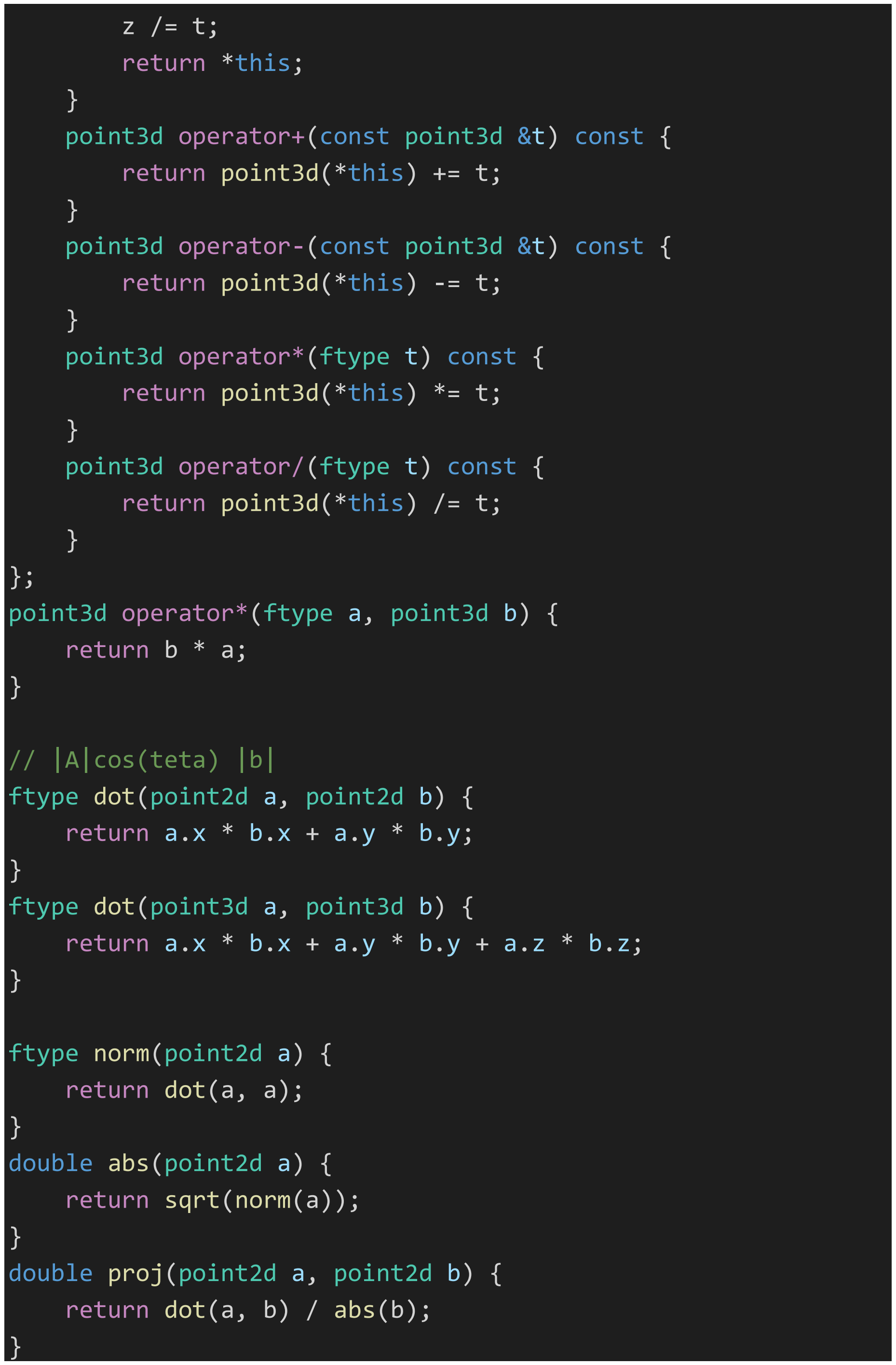
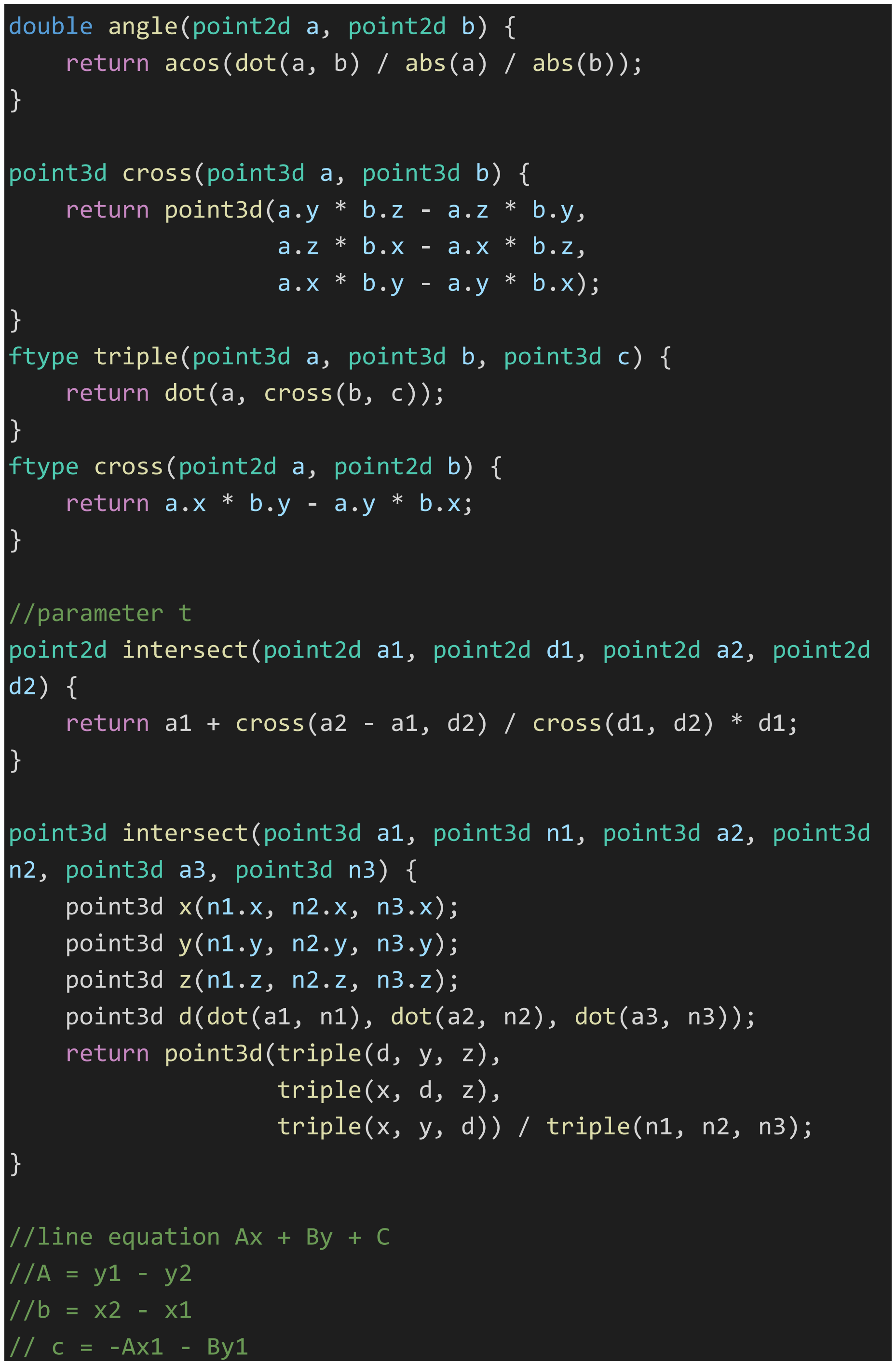
```cpp
#include "bits/stdc++.h"
using namespace std;

typedef pair<int, int> ii;

typedef double ftype;

struct point2d {
    ftype x, y;
    point2d() {}
    point2d(ftype x, ftype y): x(x), y(y) {}
    point2d& operator+=(const point2d &t) {
        x += t.x;
        y += t.y;
        return *this;
    }
    point2d& operator-=(const point2d &t) {
        x -= t.x;
        y -= t.y;
        return *this;
    }
    point2d& operator*=(ftype t) {
        x *= t;
        y *= t;
        return *this;
    }
    point2d& operator/=(ftype t) {
        x /= t;
        y /= t;
        return *this;
    }
    point2d operator+(const point2d &t) const {
        return point2d(*this) += t;
    }
    point2d operator-(const point2d &t) const {
        return point2d(*this) -= t;
    }
    point2d operator*(ftype t) const {
        return point2d(*this) *= t;
    }
    point2d operator/(ftype t) const {
        return point2d(*this) /= t;
    }
};

point2d operator*(ftype a, point2d b) {
    return b * a;
}

struct point3d {
    ftype x, y, z;
    point3d() {}
    point3d(ftype x, ftype y, ftype z): x(x), y(y), z(z) {}
    point3d& operator+=(const point3d &t) {
        x += t.x;
        y += t.y;
        z += t.z;
        return *this;
    }
    point3d& operator-=(const point3d &t) {
        x -= t.x;
        y -= t.y;
        z -= t.z;
        return *this;
    }
    point3d& operator*=(ftype t) {
        x *= t;
        y *= t;
        z *= t;
        return *this;
    }
    point3d& operator/=(ftype t) {
        x /= t;
        y /= t;
```



```
        z /= t;
        return *this;
    }
    point3d operator+(const point3d &t) const {
        return point3d(*this) += t;
    }
    point3d operator-(const point3d &t) const {
        return point3d(*this) -= t;
    }
    point3d operator*(ftype t) const {
        return point3d(*this) *= t;
    }
    point3d operator/(ftype t) const {
        return point3d(*this) /= t;
    }
};
point3d operator*(ftype a, point3d b) {
    return b * a;
}

// |A|cos(teta) |b|
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
ftype dot(point3d a, point3d b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

ftype norm(point2d a) {
    return dot(a, a);
}
double abs(point2d a) {
    return sqrt(norm(a));
}
double proj(point2d a, point2d b) {
    return dot(a, b) / abs(b);
}
```



```
double angle(point2d a, point2d b) {
    return acos(dot(a, b) / abs(a) / abs(b));
}

point3d cross(point3d a, point3d b) {
    return point3d(a.y * b.z - a.z * b.y,
                   a.z * b.x - a.x * b.z,
