a.x - b.x, a.y - b.y);
}

bool zero(double x) {
  return fabs(x) <= EPS;
}

double cross(pt a, pt b) {
  return a.x*b.y - a.y*b.x;
}

// true if left or straight
// sometimes useful to instead return an int
// -1, 0 or 1: the sign of the cross product
bool ccw(pt a, pt b, pt c) {
  return cross(b - a, c - a) >= -EPS;
}
```

Segment-Segment Intersection

- Given two segments AB and CD, do they intersect?

- Intersections can be proper or non-proper.

- Two segments have a proper intersection if there exists a single point which is strictly inside both segments.

Segment-Segment Intersection

- We can use the ccw operation from before to check if two segments intersect.
- If the two segments have a proper intersection, then the path ABC should be in a different direction to the path ABD.
- The same is true of CDA and CDB.

Segment-Segment Intersection

- It turns out that these two conditions are both sufficient and necessary.

- However, if we also want to detect non-proper intersections, we also have to very carefully consider the other cases.

Segment-Segment Intersection

- What happens when the points are all collinear?
  - We can check for every cross product evaluating to zero.
- What happens if the intersection doesn't have to be strictly inside both segments? This is the case where one of the segments just touches the other one.
- One of the cross products will evaluate to zero.

Segment-Segment Intersection

- **Implementation**

```
typedef pair<pt, pt> seg;

bool collinear(seg ab, seg cd) { // all four points collinear
  pt a = ab.first, b = ab.second, c = cd.first, d = cd.second;
  return zero(cross(b - a, c - a)) &&
         zero(cross(b - a, d - a));
}

inline double sq(double t) { return t * t; }

double dist(pt p, pt q) { return sqrt(sq(p.x - q.x) + sq(p.y - q.y)); }

bool intersect(seg ab, seg cd) {
  pt a = ab.first, b = ab.second, c = cd.first, d = cd.second;

  if (collinear(ab, cd)) {
    double maxDist = max({dist(a, b), dist(a, c), dist(a, d),
                          dist(b, c), dist(b, d), dist(c, d)});
    return maxDist < dist(a, b) + dist(c, d) + EPS;
  }

  // only finds proper intersections
  // for non-proper, have ccw return an int
  // then return whether both products of ccws are <= 0
  return ccw(a, b, c) != ccw(a, b, d) &&
         ccw(c, d, a) != ccw(c, d, b);
}
```

Polygon Area

- Given a simple (not self-intersecting) polygon, what is its area?
- We can calculate this easily in time linear to the number of vertices in the polygon by using the concept of signed area.
- Intuitively, we want to add together some overestimate of the area of the polygon, but then subtract away the parts that aren't in the polygon.

Polygon Area

- Two common methods:
  - The trapezoidal rule, moving along the vertices of the polygon and adding together the areas of these signed trapezoids.
  - For every pair of adjacent vertices on the polygon, calculate the signed area of the triangle formed between those two vertices and some distinguished vertex, using the cross product, then sum all of these.

Polygon Area

- Both of these methods can be implemented using only integers if the input consists of only integers.

- The area of a trapezium is $\frac{h}{2}(a+b)$, and the 2D cross product is double the signed area of the triangle formed by the two vectors.

Polygon Area

- **Implementation (trapezoidal rule)**

```
double area(vector<pt> pts) {
  double res = 0;
  int n = pts.size();
  for (int i = 0; i < n; i++) {
    //     (a        + b             ) * h/2 (/2 moved to the end)
    res += (pts[i].y + pts[(i+1)%n].y) * (pts[(i+1)%n].x - pts[i].x);
    // sometimes, h will be negative, which means we subtract area
  }
  return res/2.0;
}
```

Polygon Area

- **Implementation (cross product)**

