计算几何

目录

㈠ 点的基本运算

1. 平面上两点之间距离 1

2. 判断两点是否重合 1

3. 矢量叉乘 1

4. 矢量点乘 2

5. 判断点是否在线段上 2

6. 求一点饶某点旋转后的坐标 2

7. 求矢量夹角 2

㈡ 线段及直线的基本运算

1. 点与线段的关系 3

2. 求点到线段所在直线垂线的垂足 4

3. 点到线段的最近点 4

4. 点到线段所在直线的距离 4

5. 点到折线集的最近距离 4

6. 判断圆是否在多边形内 5

7. 求矢量夹角余弦 5

8. 求线段之间的夹角 5

9. 判断线段是否相交 6

10.判断线段是否相交但不交在端点处 6

11.求线段所在直线的方程 6

12.求直线的斜率 7

13.求直线的倾斜角 7

14.求点关于某直线的对称点 7

15.判断两条直线是否相交及求直线交点 7

16.判断线段是否相交，如果相交返回交点 7

㈢ 多边形常用算法模块

1. 判断多边形是否简单多边形 8

2. 检查多边形顶点的凸凹性 9

3. 判断多边形是否凸多边形 9

4. 求多边形面积 9

5. 判断多边形顶点的排列方向，方法一 10

6. 判断多边形顶点的排列方向，方法二 10

7. 射线法判断点是否在多边形内 10

8. 判断点是否在凸多边形内 11

9. 寻找点集的graham算法 12

10.寻找点集凸包的卷包裹法 13

11.判断线段是否在多边形内 14

12.求简单多边形的重心 15

13.求凸多边形的重心 17

14.求肯定在给定多边形内的一个点 17

15.求从多边形外一点出发到该多边形的切线 18

16.判断多边形的核是否存在 19

㈣ 圆的基本运算

.点是否在圆内 20

.求不共线的三点所确定的圆 21

㈤ 矩形的基本运算

1.已知矩形三点坐标，求第4点坐标 22

㈥ 常用算法的描述 22

㈦ 补充

1．两圆关系： 24

2．判断圆是否在矩形内： 24

3．点到平面的距离： 25

4．点是否在直线同侧： 25

5．镜面反射线： 25

6．矩形包含： 26

7．两圆交点： 27

8．两圆公共面积： 28

9. 圆和直线关系： 29

10. 内切圆： 30

11. 求切点： 31

12. 线段的左右旋： 31

13．公式： 32

\*/

/\* 需要包含的头文件 \*/

#include <cmath >

/\* 常用的常量定义 \*/

const double INF = 1E200

const double EP = 1E-10

const int MAXV = 300

const double PI = 3.14159265

/\* 基本几何结构 \*/

struct POINT

{

double x;

double y;

POINT(double a=0, double b=0) { x=a; y=b;} //constructor

};

struct LINESEG

{

POINT s;

POINT e;

LINESEG(POINT a, POINT b) { s=a; e=b;}

LINESEG() { }

};

struct LINE // 直线的解析方程 a\*x+b\*y+c=0 为统一表示，约定 a >= 0

{

double a;

double b;

double c;

LINE(double d1=1, double d2=-1, double d3=0) {a=d1; b=d2; c=d3;}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* 点的基本运算 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double dist(POINT p1,POINT p2) // 返回两点之间欧氏距离

{

return( sqrt( (p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y) ) );

}

bool equal\_point(POINT p1,POINT p2) // 判断两个点是否重合

{

return ( (abs(p1.x-p2.x)<EP)&&(abs(p1.y-p2.y)<EP) );

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

r=multiply(sp,ep,op),得到(sp-op)和(ep-op)的叉积

r>0：ep在矢量opsp的逆时针方向；

r=0：opspep三点共线；

r<0：ep在矢量opsp的顺时针方向

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double multiply(POINT sp,POINT ep,POINT op)

{

return((sp.x-op.x)\*(ep.y-op.y)-(ep.x-op.x)\*(sp.y-op.y));

}

/\*

r=dotmultiply(p1,p2,op),得到矢量(p1-op)和(p2-op)的点积，如果两个矢量都非零矢量

r<0：两矢量夹角为钝角；

r=0：两矢量夹角为直角；

r>0：两矢量夹角为锐角

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double dotmultiply(POINT p1,POINT p2,POINT p0)

{

return ((p1.x-p0.x)\*(p2.x-p0.x)+(p1.y-p0.y)\*(p2.y-p0.y));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

判断点p是否在线段l上

条件：(p在线段l所在的直线上) && (点p在以线段l为对角线的矩形内)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool online(LINESEG l,POINT p)

{

return( (multiply(l.e,p,l.s)==0) &&( ( (p.x-l.s.x)\*(p.x-l.e.x)<=0 )&&( (p.y-l.s.y)\*(p.y-l.e.y)<=0 ) ) );

}

// 返回点p以点o为圆心逆时针旋转alpha(单位：弧度)后所在的位置

POINT rotate(POINT o,double alpha,POINT p)

{

POINT tp;

p.x-=o.x;

p.y-=o.y;

tp.x=p.x\*cos(alpha)-p.y\*sin(alpha)+o.x;

tp.y=p.y\*cos(alpha)+p.x\*sin(alpha)+o.y;

return tp;

}

/\* 返回顶角在o点，起始边为os，终止边为oe的夹角(单位：弧度)

角度小于pi，返回正值

角度大于pi，返回负值

可以用于求线段之间的夹角

原理：

r = dotmultiply(s,e,o) / (dist(o,s)\*dist(o,e))

r'= multiply(s,e,o)

r >= 1 angle = 0;

r <= -1 angle = -PI

-1<r<1 && r'>0 angle = arccos(r)

-1<r<1 && r'<=0 angle = -arccos(r)

\*/

double angle(POINT o,POINT s,POINT e)

{

double cosfi,fi,norm;

double dsx = s.x - o.x;

double dsy = s.y - o.y;

double dex = e.x - o.x;

double dey = e.y - o.y;

cosfi=dsx\*dex+dsy\*dey;

norm=(dsx\*dsx+dsy\*dsy)\*(dex\*dex+dey\*dey);

cosfi /= sqrt( norm );

if (cosfi >= 1.0 ) return 0;

if (cosfi <= -1.0 ) return -3.1415926;

fi=acos(cosfi);

if (dsx\*dey-dsy\*dex>0) return fi; // 说明矢量os 在矢量 oe的顺时针方向

return -fi;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 线段及直线的基本运算 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 判断点与线段的关系,用途很广泛