                   a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {
    return dot(a, cross(b, c));
}
ftype cross(point2d a, point2d b) {
    return a.x * b.y - a.y * b.x;
}

//parameter t
point2d intersect(point2d a1, point2d d1, point2d a2, point2d
d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
}

point3d intersect(point3d a1, point3d n1, point3d a2, point3d
n2, point3d a3, point3d n3) {
    point3d x(n1.x, n2.x, n3.x);
    point3d y(n1.y, n2.y, n3.y);
    point3d z(n1.z, n2.z, n3.z);
    point3d d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
    return point3d(triple(d, y, z),
                   triple(x, d, z),
                   triple(x, y, d)) / triple(n1, n2, n3);
}

//line equation Ax + By + C
//A = y1 - y2
//b = x2 - x1
// c = -Ax1 - By1
```

```cpp
//Real numbers divide by z = sqrt(a^2 + b^2)

//3d  p+vt

struct pt {
    double x, y;

    bool operator<(const pt& p) const
    {
        return x < p.x - EPS || (abs(x - p.x) < EPS && y < p.y
- EPS);
    }
};

struct line {
    double a, b, c;

    line() {}
    line(pt p, pt q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = -a * p.x - b * p.y;
        norm();
    }

    void norm()
    {
        double z = sqrt(a * a + b * b);
        if (abs(z) > EPS)
            a /= z, b /= z, c /= z;
    }

    double dist(pt p) const { return a * p.x + b * p.y + c; }
};
```

```cpp
//intersection point of lines
const double EPS = 1e-9;

double det(double a, double b, double c, double d) {
    return a*d - b*c;
}

bool intersect(line m, line n, pt & res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS)
        return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}

bool parallel(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS;
}

bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS
        && abs(det(m.a, m.c, n.a, n.c)) < EPS
        && abs(det(m.b, m.c, n.b, n.c)) < EPS;
}

//Check if 2 segments intersect

struct pt2 {
    long long x, y;
    pt2() {}
    pt2(long long _x, long long _y) : x(_x), y(_y) {}
    pt2 operator-(const pt2& p) const { return pt2(x - p.x, y
- p.y); }
    long long cross(const pt2& p) const { return x * p.y - y *
p.x; }
```

```
    long long cross(const pt2& a, const pt2& b) const { return
(a - *this).cross(b - *this); }
};

int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1;
}

bool inter1(long long a, long long b, long long c, long long
d) {
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d);
}

bool check_inter(const pt2& a, const pt2& b, const pt2& c,
const pt2& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y,
c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}