```
double area(vector<pt> pts) {
  double res = 0;
  int n = pts.size();
  for (int i = 1; i < n-1; i++) {
    // i = 0 and i = n-1 are degenerate triangles, OK to omit
    // e.g. if i = 1 is ABC, and i = 2 is ACD, then i = 0 is AAB
    res += cross(pts[i] - pts[0], pts[i+1] - pts[0]);
  }
  return res/2.0;
}
```

Example Problem: Polygon Area

- **Problem statement** Given a simple polygon with n points and two points which define a line of some width w, what is the total area of the polygon, minus the area which intersects the line?
- **Input** A simple polygon with n vertices ($1 \leq n \leq 1,000,000$), two points A and B, and the width w, a single floating point number.
- **Output** A single number, the required area.

Example Problem: Polygon Area

- We already know how to compute the area of a simple polygon, but there doesn't seem to be a simple way to reduce this problem to just finding a polygon area.
- We can try to intersect the line, which is really just a really, really long rectangle, with our polygon, calculate the points that form the resulting polygons, and use our polygon area algorithm.
- However, this is very hard to implement, and prone to error.

Example Problem: Polygon Area

- Let's try to solve an easier problem in an easier way.
- If the rectangle was vertical, we could use a slight modification of our trapezoidal rule based algorithm.
- For every trapezium that we add the area of, we clip the sides by the rectangle if we intersect it.

Example Problem: Polygon Area

- Notice that rotating the entire input of our problem does not change the answer.
- So we can just rotate the input so that our line is vertical, and apply the algorithm we have for our special case to solve the entire problem.
    - To figure out the angle of rotation, we can use `atan2(y, x)`.
- Since rotation is a constant time operation, our entire algorithm is still linear.

Example Problem: Polygon Area

- **Implementation (rotation)**

```
pt operator-(pt p, pt q) {
  return (pt){p.x - q.x, p.y - q.y};
}

pt operator+(pt p, pt q) {
  return (pt){p.x + q.x, p.y + q.y};
}

pt rotate(pt p, double a) {
  pt res;
  res.x = p.x * cos(a) - p.y * sin(a);
  res.y = p.x * sin(a) + p.y * cos(a);
  return res;
}

pt rotate(pt p, double a, pt o) {
  return rotate(p - o, a) + o;
}

// this is pi/2 - atan2(a.y - b.y, a.x - b.x);
double theta = atan2(a.x - b.x, a.y - b.y);
a = rotate(a, theta, b);
for (int i = 0; i < n; i++) {
  pts[i] = rotate(pts[i], theta, b);
}
```

Example Problem: Polygon Area

- **Implementation (calculating the area)**

```
double res = 0;
for (int i = 0; i < n; i++) {
  pt p = pts[i];
  pt q = pts[(i+1)%n];

  int sign = 1;
  if (p.x > q.x) {
    sign *= -1;
    swap(p, q);
  }

  // vertical or almost vertical means no area
  if (fabs(q.x - p.x) < 1e-9) { continue; }

  double m = (q.y - p.y) / (q.x - p.x);
  double c = p.y - m*p.x;

  // the rectangle cuts out the bit from nx to mx
  double nx = max(p.x, min(q.x, a.x - w));
  double ny = m*nx + c;
  double mx = min(q.x, max(p.x, a.x + w));
  double my = m*mx + c;

  // find the areas of the at most two pieces that aren't cut out
  res += sign * abs(nx - p.x) * (ny + p.y) / 2;
  res += sign * abs(q.x - mx) * (my + q.y) / 2;
}
```

Convex Hull

- The convex hull of a set of points P is the smallest convex shape that contains all points in P.
- Intuitively, imagine the points as nails in a wooden board. The convex hull can be formed by taking a rubber band and stretching it around all nails so that every nail is inside it.

Convex Hull

- It is relatively simple to compute the convex hull of a set of points in $O(n \log n)$ time, given that we have our ccw operation.
- Classically, this is done with the Graham scan, which starts by sorting all points by their angle relative to some point which is known to be on the convex hull and walking through this angle sorted list using a stack.

Convex Hull

- Andrew's monotone chain algorithm is simpler, because it avoids the angle sort, and achieves the same time complexity.
- The monotone chain algorithm computes two half-hulls, the upper and the lower hull, which are combined to form the final result.

Convex Hull

- To compute the upper hull, we first sort all points by x-coordinate, breaking ties by y-coordinate. We then walk through the points in this order, maintaining a "candidate half hull" with a stack.

- When adding a point, we must maintain the invariant that the point we just added as well as the two most recent points on the stack is still convex, i.e. forms a right turn.

- We can compute the lower hull in the same way, after reversing the order of the points.

- Sticking these together after removing duplicates gives the full convex hull.

Convex Hull

Implementation