本函数是根据下面的公式写的，P是点C到线段AB所在直线的垂足

AC dot AB

r = ---------

||AB||^2

(Cx-Ax)(Bx-Ax) + (Cy-Ay)(By-Ay)

= -------------------------------

L^2

r has the following meaning:

r=0 P = A

r=1 P = B

r<0 P is on the backward extension of AB

r>1 P is on the forward extension of AB

0<r<1 P is interior to AB

\*/

double relation(POINT p,LINESEG l)

{

LINESEG tl;

tl.s=l.s;

tl.e=p;

return dotmultiply(tl.e,l.e,l.s)/(dist(l.s,l.e)\*dist(l.s,l.e));

}

// 求点C到线段AB所在直线的垂足 P

POINT perpendicular(POINT p,LINESEG l)

{

double r=relation(p,l);

POINT tp;

tp.x=l.s.x+r\*(l.e.x-l.s.x);

tp.y=l.s.y+r\*(l.e.y-l.s.y);

return tp;

}

/\* 求点p到线段l的最短距离,并返回线段上距该点最近的点np

注意：np是线段l上到点p最近的点，不一定是垂足 \*/

double ptolinesegdist(POINT p,LINESEG l,POINT &np)

{

double r=relation(p,l);

if(r<0)

{

np=l.s;

return dist(p,l.s);

}

if(r>1)

{

np=l.e;

return dist(p,l.e);

}

np=perpendicular(p,l);

return dist(p,np);

}

// 求点p到线段l所在直线的距离,请注意本函数与上个函数的区别

double ptoldist(POINT p,LINESEG l)

{

return abs(multiply(p,l.e,l.s))/dist(l.s,l.e);

}

/\* 计算点到折线集的最近距离,并返回最近点.

注意：调用的是ptolineseg()函数 \*/

double ptopointset(int vcount,POINT pointset[],POINT p,POINT &q)

{

int i;

double cd=double(INF),td;

LINESEG l;

POINT tq,cq;

for(i=0;i<vcount-1;i++)

{

l.s=pointset[i];

l.e=pointset[i+1];

td=ptolinesegdist(p,l,tq);

if(td<cd)

{

cd=td;

cq=tq;

}

}

q=cq;

return cd;

}

/\* 判断圆是否在多边形内.ptolineseg()函数的应用2 \*/

bool CircleInsidePolygon(int vcount,POINT center,double radius,POINT polygon[])

{

POINT q;

double d;

q.x=0;

q.y=0;

d=ptopointset(vcount,polygon,center,q);

if(d<radius||fabs(d-radius)<EP)

return true;

else

return false;

}

/\* 返回两个矢量l1和l2的夹角的余弦(-1 --- 1)注意：如果想从余弦求夹角的话，注意反余弦函数的定义域是从 0到pi \*/

double cosine(LINESEG l1,LINESEG l2)

{

return (((l1.e.x-l1.s.x)\*(l2.e.x-l2.s.x) +

(l1.e.y-l1.s.y)\*(l2.e.y-l2.s.y))/(dist(l1.e,l1.s)\*dist(l2.e,l2.s))) );

}

// 返回线段l1与l2之间的夹角 单位：弧度 范围(-pi，pi)

double lsangle(LINESEG l1,LINESEG l2)

{

POINT o,s,e;

o.x=o.y=0;

s.x=l1.e.x-l1.s.x;

s.y=l1.e.y-l1.s.y;

e.x=l2.e.x-l2.s.x;

e.y=l2.e.y-l2.s.y;

return angle(o,s,e);

}

// 如果线段u和v相交(包括相交在端点处)时，返回true

//

//判断P1P2跨立Q1Q2的依据是：( P1 - Q1 ) × ( Q2 - Q1 ) \* ( Q2 - Q1 ) × ( P2 - Q1 ) >= 0。

//判断Q1Q2跨立P1P2的依据是：( Q1 - P1 ) × ( P2 - P1 ) \* ( P2 - P1 ) × ( Q2 - P1 ) >= 0。

bool intersect(LINESEG u,LINESEG v)

{

return( (max(u.s.x,u.e.x)>=min(v.s.x,v.e.x))&& //排斥实验

(max(v.s.x,v.e.x)>=min(u.s.x,u.e.x))&&

(max(u.s.y,u.e.y)>=min(v.s.y,v.e.y))&&

(max(v.s.y,v.e.y)>=min(u.s.y,u.e.y))&&

(multiply(v.s,u.e,u.s)\*multiply(u.e,v.e,u.s)>=0)&& //跨立实验

(multiply(u.s,v.e,v.s)\*multiply(v.e,u.e,v.s)>=0));

}

// (线段u和v相交)&&(交点不是双方的端点) 时返回true

bool intersect\_A(LINESEG u,LINESEG v)

{

return ((intersect(u,v))&&

(!online(u,v.s))&&

(!online(u,v.e))&&

(!online(v,u.e))&&

(!online(v,u.s)));

}

// 线段v所在直线与线段u相交时返回true；方法：判断线段u是否跨立线段v

bool intersect\_l(LINESEG u,LINESEG v)

{

return multiply(u.s,v.e,v.s)\*multiply(v.e,u.e,v.s)>=0;

}

// 根据已知两点坐标，求过这两点的直线解析方程： a\*x+b\*y+c = 0 (a >= 0)

LINE makeline(POINT p1,POINT p2)

{

LINE tl;

int sign = 1;

tl.a=p2.y-p1.y;

if(tl.a<0)

{

sign = -1;

tl.a=sign\*tl.a;

}

tl.b=sign\*(p1.x-p2.x);

tl.c=sign\*(p1.y\*p2.x-p1.x\*p2.y);

return tl;

}

// 根据直线解析方程返回直线的斜率k,水平线返回 0,竖直线返回 1e200

double slope(LINE l)

{

if(abs(l.a) < 1e-20)

return 0;

if(abs(l.b) < 1e-20)

return INF;

return -(l.a/l.b);

}

// 返回直线的倾斜角alpha ( 0 - pi)

double alpha(LINE l)