//interseciton of segments
inline bool betw(double l, double r, double x)
{
    return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}

inline bool intersect_1d(double a, double b, double c, double
d)
{
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d) + EPS;
}

bool intersect(pt a, pt b, pt c, pt d, pt& left, pt& right)
{
    if (!intersect_1d(a.x, b.x, c.x, d.x) ||
!intersect_1d(a.y, b.y, c.y, d.y))
        return false;
    line m(a, b);
    line n(c, d);
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS) {
        if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS)
            return false;
        if (b < a)
            swap(a, b);
        if (d < c)
            swap(c, d);
        left = max(a, c);
        right = min(b, d);
        return true;
    } else {
        left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;
        left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;
        return betw(a.x, b.x, left.x) && betw(a.y, b.y,
left.y) &&
               betw(c.x, d.x, left.x) && betw(c.y, d.y,
left.y);
    }
}

//for circle circle A= -2x2   B= -2y2   C=x2^2+y2^2 +r1^2 -
r2^2
void lineCircleIntersection(double r, double a, double b,
double c) {
  double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
  if (c*c > r*r*(a*a+b*b)+EPS)
      puts ("no points");
  else if (abs (c*c - r*r*(a*a+b*b)) < EPS) {
      puts ("1 point");
      cout << x0 << ' ' << y0 << '\n';
  }
  else {
      double d = r*r - c*c/(a*a+b*b);
      double mult = sqrt (d / (a*a+b*b));
      double ax, ay, bx, by;
      ax = x0 + b * mult;
      bx = x0 - b * mult;
      ay = y0 - a * mult;
      by = y0 + a * mult;
      puts ("2 points");
      cout << ax << ' ' << ay << '\n' << bx << ' ' << by <<
'\n';
  }
}


int signed_area_parallelogram(point2d p1, point2d p2,
point2d p3) {
    return cross(p2 - p1, p3 - p2);
}

double triangle_area(point2d p1, point2d p2, point2d p3) {
    return abs(signed_area_parallelogram(p1, p2, p3)) /
2.0;
}

bool clockwise(point2d p1, point2d p2, point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) < 0;
}

bool counter_clockwise(point2d p1, point2d p2, point2d p3)
{
```

```
    return signed_area_parallelogram(p1, p2, p3) > 0;
}

double area(const vector<point>& fig) {
    double res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        point p = i ? fig[i - 1] : fig.back();
        point q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
}

bool lexComp(const pt &l, const pt &r) {
    return l.x < r.x || (l.x == r.x && l.y < r.y);
}

int sgn(long long val) { return val > 0 ? 1 : (val == 0 ? 0
: -1); }

vector<pt> seq;
pt translation;
int n;

