

```
1 bool fge(double x, double y) { return x >= y - eps; }
2 double fsqrt(double x) { return feq(x, 0) ? 0 : sqrt(x); }
3
4 // polygon
5
6 struct pt_t {
7     double x, y;
8     pt_t operator+(const pt_t &p) const { return { x + p.x, y + p.y }; }
9     pt_t operator-(const pt_t &p) const { return { x - p.x, y - p.y }; }
10    pt_t operator*(const double &c) const { return { x * c, y * c }; }
11    bool operator<(const pt_t &another) const {
12        return (x != another.x ? x < another.x : y < another.y);
13    }
14 };
15
16 // aX + bY + c = 0
17 struct line_t {
18     double a, b, c;
19 };
20
21 // (X - x)^2 + (Y - y)^2 = r^2
22 struct circle_t {
23     double x, y, r;
24 };
25
26 // normal vector = (a, b), passing p
27 line_t solve_line(double a, double b, pt_t p) {
28     return { a, b, -a * p.x - b * p.y };
29 }
30
31 // passing p, q
32 line_t solve_line(pt_t p, pt_t q) {
33     return solve_line(q.y - p.y, -q.x + p.x, p);
34 }
35
36 // t should be radius
37 pt_t rot(pt_t p, double r) {
38     return {
39         cos(r) * p.x - sin(r) * p.y,
40         sin(r) * p.x + cos(r) * p.y
41     };
42 }
43
44 double norm2(pt_t p) {
45     return p.x * p.x + p.y * p.y;
46 }
47
48 double norm(pt_t p) {
49     return sqrt(norm2(p));
50 }
51
52 double dist(line_t l, pt_t p) {
53     return abs(l.a * p.x + l.b * p.y + l.c)
54         / sqrt(l.a * l.a + l.b * l.b);
55 }
56
57 bool on_same_line(pt_t s, pt_t t, pt_t p) {
58     line_t l = solve_line(s, t);
59     if (feq(dist(l, p), 0)) return true;
60     else return false;
61 }
62
63 bool in_segment(pt_t s, pt_t t, pt_t p) {
64     line_t l = solve_line(s, t);
65     if (feq(dist(l, p), 0)
66         && fge(p.x, min(s.x, t.x))
67         && fge(max(s.x, t.x), p.x)
68         && fge(p.y, min(s.y, t.y))
69         && fge(max(s.y, t.y), p.y)) return true;
70     else return false;
71 }
```

```

72
73 // (NAN, NAN) if lines coincide with each other
74 // (INF, INF) if lines are parallel but not coincide
75 pt_t cross_point(line_t l, line_t m) {
76     double d = l.a * m.b - l.b * m.a;
77     if (feq(d, 0)) {
78         if (feq(l.a * m.c - l.c * m.a, 0)) return { INF, INF };
79         else return { NAN, NAN };
80     }
81     else {
82         double x = l.b * m.c - m.b * l.c;
83         double y = l.a * m.c - m.a * l.c;
84         return { x / d, y / -d };
85     }
86 }
87
88 // if size is 0, then not crossed
89 vector<pt_t> cross_point(circle_t f, line_t l) {
90     double d = dist(l, { f.x, f.y });
91     if (!fge(f.r, d)) return {};
92     line_t m = solve_line(l.b, -l.a, { f.x, f.y });
93     pt_t p = cross_point(l, m);
94     if (feq(d, f.r)) return { p };
95     else {
96         pt_t u = { l.b, -l.a };
97         pt_t v = u * (sqrt(pow(f.r, 2) - pow(d, 2)) / norm(u));
98         return { p + v, p - v };
99     }
100 }
101
102 // if size is 0, then not crossed
103 vector<pt_t> cross_point(circle_t f, circle_t g) {
104     line_t l = {
105         -2 * f.x + 2 * g.x,
106         -2 * f.y + 2 * g.y,
107         (f.x * f.x + f.y * f.y - f.r * f.r) - (g.x * g.x + g.y * g.y - g.r * g.r)
108     };
109     return cross_point(f, l);
110 }
111
112 // tangent points of f through p
113 // if size is 0, then p is strictly contained in f
114 // if size is 1, then p is on f
115 // otherwise size is 2
116 vector<pt_t> tangent_point(circle_t f, pt_t p) {
117     vector<pt_t> ret;