{

if(abs(l.a)< EP)

return 0;

if(abs(l.b)< EP)

return PI/2;

double k=slope(l);

if(k>0)

return atan(k);

else

return PI+atan(k);

}

// 求点p关于直线l的对称点

POINT symmetry(LINE l,POINT p)

{

POINT tp;

tp.x=((l.b\*l.b-l.a\*l.a)\*p.x-2\*l.a\*l.b\*p.y-2\*l.a\*l.c)/(l.a\*l.a+l.b\*l.b);

tp.y=((l.a\*l.a-l.b\*l.b)\*p.y-2\*l.a\*l.b\*p.x-2\*l.b\*l.c)/(l.a\*l.a+l.b\*l.b);

return tp;

}

// 如果两条直线 l1(a1\*x+b1\*y+c1 = 0), l2(a2\*x+b2\*y+c2 = 0)相交，返回true，且返回交点p

bool lineintersect(LINE l1,LINE l2,POINT &p) // 是 L1，L2

{

double d=l1.a\*l2.b-l2.a\*l1.b;

if(abs(d)<EP) // 不相交

return false;

p.x = (l2.c\*l1.b-l1.c\*l2.b)/d;

p.y = (l2.a\*l1.c-l1.a\*l2.c)/d;

return true;

}

// 如果线段l1和l2相交，返回true且交点由(inter)返回，否则返回false

bool intersection(LINESEG l1,LINESEG l2,POINT &inter)

{

LINE ll1,ll2;

ll1=makeline(l1.s,l1.e);

ll2=makeline(l2.s,l2.e);

if(lineintersect(ll1,ll2,inter))

return online(l1,inter) && online(l2,inter);

else

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 多边形常用算法模块 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 如果无特别说明，输入多边形顶点要求按逆时针排列

/\*

返回值：输入的多边形是简单多边形，返回true

要 求：输入顶点序列按逆时针排序

说 明：简单多边形定义：

1：循环排序中相邻线段对的交是他们之间共有的单个点

2：不相邻的线段不相交

本程序默认第一个条件已经满足

\*/

bool issimple(int vcount,POINT polygon[])

{

int i,cn;

LINESEG l1,l2;

for(i=0;i<vcount;i++)

{

l1.s=polygon[i];

l1.e=polygon[(i+1)%vcount];

cn=vcount-3;

while(cn)

{

l2.s=polygon[(i+2)%vcount];

l2.e=polygon[(i+3)%vcount];

if(intersect(l1,l2))

break;

cn--;

}

if(cn)

return false;

}

return true;

}

// 返回值：按输入顺序返回多边形顶点的凸凹性判断，bc[i]=1,iff:第i个顶点是凸顶点

void checkconvex(int vcount,POINT polygon[],bool bc[])

{

int i,index=0;

POINT tp=polygon[0];

for(i=1;i<vcount;i++) // 寻找第一个凸顶点

{

if(polygon[i].y<tp.y||(polygon[i].y == tp.y&&polygon[i].x<tp.x))

{

tp=polygon[i];

index=i;

}

}

int count=vcount-1;

bc[index]=1;

while(count) // 判断凸凹性

{

if(multiply(polygon[(index+1)%vcount],polygon[(index+2)%vcount],polygon[index])>=0 )

bc[(index+1)%vcount]=1;

else

bc[(index+1)%vcount]=0;

index++;

count--;

}

}

// 返回值：多边形polygon是凸多边形时，返回true

bool isconvex(int vcount,POINT polygon[])

{

bool bc[MAXV];

checkconvex(vcount,polygon,bc);

for(int i=0;i<vcount;i++) // 逐一检查顶点，是否全部是凸顶点

if(!bc[i])

return false;

return true;

}

// 返回多边形面积(signed)；输入顶点按逆时针排列时，返回正值；否则返回负值

double area\_of\_polygon(int vcount,POINT polygon[])

{

int i;

double s;

if (vcount<3)

return 0;

s=polygon[0].y\*(polygon[vcount-1].x-polygon[1].x);

for (i=1;i<vcount;i++)

s+=polygon[i].y\*(polygon[(i-1)].x-polygon[(i+1)%vcount].x);

return s/2;

}

// 如果输入顶点按逆时针排列，返回true

bool isconterclock(int vcount,POINT polygon[])

{

return area\_of\_polygon(vcount,polygon)>0;

}

// 另一种判断多边形顶点排列方向的方法

bool isccwize(int vcount,POINT polygon[])

{

int i,index;

POINT a,b,v;

v=polygon[0];

index=0;

for(i=1;i<vcount;i++) // 找到最低且最左顶点，肯定是凸顶点

{

if(polygon[i].y<v.y||polygon[i].y == v.y && polygon[i].x<v.x)

{

index=i;

}

}

a=polygon[(index-1+vcount)%vcount]; // 顶点v的前一顶点

b=polygon[(index+1)%vcount]; // 顶点v的后一顶点

return multiply(v,b,a)>0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

射线法判断点q与多边形polygon的位置关系，要求polygon为简单多边形，顶点逆时针排列

如果点在多边形内： 返回0

如果点在多边形边上： 返回1

如果点在多边形外： 返回2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int insidepolygon(int vcount,POINT Polygon[],POINT q)

{

int c=0,i,n;

LINESEG l1,l2;

bool bintersect\_a,bonline1,bonline2,bonline3;

double r1,r2;

l1.s=q;

l1.e=q;

l1.e.x=double(INF);

n=vcount;

for (i=0;i<vcount;i++)

{

l2.s=Polygon[i];

l2.e=Polygon[(i+1)%n];

if(online(l2,q))

return 1; // 如果点在边上，返回1

if ( (bintersect\_a=intersect\_A(l1,l2))|| // 相交且不在端点

( (bonline1=online(l1,Polygon[(i+1)%n]))&& // 第二个端点在射线上

( (!(bonline2=online(l1,Polygon[(i+2)%n])))&& /\* 前一个端点和后一个端点在射线两侧 \*/

((r1=multiply(Polygon[i],Polygon[(i+1)%n],l1.s)\*multiply(Polygon[(i+1)%n],Polygon[(i+2)%n],l1.s))>0) ||

(bonline3=online(l1,Polygon[(i+2)%n]))&& /\* 下一条边是水平线，前一个端点和后一个端点在射线两侧 \*/

((r2=multiply(Polygon[i],Polygon[(i+2)%n],l1.s)\*multiply(Polygon[(i+2)%n],

Polygon[(i+3)%n],l1.s))>0)