bool pointInTriangle(pt a, pt b, pt c, pt point) {
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) +
abs(point.cross(b, c)) + abs(point.cross(c, a));
    return s1 == s2;
}

void prepare(vector<pt> &points) {
    n = points.size();
    int pos = 0;
    for (int i = 1; i < n; i++) {
        if (lexComp(points[i], points[pos]))
            pos = i;
    }
    rotate(points.begin(), points.begin() + pos,
points.end());

    n--;
    seq.resize(n);
    for (int i = 0; i < n; i++)
        seq[i] = points[i + 1] - points[0];
    translation = points[0];
}

bool pointInConvexPolygon(pt point) {
    point = point - translation;
    if (seq[0].cross(point) != 1 &&
            sgn(seq[0].cross(point)) !=
sgn(seq[0].cross(seq[n - 1])))
        return false;
    if (seq[n - 1].cross(point) != 0 &&
            sgn(seq[n - 1].cross(point)) != sgn(seq[n -
1].cross(seq[0])))
        return false;

    if (seq[0].cross(point) == 0)
        return seq[0].sqrLen() >= point.sqrLen();

    int l = 0, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
            l = mid;
        else
            r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos], seq[pos + 1], pt(0,
0), point);
```

```
}


void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y == P[pos].y &&
P[i].x < P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
}

vector<pt> minkowski(vector<pt> P, vector<pt> Q){
    // the first vertex must be the lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while(i < P.size() - 2 || j < Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] - P[i]).cross(Q[j + 1] -
Q[j]);
        if(cross >= 0)
            ++i;
        if(cross <= 0)
            ++j;
    }
    return result;
}
```

```
int count_lattices(Fraction k, Fraction b, long long n) {
    auto fk = k.floor();
    auto fb = b.floor();
    auto cnt = 0LL;
    if (k >= 1 || b >= 1) {
        cnt += (fk * (n - 1) + 2 * fb) * n / 2;
        k -= fk;
        b -= fb;
    }
    auto t = k * n + b;
    auto ft = t.floor();
    if (ft >= 1) {
        cnt += count_lattices(1 / k, (t - t.floor()) / k,
t.floor());
    }
    return cnt;
}

struct cmp_x {
    bool operator()(const pt & a, const pt & b) const {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};

struct cmp_y {
    bool operator()(const pt & a, const pt & b) const {
        return a.y < b.y;
    }
};

int n;
vector<pt> a;

double mindist;
pair<int, int> best_pair;

void upd_ans(const pt & a, const pt & b) {
    double dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y -
b.y)*(a.y - b.y));
    if (dist < mindist) {
        mindist = dist;
        best_pair = {a.id, b.id};
    }
}
```