118     double d2 = norm2(pt_t({ f.x, f.y }) - p);
119     double r2 = d2 - f.r * f.r;
120     if (fge(r2, 0)) {
121         circle_t g = { p.x, p.y, fsqrt(r2) };
122         ret = cross_point(f, g);
123     }
124     return ret;
125 }
126
127 // tangent lines of f through p
128 // if size is 0, then p is strictly contained in f
129 // if size is 1, then p is on f
130 // otherwise size is 2
131 vector<line_t> tangent_line(circle_t f, pt_t p) {
132     vector<pt_t> qs = tangent_point(f, p);
133     vector<line_t> ret(qs.size());
134     Loop(i, ret.size()) {
135         ret[i] = solve_line(qs[i].x - f.x, qs[i].y - f.y, qs[i]);
136     }
137     return ret;
138 }
139
140 // tangent points on f through which there is a line tangent to g
141 // if size is 0, then one is strictly contained in the other
142 // if size is 1, then they are touched inside

```

```

143 // if size is 2, then they are crossed
144 // if size is 3, then they are touched outside
145 // otherwise size is 4
146 vector<pt_t> tangent_point(circle_t f, circle_t g) {
147     vector<pt_t> ret;
148     double d2 = norm2({ g.x - f.x, g.y - f.y });
149     vector<double> r2(2);
150     r2[0] = d2 - f.r * f.r + 2 * f.r * g.r;
151     r2[1] = d2 - f.r * f.r - 2 * f.r * g.r;
152     Loop(k, 2) {
153         if (fge(r2[k], 0)) {
154             circle_t g2 = { g.x, g.y, fsqrt(r2[k]) };
155             vector<pt_t> buf = cross_point(f, g2);
156             Loop(i, buf.size()) ret.push_back(buf[i]);
157         }
158     }
159     return ret;
160 }
161
162 // common tangent lines between two circles
163 // if size is 0, then one is strictly contained in the other
164 // if size is 1, then they are touched inside
165 // if size is 2, then they are crossed
166 // if size is 3, then they are touched outside
167 // otherwise size is 4
168 vector<line_t> tangent_line(circle_t f, circle_t g) {
169     vector<pt_t> qs = tangent_point(f, g);
170     vector<line_t> ret(qs.size());
171     Loop(i, ret.size()) {
172         ret[i] = tangent_line(f, qs[i]).front();
173     }
174     return ret;
175 }
176
177 // inner product
178 double dot(pt_t p, pt_t q) {
179     return p.x * q.x + p.y * q.y;
180 }
181
182 // outer product
183 double cross(pt_t p, pt_t q) {
184     return p.x * q.y - p.y * q.x;
185 }
186
187 // suppose a is counterclockwise, a.size() >= 3
188 double polygon_area(vector<pt_t> a) {
189     double ret = 0;
190     Loop(i, a.size()) {
191         int j = (i + 1 < a.size() ? i + 1 : 0);
192         ret += cross(a[i], a[j]);
193     }
194     ret = abs(ret) / 2;
195     return ret;
196 }
197
198 class Triangulate {
199 private:
200     vvi tri_ids;
201     vector<vector<pt_t>> tri_pts;
202     vector<pt_t> a;
203     bool enable(pt_t p, pt_t q, pt_t r) {
204         line_t l = solve_line(q, r);
205         if (feq(dist(l, p), 0)) return false;
206         if (fge(cross(q - p, r - p), 0)) return true;
207         else return false;
208     }
209     void contraction(vi &ids) {
210         int n = ids.size();
211         if (n < 3) return;
212         Loop(i, n) {
213             int id_p = (i - 1 + n) % n;

```

```

214     int id_q = i;
215     int id_r = (i + 1) % n;
216     pt_t p = a[ids[id_p]];
217     pt_t q = a[ids[id_q]];
218     pt_t r = a[ids[id_r]];
219     line_t l = solve_line(p, r);
220     if (feq(dist(l, q), 0)) {
221         ids.erase(ids.begin() + i);
222         contraction(ids);
