1 Geometry

1.1 Vectors/Points

```
#include "../../headers/headers.h"

const double PI = acos(-1);

struct vector2D
{
    double x, y;

    vector2D &operator+=(const vector2D &o)
    {
        this->x += o.x;
        this->y += o.y;
        return *this;
    }

    vector2D &operator-=(const vector2D &o)
    {
        this->x -= o.x;
        this->y -= o.y;
        return *this;
    }

    vector2D operator+(const vector2D &o)
    {
        return {x + o.x, y + o.y};
    }

    vector2D operator-(const vector2D &o)
    {
        return {x - o.x, y - o.y};
    }

    vector2D operator*(const double &o)
    {
        return {x * o, y * o};
    }

    bool operator==(const vector2D &o)
    {
        return x == o.x and y == o.y;
    }

    double norm2() { return x * x + y * y; }
    double norm() { return sqrt(norm2()); }
    double dot(const vector2D &o) { return x * o.x + y * o.y; }
    double cross(const vector2D &o) { return x * o.y - y * o.x; }
    double angle()
    {
        double angle = atan2(y, x);
        if (angle < 0)
            angle += 2 * PI;
        return angle;
    }

    vector2D Unit()
    {
        return {x / norm(), y / norm()};
    }
};

/* ========================================================== */
/* Cross Product -> orientation of vector2D with respect to ray */
/* ========================================================== */
// cross product (b - a) x (c - a)
ll cross(vector2D &a, vector2D &b, vector2D &c)
{
    ll dx0 = b.x - a.x, dy0 = b.y - a.y;
    ll dx1 = c.x - a.x, dy1 = c.y - a.y;
    return dx0 * dy1 - dx1 * dy0;
    // return (b - a).cross(c - a); // alternatively, using struct function
}

// calculates the cross product (b - a) x (c - a)
// and returns orientation:
// LEFT (1):      c is to the left of  ray (a -> b)
// RIGHT (-1):    c is to the right of ray (a -> b)
// COLLINEAR (0): c is collinear to    ray (a -> b)
// inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-points/
int orientation(vector2D &a, vector2D &b, vector2D &c)
{
    ll tmp = cross(a, b, c);
    return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
}

/* ========================================================= */
/* Check if a segment is below another segment (wrt a ray) */
/* ========================================================= */
// i.e: check if a segment is intersected by the ray first
// Assumptions:
// 1) for each segment:
//  p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or COLLINEAR) wrt ray
// 2) segments do not intersect each other
// 3) segments are not collinear to the ray
// 4) the ray intersects all segments
struct Segment
{
    vector2D p1, p2;
};
#define MAXN (int)1e6   //Example
Segment segments[MAXN]; // array of line segments
bool is_si_below_sj(int i, int j)
{ // custom comparator based on cross product
```

```cpp
    Segment &si = segments[i];
    Segment &sj = segments[j];
    return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0 : cross
        (sj.p1, si.p1, si.p2) > 0;
}
// this can be used to keep a set of segments ordered by order of
    intersection
// by the ray, for example, active segments during a SWEEP LINE
set<int, bool (*)(int, int)> active_segments(is_si_below_sj); // ordered
    set

/* ======================= */
/* Rectangle Intersection */
/* ======================= */
bool do_rectangles_intersect(vector2D &dl1, vector2D &ur1, vector2D &dl2
    , vector2D &ur2)
{
    return max(dl1.x, dl2.x) <= min(ur1.x, ur2.x) && max(dl1.y, dl2.y)
        <= min(ur1.y, ur2.y);
}

/* ========================= */
/* Line Segment Intersection */
/* ========================= */
// returns whether segments p1q1 and p2q2 intersect, inspired by:
// https://www.geeksforgeeks.org/check-if-two-given-line-segments-
    intersect/
bool do_segments_intersect(vector2D &p1, vector2D &q1, vector2D &p2,
    vector2D &q2)
{
    int o11 = orientation(p1, q1, p2);
    int o12 = orientation(p1, q1, q2);
    int o21 = orientation(p2, q2, p1);
    int o22 = orientation(p2, q2, q1);
    if (o11 != o12 and o21 != o22) // general case -> non-collinear
        intersection
        return true;
    if (o11 == o12 and o11 == 0)
    { // particular case -> segments are collinear
        vector2D dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
        vector2D ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
        vector2D dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
        vector2D ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
        return do_rectangles_intersect(dl1, ur1, dl2, ur2);
    }
    return false;
}

/* =================== */
/* Circle Intersection */
/* =================== */
struct Circle
{
    double x, y, r;
};
bool is_fully_outside(double r1, double r2, double d_sqr)
{
    double tmp = r1 + r2;
    return d_sqr > tmp * tmp;
}
bool is_fully_inside(double r1, double r2, double d_sqr)
{
    if (r1 > r2)
        return false;
    double tmp = r2 - r1;
    return d_sqr < tmp * tmp;
}
bool do_circles_intersect(Circle &c1, Circle &c2)
{
    double dx = c1.x - c2.x;
    double dy = c1.y - c2.y;
    double d_sqr = dx * dx + dy * dy;
    if (is_fully_inside(c1.r, c2.r, d_sqr))
        return false;
    if (is_fully_inside(c2.r, c1.r, d_sqr))
        return false;
    if (is_fully_outside(c1.r, c2.r, d_sqr))
        return false;
    return true;
}

/* ===================== */
/* vector2D - Line distance */
/* ===================== */
// get distance between p and projection of p on line <- a - b ->
double point_line_dist(vector2D &p, vector2D &a, vector2D &b)
{
    vector2D d = b - a;
    double t = d.dot(p - a) / d.norm2();
    return (a + d * t - p).norm();
}

/* ======================== */
/* vector2D - Segment distance */
/* ======================== */
// get distance between p and truncated projection of p on segment a ->
    b
double point_segment_dist(vector2D &p, vector2D &a, vector2D &b)
{
    if (a == b)
        return (p - a).norm(); // segment is a single vector2D
    vector2D d = b - a;        // direction
    double t = d.dot(p - a) / d.norm2();
    if (t <= 0)
        return (p - a).norm(); // truncate left
    if (t >= 1)
        return (p - b).norm(); // truncate right
    return (a + d * t - p).norm();
}
```

```
/* ===================================== */
/* Straight Line Hashing (integer coords) */
/* ===================================== */
// task: given 2 points p1, p2 with integer coordinates, output a unique
// representation {a,b,c} such that a*x + b*y + c = 0 is the equation
// of the straight line defined by p1, p2. This representation must be
// unique for each straight line, no matter which p1 and p2 are sampled.
struct Line
{
    int a, b, c;
};
int gcd(int a, int b)
{ // greatest common divisor
    a = abs(a);
    b = abs(b);
    while (b)
    {
        int c = a;
        a = b;
        b = c % b;
    }
    return a;
}
Line getLine(vector2D p1, vector2D p2)
{
    int a = p1.y - p2.y;
    int b = p2.x - p1.x;
    int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
    int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
    int f = gcd(a, gcd(b, c)) * sgn;
    a /= f;
    b /= f;
    c /= f;
    return {a, b, c};
}
```