)

)

) c++;

}

if(c%2 == 1)

return 0;

else

return 2;

}

//点q是凸多边形polygon内时，返回true；注意：多边形polygon一定要是凸多边形

bool InsideConvexPolygon(int vcount,POINT polygon[],POINT q) // 可用于三角形！

{

POINT p;

LINESEG l;

int i;

p.x=0;p.y=0;

for(i=0;i<vcount;i++) // 寻找一个肯定在多边形polygon内的点p：多边形顶点平均值

{

p.x+=polygon[i].x;

p.y+=polygon[i].y;

}

p.x /= vcount;

p.y /= vcount;

for(i=0;i<vcount;i++)

{

l.s=polygon[i];l.e=polygon[(i+1)%vcount];

if(multiply(p,l.e,l.s)\*multiply(q,l.e,l.s)<0) /\* 点p和点q在边l的两侧，说明点q肯定在多边形外 \*/

break;

}

return (i==vcount);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

寻找凸包的graham 扫描法

PointSet为输入的点集；

ch为输出的凸包上的点集，按照逆时针方向排列;

n为PointSet中的点的数目

len为输出的凸包上的点的个数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Graham\_scan(POINT PointSet[],POINT ch[],int n,int &len)

{

int i,j,k=0,top=2;

POINT tmp;

// 选取PointSet中y坐标最小的点PointSet[k]，如果这样的点有多个，则取最左边的一个

for(i=1;i<n;i++)

if ( PointSet[i].y<PointSet[k].y || (PointSet[i].y==PointSet[k].y) && (PointSet[i].x<PointSet[k].x) )

k=i;

tmp=PointSet[0];

PointSet[0]=PointSet[k];

PointSet[k]=tmp; // 现在PointSet中y坐标最小的点在PointSet[0]

for (i=1;i<n-1;i++) /\* 对顶点按照相对PointSet[0]的极角从小到大进行排序，极角相同的按照距离PointSet[0]从近到远进行排序 \*/

{

k=i;

for (j=i+1;j<n;j++)

if ( multiply(PointSet[j],PointSet[k],PointSet[0])>0 || // 极角更小

(multiply(PointSet[j],PointSet[k],PointSet[0])==0) && /\* 极角相等，距离更短 \*/

dist(PointSet[0],PointSet[j])<dist(PointSet[0],PointSet[k])

)

k=j;

tmp=PointSet[i];

PointSet[i]=PointSet[k];

PointSet[k]=tmp;

}

ch[0]=PointSet[0];

ch[1]=PointSet[1];

ch[2]=PointSet[2];

for (i=3;i<n;i++)

{

while (multiply(PointSet[i],ch[top],ch[top-1])>=0)

top--;

ch[++top]=PointSet[i];

}

len=top+1;

}

// 卷包裹法求点集凸壳，参数说明同graham算法

void ConvexClosure(POINT PointSet[],POINT ch[],int n,int &len)

{

int top=0,i,index,first;

double curmax,curcos,curdis;

POINT tmp;

LINESEG l1,l2;

bool use[MAXV];

tmp=PointSet[0];

index=0;

// 选取y最小点，如果多于一个，则选取最左点

for(i=1;i<n;i++)

{

if(PointSet[i].y<tmp.y||PointSet[i].y == tmp.y&&PointSet[i].x<tmp.x)

{

index=i;

}

use[i]=false;

}

tmp=PointSet[index];

first=index;

use[index]=true;

index=-1;

ch[top++]=tmp;

tmp.x-=100;

l1.s=tmp;

l1.e=ch[0];

l2.s=ch[0];

while(index!=first)

{

curmax=-100;

curdis=0;

// 选取与最后一条确定边夹角最小的点，即余弦值最大者

for(i=0;i<n;i++)

{

if(use[i])continue;

l2.e=PointSet[i];

curcos=cosine(l1,l2); // 根据cos值求夹角余弦，范围在 （-1 -- 1 ）

if(curcos>curmax || fabs(curcos-curmax)<1e-6 && dist(l2.s,l2.e)>curdis)

{

curmax=curcos;

index=i;

curdis=dist(l2.s,l2.e);

}

}

use[first]=false; //清空第first个顶点标志，使最后能形成封闭的hull

use[index]=true;

ch[top++]=PointSet[index];

l1.s=ch[top-2];

l1.e=ch[top-1];

l2.s=ch[top-1];

}

len=top-1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

判断线段是否在简单多边形内(注意：如果多边形是凸多边形，下面的算法可以化简)

必要条件一：线段的两个端点都在多边形内；

必要条件二：线段和多边形的所有边都不内交；

用途： 1. 判断折线是否在简单多边形内

2. 判断简单多边形是否在另一个简单多边形内

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool LinesegInsidePolygon(int vcount,POINT polygon[],LINESEG l)

{

// 判断线端l的端点是否不都在多边形内

if(!insidepolygon(vcount,polygon,l.s)||!insidepolygon(vcount,polygon,l.e))

return false;

int top=0,i,j;

POINT PointSet[MAXV],tmp;

LINESEG s;

for(i=0;i<vcount;i++)

{

s.s=polygon[i];

s.e=polygon[(i+1)%vcount];

if(online(s,l.s)) //线段l的起始端点在线段s上

PointSet[top++]=l.s;

else if(online(s,l.e)) //线段l的终止端点在线段s上

PointSet[top++]=l.e;

else

{

if(online(l,s.s)) //线段s的起始端点在线段l上

PointSet[top++]=s.s;

else if(online(l,s.e)) // 线段s的终止端点在线段l上

PointSet[top++]=s.e;

else

{

if(intersect(l,s)) // 这个时候如果相交，肯定是内交，返回false

return false;

}

}

}

for(i=0;i<top-1;i++) /\* 冒泡排序，x坐标小的排在前面；x坐标相同者，y坐标小的排在前面 \*/

{

for(j=i+1;j<top;j++)

{

if( PointSet[i].x>PointSet[j].x || fabs(PointSet[i].x-PointSet[j].x)<EP && PointSet[i].y>PointSet[j].y )

{

tmp=PointSet[i];

PointSet[i]=PointSet[j];

PointSet[j]=tmp;

}

}

}

for(i=0;i<top-1;i++)