223         return;
224     }
225 }
226 }
227 void divide(vi &ids) {
228     contraction(ids);
229     int n = ids.size();
230     if (n < 3) return;
231     Loop(i, n) {
232         int id_p = (i - 1 + n) % n;
233         int id_q = i;
234         int id_r = (i + 1) % n;
235         pt_t p = a[ids[id_p]];
236         pt_t q = a[ids[id_q]];
237         pt_t r = a[ids[id_r]];
238         if (enable(p, q, r)) {
239             line_t l = solve_line(p, r);
240             bool judge = true;
241             Loop(j, n) {
242                 if (j == id_p || j == id_q || j == id_r) continue;
243                 pt_t xp = a[ids[j]];
244                 if (in_triangle({ p, q, r }, xp)) judge = false;
245             }
246             if (judge) {
247                 tri_ids.push_back({ id_p, id_q, id_r });
248                 tri_pts.push_back({ p, q, r });
249                 ids.erase(ids.begin() + i);
250                 divide(ids);
251                 return;
252             }
253         }
254     }
255 }
256 int in_triangle(const vector<pt_t> &a, pt_t p) {
257     int ret = 2;
258     Loop(i, 3) {
259         int j = (i + 1) % 3;
260         line_t l = solve_line(a[i], a[j]);
261         double d = dist(l, p);
262         if (feq(d, 0)) ret = 1;
263         else if (fge(M_PI, angle(a[j] - a[i], p - a[i])))
264             else return 0;
265     }
266     return ret;
267 }
268 public:
269     // each triangle will be represented counterclockwisely
270     Triangulate(const vector<pt_t> &a) {
271         this->a = a;
272         vi ids(a.size());
273         Loop(i, ids.size()) ids[i] = i;
274         divide(ids);
275     }
276     vi get_ids() {
277         return tri_ids;
278     }
279     vector<vector<pt_t>> get_pts() {
280         return tri_pts;
281     }
282     // suppose a is counterclockwise, a.size() >= 3
283     // return 0 if not, return 1 if on line, return 2 if strictly included
284     int in_polygon(pt_t p) {

```

```
285     int ret = 0;
286     Loop(i, tri_pts.size()) {
287         if (in_triangle(tri_pts[i], p)) {
288             ret = 2;
289         }
290     }
291     if (ret != 0) {
292         Loop(i, a.size()) {
293             int j = (i + 1) % a.size();
294             if (in_segment(a[i], a[j], p)) ret = 1;
295         }
296     }
297     return ret;
298 }
299 };
300
301 vector<pt_t> convex_hull(vector<pt_t> ps) {
302     int n = ps.size();
303     sort(ps.begin(), ps.end());
304     Loop(i, n - 1) ps.push_back(ps[n - 2 - i]);
305     vector<pt_t> ret;
306     int m = 2;
307     Loop(i, n * 2 - 1) {
308         if (i == n) m = ret.size() + 1;
309         while (ret.size() >= m) {
310             int k = ret.size();
311             if (in_segment(ret[k - 2], ps[i], ret[k - 1])) break;
312             else if (fge(cross(ret[k - 1] - ret[k - 2], ps[i] - ret[k - 2]), 0)) ret.pop_back();
313             else break;
314         }
315         ret.push_back(ps[i]);
316     }
317     ret.pop_back();
318     reverse(ret.begin(), ret.end());
319     return ret;
320 }
321
322 // angle [0, 2PI) of vector p to vector q
323 double angle(pt_t p, pt_t q) {
324     p = p * (1.0 / norm(p));
325     q = q * (1.0 / norm(q));
326     double r0 = acos(max(min(dot(p, q), 1.0), -1.0));
327     double r1 = asin(max(min(dot(p, q), 1.0), -1.0));
328     if (r1 >= 0) return r0;
329     else return 2 * M_PI - r0;
330 }
```