1.2 Calculate Areas

1.2.1 Integration via Simpson's Method

```
#include "../../headers/headers.h"

//O(Evaluate f)=g(f)
//Numerical Integration of f in interval [a,b]
double simpsons_rule(function<double(double)> f, double a, double b)
{
    double c = (a + b) / 2;
    double h3 = abs(b - a) / 6;
    return h3 * (f(a) + 4 * f(c) + f(b));
}

//O(n g(f))
//Integrate f between a and b, using intervals of length (b-a)/n
double simpsons_rule(function<double(double)> f, double a, double b, int
      n)
{
    //n sets the precision for the result
    double ans = 0;
    double step = 0, h = (b - a) / n;
    rep(i, n)
    {
        ans += simpsons_rule(f, step, step + h);
        step += h;
    }
    return ans;
}
```

1.2.2 Green's Theorem

```
#include "../../headers/headers.h"

// O(1)
// Circle Arc
double arc(double theta, double phi)
{
}

// O(1)
// Line
double line(double x1, double y1, double x2, double y2)
{
}
```

1.3 Pick's Theorem

Given a simple polygon (no self intersections) in a lattice such that all vertices are grid points. Pick's theorem relates the Area A, points inside of the polygon i and the points of the border of the polygon b, in the following way:

$$A = i + \frac{b}{2} - 1$$