{

tmp.x=(PointSet[i].x+PointSet[i+1].x)/2; //得到两个相邻交点的中点

tmp.y=(PointSet[i].y+PointSet[i+1].y)/2;

if(!insidepolygon(vcount,polygon,tmp))

return false;

}

return true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

求任意简单多边形polygon的重心

需要调用下面几个函数：

void AddPosPart(); 增加右边区域的面积

void AddNegPart(); 增加左边区域的面积

void AddRegion(); 增加区域面积

在使用该程序时，如果把xtr,ytr,wtr,xtl,ytl,wtl设成全局变量就可以使这些函数的形式得到化简,

但要注意函数的声明和调用要做相应变化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void AddPosPart(double x, double y, double w, double &xtr, double &ytr, double &wtr)

{

if (abs(wtr + w)<1e-10 ) return; // detect zero regions

xtr = ( wtr\*xtr + w\*x ) / ( wtr + w );

ytr = ( wtr\*ytr + w\*y ) / ( wtr + w );

wtr = w + wtr;

return;

}

void AddNegPart(double x, ouble y, double w, double &xtl, double &ytl, double &wtl)

{

if ( abs(wtl + w)<1e-10 )

return; // detect zero regions

xtl = ( wtl\*xtl + w\*x ) / ( wtl + w );

ytl = ( wtl\*ytl + w\*y ) / ( wtl + w );

wtl = w + wtl;

return;

}

void AddRegion ( double x1, double y1, double x2, double y2, double &xtr, double &ytr,

double &wtr, double &xtl, double &ytl, double &wtl )

{

if ( abs (x1 - x2)< 1e-10 )

return;

if ( x2 > x1 )

{

AddPosPart ((x2+x1)/2, y1/2, (x2-x1) \* y1,xtr,ytr,wtr); /\* rectangle 全局变量变化处 \*/

AddPosPart ((x1+x2+x2)/3, (y1+y1+y2)/3, (x2-x1)\*(y2-y1)/2,xtr,ytr,wtr);

// triangle 全局变量变化处

}

else

{

AddNegPart ((x2+x1)/2, y1/2, (x2-x1) \* y1,xtl,ytl,wtl);

// rectangle 全局变量变化处

AddNegPart ((x1+x2+x2)/3, (y1+y1+y2)/3, (x2-x1)\*(y2-y1)/2,xtl,ytl,wtl);

// triangle 全局变量变化处

}

}

POINT cg\_simple(int vcount,POINT polygon[])

{

double xtr,ytr,wtr,xtl,ytl,wtl;

//注意： 如果把xtr,ytr,wtr,xtl,ytl,wtl改成全局变量后这里要删去

POINT p1,p2,tp;

xtr = ytr = wtr = 0.0;

xtl = ytl = wtl = 0.0;

for(int i=0;i<vcount;i++)

{

p1=polygon[i];

p2=polygon[(i+1)%vcount];

AddRegion(p1.x,p1.y,p2.x,p2.y,xtr,ytr,wtr,xtl,ytl,wtl); //全局变量变化处

}

tp.x = (wtr\*xtr + wtl\*xtl) / (wtr + wtl);

tp.y = (wtr\*ytr + wtl\*ytl) / (wtr + wtl);

return tp;

}

// 求凸多边形的重心,要求输入多边形按逆时针排序

POINT gravitycenter(int vcount,POINT polygon[])

{

POINT tp;

double x,y,s,x0,y0,cs,k;

x=0;y=0;s=0;

for(int i=1;i<vcount-1;i++)

{

x0=(polygon[0].x+polygon[i].x+polygon[i+1].x)/3;

y0=(polygon[0].y+polygon[i].y+polygon[i+1].y)/3; //求当前三角形的重心

cs=multiply(polygon[i],polygon[i+1],polygon[0])/2;

//三角形面积可以直接利用该公式求解

if(abs(s)<1e-20)

{

x=x0;y=y0;s+=cs;continue;

}

k=cs/s; //求面积比例

x=(x+k\*x0)/(1+k);

y=(y+k\*y0)/(1+k);

s += cs;

}

tp.x=x;

tp.y=y;

return tp;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

给定一简单多边形，找出一个肯定在该多边形内的点

定理1 ：每个多边形至少有一个凸顶点

定理2 ：顶点数>=4的简单多边形至少有一条对角线

结论 ： x坐标最大，最小的点肯定是凸顶点

y坐标最大，最小的点肯定是凸顶点

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

POINT a\_point\_insidepoly(int vcount,POINT polygon[])

{

POINT v,a,b,r;

int i,index;

v=polygon[0];

index=0;

for(i=1;i<vcount;i++) //寻找一个凸顶点

{

if(polygon[i].y<v.y)

{

v=polygon[i];

index=i;

}

}

a=polygon[(index-1+vcount)%vcount]; //得到v的前一个顶点

b=polygon[(index+1)%vcount]; //得到v的后一个顶点

POINT tri[3],q;

tri[0]=a;tri[1]=v;tri[2]=b;

double md=INF;

int in1=index;

bool bin=false;

for(i=0;i<vcount;i++) //寻找在三角形avb内且离顶点v最近的顶点q

{

if(i == index)continue;

if(i == (index-1+vcount)%vcount)continue;

if(i == (index+1)%vcount)continue;

if(!InsideConvexPolygon(3,tri,polygon[i]))continue;

bin=true;

if(dist(v,polygon[i])<md)

{

q=polygon[i];

md=dist(v,q);

}

}

if(!bin) //没有顶点在三角形avb内，返回线段ab中点

{

r.x=(a.x+b.x)/2;

r.y=(a.y+b.y)/2;

return r;

}

r.x=(v.x+q.x)/2; //返回线段vq的中点

r.y=(v.y+q.y)/2;

return r;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

求从多边形外一点p出发到一个简单多边形的切线,如果存在返回切点,其中rp点是右切点,lp是左切点

注意：p点一定要在多边形外 ,输入顶点序列是逆时针排列

原 理： 如果点在多边形内肯定无切线;凸多边形有唯一的两个切点,凹多边形就可能有多于两个的切点;

如果polygon是凸多边形，切点只有两个只要找到就可以,可以化简此算法

如果是凹多边形还有一种算法可以求解:先求凹多边形的凸包,然后求凸包的切线

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void pointtangentpoly(int vcount,POINT polygon[],POINT p,POINT &rp,POINT &lp)