```
vector<pt> half_hull(vector<pt> pts) {
  vector<pt> res;
  for (int i = 0; i < pts.size(); i++) {
    // ccw means we have a left turn; we don't want that
    while (res.size() >= 2 &&
           ccw(res[res.size()-2], res[res.size()-1], pts[i])) {
      res.pop_back();
    }
    res.push_back(pts[i]);
  }
  return res;
}

vector<pt> convex_hull(vector<pt> pts) {
  sort(pts.begin(), pts.end());
  vector<pt> top = half_hull(pts);

  reverse(pts.begin(), pts.end());
  vector<pt> bottom = half_hull(pts);

  // remove repeated endpoints
  top.pop_back(); bottom.pop_back();
  vector<pt> res(top.begin(), top.end());
  res.insert(res.end(), bottom.begin(), bottom.end());
  return res;
}
```

Intersection of Half-Planes

- Given a set of half-planes in the form $ax + by + c \leq 0$, what is the shape defined by their intersection?
- If the intersection is bounded, the intersection will be a convex polygon (otherwise it will be a convex area).
- Usually, in these problems, the possible point set is restricted by a bounding box, so we can usually just add the four sides of this box to our set of half planes and ensure the intersection is bounded.

Intersection of Half-Planes

- It's clear that the vertices of the intersection will be defined by the intersections of the lines that bound the half-planes.

- Each of these $O(n^2)$ intersections will either lie in the final result, or be excluded by some other half-plane.

- We can just use a $O(n)$ check, looping over all other half-planes for each intersection, to see if our intersection is excluded by some other half-plane.

Intersection of Half-Planes

- The remaining intersection points will be exactly the vertices of the polygon required.

- Taking the convex hull of these points will return the polygon vertices in a sensible sorted order.

- This algorithm runs in $O(n^3)$ time, and can be optimised with a small insight to run in $O(n^2)$ time. There exists a difficult $O(n \log n)$ algorithm.

Intersection of Half-Planes

- **Implementation (primitives)**

```
typedef pair<double, double> pt;

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

struct half_plane {
  line l;
  bool neg; // is the inequality <= or >=
};

pt intersect(line f, line g) {
  double d = f.a*g.b - f.b*g.a;
  double y = (f.a*g.c - f.c*g.a)/(f.b*g.a - f.a*g.b);
  double x = (f.c*g.b - f.b*g.c)/(f.b*g.a - f.a*g.b);
  return pt(x, y);
}

bool in_half_plane(half_plane hp, pt q) {
  double eval = hp.l.a*q.x + hp.l.b*q.y + hp.l.c;
  if (hp.neg) { return eval <=  EPS; }
  else        { return eval >= -EPS; }
}
```

Intersection of Half-Planes

- **Implementation (algorithm)**