{

LINESEG ep,en;

bool blp,bln;

rp=polygon[0];

lp=polygon[0];

for(int i=1;i<vcount;i++)

{

ep.s=polygon[(i+vcount-1)%vcount];

ep.e=polygon[i];

en.s=polygon[i];

en.e=polygon[(i+1)%vcount];

blp=multiply(ep.e,p,ep.s)>=0; // p is to the left of pre edge

bln=multiply(en.e,p,en.s)>=0; // p is to the left of next edge

if(!blp&&bln)

{

if(multiply(polygon[i],rp,p)>0) // polygon[i] is above rp

rp=polygon[i];

}

if(blp&&!bln)

{

if(multiply(lp,polygon[i],p)>0) // polygon[i] is below lp

lp=polygon[i];

}

}

return ;

}

// 如果多边形polygon的核存在，返回true，返回核上的一点p.顶点按逆时针方向输入

bool core\_exist(int vcount,POINT polygon[],POINT &p)

{

int i,j,k;

LINESEG l;

LINE lineset[MAXV];

for(i=0;i<vcount;i++)

{

lineset[i]=makeline(polygon[i],polygon[(i+1)%vcount]);

}

for(i=0;i<vcount;i++)

{

for(j=0;j<vcount;j++)

{

if(i == j)continue;

if(lineintersect(lineset[i],lineset[j],p))

{

for(k=0;k<vcount;k++)

{

l.s=polygon[k];

l.e=polygon[(k+1)%vcount];

if(multiply(p,l.e,l.s)>0)

//多边形顶点按逆时针方向排列，核肯定在每条边的左侧或边上

break;

}

if(k == vcount) //找到了一个核上的点

break;

}

}

if(j<vcount) break;

}

if(i<vcount)

return true;

else

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 圆的基本运算 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

返回值 ： 点p在圆内(包括边界)时，返回true

用途 ： 因为圆为凸集，所以判断点集，折线，多边形是否在圆内时，

只需要逐一判断点是否在圆内即可。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool point\_in\_circle(POINT o,double r,POINT p)

{

double d2=(p.x-o.x)\*(p.x-o.x)+(p.y-o.y)\*(p.y-o.y);

double r2=r\*r;

return d2<r2||abs(d2-r2)<EP;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

用 途 ：求不共线的三点确定一个圆

输 入 ：三个点p1,p2,p3

返回值 ：如果三点共线，返回false；反之，返回true。圆心由q返回，半径由r返回

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool cocircle(POINT p1,POINT p2,POINT p3,POINT &q,double &r)

{

double x12=p2.x-p1.x;

double y12=p2.y-p1.y;

double x13=p3.x-p1.x;

double y13=p3.y-p1.y;

double z2=x12\*(p1.x+p2.x)+y12\*(p1.y+p2.y);

double z3=x13\*(p1.x+p3.x)+y13\*(p1.y+p3.y);

double d=2.0\*(x12\*(p3.y-p2.y)-y12\*(p3.x-p2.x));

if(abs(d)<EP) //共线，圆不存在

return false;

q.x=(y13\*z2-y12\*z3)/d;

q.y=(x12\*z3-x13\*z2)/d;

r=dist(p1,q);

return true;

}

int line\_circle(LINE l,POINT o,double r,POINT &p1,POINT &p2)

{

return true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 矩形的基本运算 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

说明：因为矩形的特殊性，常用算法可以化简：

1.判断矩形是否包含点

只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间。

2.判断线段、折线、多边形是否在矩形中

因为矩形是个凸集，所以只要判断所有端点是否都在矩形中就可以了。

3.判断圆是否在矩形中

圆在矩形中的充要条件是：圆心在矩形中且圆的半径小于等于圆心到矩形四边的距离的最小值。

\*/

// 已知矩形的三个顶点(a,b,c)，计算第四个顶点d的坐标. 注意：已知的三个顶点可以是无序的

POINT rect4th(POINT a,POINT b,POINT c)

{

POINT d;

if(abs(dotmultiply(a,b,c))<EP) // 说明c点是直角拐角处

{

d.x=a.x+b.x-c.x;

d.y=a.y+b.y-c.y;

}

if(abs(dotmultiply(a,c,b))<EP) // 说明b点是直角拐角处

{

d.x=a.x+c.x-b.x;

d.y=a.y+c.y-b.x;

}

if(abs(dotmultiply(c,b,a))<EP) // 说明a点是直角拐角处

{

d.x=c.x+b.x-a.x;

d.y=c.y+b.y-a.y;

}

return d;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 常用算法的描述 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

尚未实现的算法：

1. 求包含点集的最小圆

2. 求多边形的交

3. 简单多边形的三角剖分

4. 寻找包含点集的最小矩形

5. 折线的化简

6. 判断矩形是否在矩形中

7. 判断矩形能否放在矩形中

8. 矩形并的面积与周长

9. 矩形并的轮廓

10.矩形并的闭包

11.矩形的交

12.点集中的最近点对

13.多边形的并

14.圆的交与并

15.直线与圆的关系

16.线段与圆的关系

17.求多边形的核监视摄象机

18.求点集中不相交点对 railwai

\*//\*

寻找包含点集的最小矩形

原理：该矩形至少一条边与点集的凸壳的某条边共线

First take the convex hull of the points. Let the resulting convex

polygon be P. It has been known for some time that the minimum

area rectangle enclosing P must have one rectangle side flush with

(i.e., collinear with and overlapping) one edge of P. This geometric

fact was used by Godfried Toussaint to develop the "rotating calipers"

algorithm in a hard-to-find 1983 paper, "Solving Geometric Problems

with the Rotating Calipers" (Proc. IEEE MELECON). The algorithm

rotates a surrounding rectangle from one flush edge to the next,

keeping track of the minimum area for each edge. It achieves O(n)

time (after hull computation). See the "Rotating Calipers Homepage"

http://www.cs.mcgill.ca/~orm/rotcal.frame.html for a description

and applet.