```
vector<pt> intersect_half_planes(vector<half_plane> half_planes) {
  int n = half_planes.size();
  vector<pt> pts;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      pt p = intersect(half_planes[i].l, half_planes[j].l);
      bool fail = false;
      for (int k = 0; k < n; k++) {
        if (!in_half_plane(half_planes[k], p)) { fail = true; }
      }
      if (!fail) { pts.push_back(p); }
    }
  }

  vector<pt> res = pts;
  if (pts.size() > 2) { pts = convex_hull(res); }
  return pts;
}
```

Table of Contents

Example Problem: Maximum Collinear Points

- **Problem statement** Given a point set P, what is the largest number of points that lie on a single line?

- **Input** A point set P with n vertices ($1 \leq n \leq 5,000$). All points will be distinct.

- **Output** A single integer, the largest number of points that are all collinear.

Example Problem: Maximum Collinear Points

- The first thing we can notice is that we can define any line with two points from the point set, so there are only $O(n^2)$ interesting lines that we need to look at.
- With this, we immediately have an $O(n^3)$ algorithm, by enumerating all interesting lines and check if any point lies on it using the cross product.
- However, this is too slow for n up to 5,000.
- Notice that if three points are collinear, then all three pairs of these points define the same line. While the equation of this line might not be calculated equal in each case, the gradient is the same each time.

Example Problem: Maximum Collinear Points

- We can enumerate all interesting lines passing through A in $O(n)$ time, and count the number of times each gradient occurs using a hash map, to achieve a total complexity of $O(n^2)$ after considering all points A.
- Note that this is similar to angle sorting all points with respect to A, but ignoring the quadrants to some extent.

Example Problem: Maximum Collinear Points

- **Implementation**

```
for (int i = 0; i < n; i++) {
  map<pair<int, int>, int> counts;
  for (int j = 0; j < n; j++) {
    if (i == j) { continue; }
    int rise = pts[j].y - pts[i].y;
    int run = pts[j].x - pts[i].x;
    if (run == 0) { counts[make_pair(1, 0)]++; }
    else {
      int g = gcd(abs(run), abs(rise));
      if (run < 0) { g *= -1; }
      counts[make_pair(rise/g, run/g)]++;
    }
  }

  for(auto cnt : counts) {
    res = max(res, cnt.second+1);
  }
}
```

Example Problem: Lawn

- **Problem statement** You are a crotchety old man, who loves the idea of kids playing on your lawn. However, wise to the dangers of the world, you know that you must keep watch on them lest anything happens to the angels which play on your property.

  You know that there are n $(1 \leq n \leq 100,000)$ kids playing on integer points on your lawn, which is a 2D plane. You decide to find a place on your porch, which covers the line $y = 0$, where you can keep a watchful eye on the kids.

  You want to find a point on your porch where the maximum distance to any one kid is minimised. This point need not have integer coordinates.

  Output the maximum distance.

Example Problem: Lawn

- Since we have to find some floating point value, our first guess is to apply binary search somehow, because we don't have any other way of searching for the answer.

- How would we apply a binary search?

- It's clear that if all points are within X distance of our chosen point on the porch, then all points are also within $X + \epsilon$ distance to our chosen point as well.

Example Problem: Lawn

- Furthermore, we don't actually care what the optimal point on the porch looks like or where it is, just that it exists.

- So we can actually directly binary search for the answer that we require.

- How do we check if there exists some point on the porch that is within some distance X of every point in our input?

Example Problem: Lawn

- The points that are within some distance X of a point P are exactly those contained within a circle with radius X with its centre at P.

- So to check if some point on the porch is within some distance X of all of our points, we can check that there exists a point on the porch that lies in the intersection of all circles of radius X that are centred on the points.

Example Problem: Lawn

- We can implement this in linear time with a routine that clips the line that represents the porch by the circles, giving us an $O(n \log T)$ algorithm (where T depends on the desired precision), which is fast enough.

- However, while the clipping algorithm isn't too bad to implement, we'd like something easier if possible.

Example Problem: Lawn

- If we were restricted to only integer points on the porch, we could iterate over all interesting points on the porch (there are a finite number of these) and take the one that has the smallest maximum distance to another point.

- However, even if we could only use integers, this approach would be too slow for coordinates of even reasonable magnitude.

Example Problem: Lawn

- How do we know there are only a finite number of interesting integer points?
- It can be seen that there exists some point on the porch that has all points to the right of it resulting in a higher maximum distance.
- In the same way, there exists some other point that has all the points to the left of it also resulting in a higher maximum distance.
- Two easy candidates for these points are the points that are just slightly to the right and just slightly to the left of the input point set.

Example Problem: Lawn

- If we then think about the point where we reach the optimal solution, we can see that this point must satisfy both of these conditions.

- Furthermore, not only is the maximum distance larger the further we move away from our maximum point, but it is strictly increasing as well.

Example Problem: Lawn

- We call functions such as these, which first decrease and then increase, *bitonic*.

- To search for maximum or minimum points on bitonic functions, we can use a modified binary search called a **ternary search**, where instead of dividing our search range into two pieces, we can divide it into three pieces and update our search range.

Example Problem: Lawn

- This gives us a simpler $O(n \log T)$ algorithm to solve this problem, which may converge faster depending on how we choose to split our ternary search range.
- Splitting the range into thirds, quartiles, by the golden ratio, or even taking the middle range to be of unit length works.

Example Problem: Lawn

- **Implementation**

```
typedef long double ld;

ld check(ld x) {
  ld ret = 0;
  for (int i = 0; i < n; i++) {
    ret = max(ret, distsqr(x, pts[i]));
  }
  return ret;
}

for (int it = 0; it < 70; it++) {
  ld mid1 = (hi + 2*lo)/3;
  ld mid2 = (2*hi + lo)/3;
  if (check(mid1) > check(mid2)) { lo = mid1; }
  else                           { hi = mid2; }
}
```

Example Problem: Union of Rectangle Area

- Given a set of *n* axis aligned rectangles, what is the area of their union?
- We can solve this using inclusion-exclusion in exponential time, but we can do better.

Example Problem: Union of Rectangle Area

- Let’s first consider the single dimensional case where we have a set of intervals on the x-axis and we want to find out the length of their union.

- We can solve this in $O(n \log n)$ with a similar algorithm to the one used in the “Stabbing” problem by considering a sorted list of the start points and end points of the intervals.

- We iterate over each start point and end point in order from left to right, keeping track of how many intervals are “open” at the current moment, adding to our total length whenever there are unclosed intervals.

Example Problem: Union of Rectangle Area

- If we take this idea, we can directly apply it to the 2D case to obtain an algorithm that runs in $O(n^2 \log n)$ time.

- We consider the horizontal start and end points of each rectangle, and process them in sorted order like with the 1D intervals.

Example Problem: Union of Rectangle Area

- At every stage, we maintain a set of “active intervals”, which correspond to the rectangles that overlap our x-position.
- To calculate the answer for this x-position, we just need to run the $O(n \log n)$ algorithm for the 1D case for every x-coordinate to immediately get an $O(Xn \log n)$ algorithm, where X is the maximum x-coordinate.
- We can easily speed this up to $O(n^2 \log n)$ by only considering the $O(n)$ “interesting” x-coordinates where our active set changes.

Example Problem: Union of Rectangle Area

- It turns out that we can still do better, and solve this problem in $O(n \log n)$.

- We are doing a lot of unnecessary computation to solve our 1D subproblem, because our active set changes very little on each event.

Example Problem: Union of Rectangle Area

- We can use a range tree to handle each subproblem in $O(\log X)$ time (per update), with $O(1)$ time queries.
- To do this, we simply make a node for each x-coordinate, and use lazy range updates to increment the number of rectangles "open" in a range when a rectangle starts, and decrementing when one ends.
- We can't use the delta trick like in Card Trick (week 5), because this isn't a normal sum tree, it's a sum of 1s for nodes that are nonzero, and 0s for the others.
- This uses $O(X)$ space and $O(\log X)$ time, but we can use **coordinate compression** to improve this to $O(n)$ space and $O(\log n)$ time.
- Coordinate compression involves only storing "interesting" positions, and some extra work to keep track of the fact that there are now varying (non-1) "distances" between nodes.