\*//\*

折线的化简 伪码如下：

Input: tol = the approximation tolerance

L = {V0,V1,,Vn-1} is any n-vertex polyline

Set start = 0;

Set k = 0;

Set W0 = V0;

for each vertex Vi (i=1,n-1)

{

if Vi is within tol from Vstart

then ignore it, and continue with the next vertex

Vi is further than tol away from Vstart

so add it as a new vertex of the reduced polyline

Increment k++;

Set Wk = Vi;

Set start = i; as the new initial vertex

}

Output: W = {W0,W1,,Wk-1} = the k-vertex simplified polyline

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* \*

\* 补充 \*

\* \*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//两圆关系：

/\* 两圆：

相离： return 1；

外切： return 2；

相交： return 3；

内切： return 4；

内含： return 5；

\*/

int CircleRelation(POINT p1, double r1, POINT p2, double r2)

{

double d = sqrt( (p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y) );

if( fabs(d-r1-r2) < EP ) // 必须保证前两个if先被判定！

return 2;

if( fabs(d-fabs(r1-r2)) < EP )

return 4;

if( d > r1+r2 )

return 1;

if( d < fabs(r1-r2) )

return 5;

if( fabs(r1-r2) < d && d < r1+r2 )

return 3;

return 0; // indicate an error!

}

//判断圆是否在矩形内：

// 判定圆是否在矩形内，是就返回true（设矩形水平，且其四个顶点由左上开始按顺时针排列）

// 调用ptoldist函数，在第4页

bool CircleRecRelation(POINT pc, double r, POINT pr1, POINT pr2, POINT pr3, POINT pr4)

{

if( pr1.x < pc.x && pc.x < pr2.x && pr3.y < pc.y && pc.y < pr2.y )

{

LINESEG line1(pr1, pr2);

LINESEG line2(pr2, pr3);

LINESEG line3(pr3, pr4);

LINESEG line4(pr4, pr1);

if( r<ptoldist(pc,line1) && r<ptoldist(pc,line2) && r<ptoldist(pc,line3) && r<ptoldist(pc,line4) )

return true;

}

return false;

}

//点到平面的距离：

//点到平面的距离,平面用一般式表示ax+by+cz+d=0

double P2planeDist(double x, double y, double z, double a, double b, double c, double d)

{

return fabs(a\*x+b\*y+c\*z+d) / sqrt(a\*a+b\*b+c\*c);

}

//点是否在直线同侧：

//两个点是否在直线同侧，是则返回true

bool SameSide(POINT p1, POINT p2, LINE line)

{

return (line.a \* p1.x + line.b \* p1.y + line.c) \*

(line.a \* p2.x + line.b \* p2.y + line.c) > 0;

}

//镜面反射线：

// 已知入射线、镜面，求反射线。

// a1,b1,c1为镜面直线方程(a1 x + b1 y + c1 = 0 ,下同)系数;

//a2,b2,c2为入射光直线方程系数;

//a,b,c为反射光直线方程系数.

// 光是有方向的，使用时注意：入射光向量:<-b2,a2>；反射光向量:<b,-a>.

// 不要忘记结果中可能会有"negative zeros"

void reflect(double a1,double b1,double c1,double a2,double b2,double c2,double &a,double &b,double &c)

{

double n,m;

double tpb,tpa;

tpb=b1\*b2+a1\*a2;

tpa=a2\*b1-a1\*b2;

m=(tpb\*b1+tpa\*a1)/(b1\*b1+a1\*a1);

n=(tpa\*b1-tpb\*a1)/(b1\*b1+a1\*a1);

if(fabs(a1\*b2-a2\*b1)<1e-20)

{

a=a2;b=b2;c=c2;

return;

}

double xx,yy; //(xx,yy)是入射线与镜面的交点。

xx=(b1\*c2-b2\*c1)/(a1\*b2-a2\*b1);

yy=(a2\*c1-a1\*c2)/(a1\*b2-a2\*b1);

a=n;

b=-m;

c=m\*yy-xx\*n;

}

//矩形包含：

// 矩形2（C，D）是否在1（A，B）内

bool r2inr1(double A,double B,double C,double D)

{

double X,Y,L,K,DMax;

if (A < B)

{

double tmp = A;

A = B;

B = tmp;

}

if (C < D)

{

double tmp = C;

C = D;

D = tmp;

}

if (A > C && B > D) // trivial case

return true;

else

if (D >= B)

return false;

else

{

X = sqrt(A \* A + B \* B); // outer rectangle's diagonal

Y = sqrt(C \* C + D \* D); // inner rectangle's diagonal

if (Y < B) // check for marginal conditions

return true; // the inner rectangle can freely rotate inside

else

if (Y > X)

return false;

else

{

L = (B - sqrt(Y \* Y - A \* A)) / 2;

K = (A - sqrt(Y \* Y - B \* B)) / 2;

DMax = sqrt(L \* L + K \* K);

if (D >= DMax)

return false;

else

return true;

}

}

}

//两圆交点：

// 两圆已经相交（相切）

void c2point(POINT p1,double r1,POINT p2,double r2,POINT &rp1,POINT &rp2)

{

double a,b,r;

a=p2.x-p1.x;

b=p2.y-p1.y;

r=(a\*a+b\*b+r1\*r1-r2\*r2)/2;

if(a==0&&b!=0)

{

rp1.y=rp2.y=r/b;

rp1.x=sqrt(r1\*r1-rp1.y\*rp1.y);

rp2.x=-rp1.x;

}

else if(a!=0&&b==0)

{

rp1.x=rp2.x=r/a;

rp1.y=sqrt(r1\*r1-rp1.x\*rp2.x);

rp2.y=-rp1.y;

}

else if(a!=0&&b!=0)

{

double delta;

delta=b\*b\*r\*r-(a\*a+b\*b)\*(r\*r-r1\*r1\*a\*a);

rp1.y=(b\*r+sqrt(delta))/(a\*a+b\*b);

rp2.y=(b\*r-sqrt(delta))/(a\*a+b\*b);

rp1.x=(r-b\*rp1.y)/a;

rp2.x=(r-b\*rp2.y)/a;

}

rp1.x+=p1.x;

rp1.y+=p1.y;

rp2.x+=p1.x;

rp2.y+=p1.y;

}

//两圆公共面积：

// 必须保证相交

double c2area(POINT p1,double r1,POINT p2,double r2)

{

POINT rp1,rp2;

c2point(p1,r1,p2,r2,rp1,rp2);

if(r1>r2) //保证r2>r1

{

swap(p1,p2);

swap(r1,r2);

}

double a,b,rr;

a=p1.x-p2.x;

b=p1.y-p2.y;

rr=sqrt(a\*a+b\*b);

double dx1,dy1,dx2,dy2;

double sita1,sita2;

dx1=rp1.x-p1.x;

dy1=rp1.y-p1.y;

dx2=rp2.x-p1.x;

dy2=rp2.y-p1.y;

sita1=acos((dx1\*dx2+dy1\*dy2)/r1/r1);

dx1=rp1.x-p2.x;

dy1=rp1.y-p2.y;

dx2=rp2.x-p2.x;

dy2=rp2.y-p2.y;

sita2=acos((dx1\*dx2+dy1\*dy2)/r2/r2);

double s=0;

if(rr<r2)//相交弧为优弧

s=r1\*r1\*(PI-sita1/2+sin(sita1)/2)+r2\*r2\*(sita2-sin(sita2))/2;

else//相交弧为劣弧

s=(r1\*r1\*(sita1-sin(sita1))+r2\*r2\*(sita2-sin(sita2)))/2;

return s;

}

//圆和直线关系：

//0----相离 1----相切 2----相交

int clpoint(POINT p,double r,double a,double b,double c,POINT &rp1,POINT &rp2)

{

int res=0;

c=c+a\*p.x+b\*p.y;

double tmp;

if(a==0&&b!=0)

{

tmp=-c/b;

if(r\*r<tmp\*tmp)

res=0;

else if(r\*r==tmp\*tmp)

{

res=1;

rp1.y=tmp;

rp1.x=0;

}

else

{

res=2;

rp1.y=rp2.y=tmp;

rp1.x=sqrt(r\*r-tmp\*tmp);

rp2.x=-rp1.x;

}

}

else if(a!=0&&b==0)

{

tmp=-c/a;

if(r\*r<tmp\*tmp)

res=0;

else if(r\*r==tmp\*tmp)

{

res=1;

rp1.x=tmp;

rp1.y=0;

}

else

{

res=2;

rp1.x=rp2.x=tmp;

rp1.y=sqrt(r\*r-tmp\*tmp);

rp2.y=-rp1.y;

}

}

else if(a!=0&&b!=0)

{

double delta;

delta=b\*b\*c\*c-(a\*a+b\*b)\*(c\*c-a\*a\*r\*r);

if(delta<0)

res=0;

else if(delta==0)

{

res=1;

rp1.y=-b\*c/(a\*a+b\*b);

rp1.x=(-c-b\*rp1.y)/a;

}

else

{

res=2;

rp1.y=(-b\*c+sqrt(delta))/(a\*a+b\*b);

rp2.y=(-b\*c-sqrt(delta))/(a\*a+b\*b);

rp1.x=(-c-b\*rp1.y)/a;

rp2.x=(-c-b\*rp2.y)/a;

}

}

rp1.x+=p.x;

rp1.y+=p.y;

rp2.x+=p.x;

rp2.y+=p.y;

return res;

}

//内切圆：

void incircle(POINT p1,POINT p2,POINT p3,POINT &rp,double &r)

{

double dx31,dy31,dx21,dy21,d31,d21,a1,b1,c1;

dx31=p3.x-p1.x;

dy31=p3.y-p1.y;

dx21=p2.x-p1.x;

dy21=p2.y-p1.y;

d31=sqrt(dx31\*dx31+dy31\*dy31);

d21=sqrt(dx21\*dx21+dy21\*dy21);

a1=dx31\*d21-dx21\*d31;

b1=dy31\*d21-dy21\*d31;

c1=a1\*p1.x+b1\*p1.y;

double dx32,dy32,dx12,dy12,d32,d12,a2,b2,c2;

dx32=p3.x-p2.x;

dy32=p3.y-p2.y;

dx12=-dx21;

dy12=-dy21;

d32=sqrt(dx32\*dx32+dy32\*dy32);

d12=d21;

a2=dx12\*d32-dx32\*d12;

b2=dy12\*d32-dy32\*d12;

c2=a2\*p2.x+b2\*p2.y;

rp.x=(c1\*b2-c2\*b1)/(a1\*b2-a2\*b1);

rp.y=(c2\*a1-c1\*a2)/(a1\*b2-a2\*b1);

r=fabs(dy21\*rp.x-dx21\*rp.y+dx21\*p1.y-dy21\*p1.x)/d21;

}

//求切点：

// p---圆心坐标， r---圆半径， sp---圆外一点， rp1,rp2---切点坐标

void cutpoint(POINT p,double r,POINT sp,POINT &rp1,POINT &rp2)

{

POINT p2;

p2.x=(p.x+sp.x)/2;

p2.y=(p.y+sp.y)/2;

double dx2,dy2,r2;

dx2=p2.x-p.x;

dy2=p2.y-p.y;

r2=sqrt(dx2\*dx2+dy2\*dy2);

c2point(p,r,p2,r2,rp1,rp2);

}

//线段的左右旋：

/\* l2在l1的左/右方向（l1为基准线）

返回 0 ： 重合；

返回 1 ： 右旋；

返回 –1 ： 左旋；

\*/

int rotat(LINESEG l1,LINESEG l2)

{

double dx1,dx2,dy1,dy2;

dx1=l1.s.x-l1.e.x;

dy1=l1.s.y-l1.e.y;

dx2=l2.s.x-l2.e.x;

dy2=l2.s.y-l2.e.y;

double d;

d=dx1\*dy2-dx2\*dy1;

if(d==0)

return 0;

else if(d>0)

return -1;

else

return 1;

}

/\*

公式：

球坐标公式：

直角坐标为 P(x, y, z) 时，对应的球坐标是(rsinφcosθ, rsinφsinθ, rcosφ),其中φ是向量OP与Z轴的夹角，范围[0，π]；是OP在XOY面上的投影到X轴的旋角，范围[0，2π]

直线的一般方程转化成向量方程：

ax+by+c=0

x-x0 y-y0

------ = ------- // (x0,y0)为直线上一点，m,n为向量

m n

转换关系：

a=n；b=-m；c=m·y0-n·x0；

m=-b; n=a;

三点平面方程：

三点为P1，P2，P3

设向量 M1=P2-P1; M2=P3-P1;

平面法向量： M=M1 x M2 （）

平面方程： M.i(x-P1.x)+M.j(y-P1.y)+M.k(z-P1.z)=0