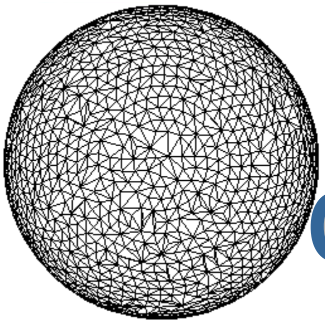

Geometry Computation

计算几何几何函数库

11.凸包 MelkMan 算法的实现
12. 凸多边形的直径
13.求凸多边形的重心
==
导引
```
/* 需要包含的头文件 */
#include <cmath >

/* 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;}
};
struct LINESEG
{
	POINT s;
	POINT e;
	LINESEG(POINT a, POINT b) { s=a; e=b;}
	LINESEG() { }
};
// 直线的解析方程 a*x+b*y+c=0  为统一表示，约定 a>= 0
struct LINE
{
    double a;
    double b;
    double c;
	LINE(double d1=1, double d2=-1, double d3=0) {a=d1; b=d2; c=d3;}
};
//线段树
struct LINETREE
{

}
//浮点误差的处理
int dblcmp(double d)
{
	if(fabs(d)<EP)
	    return 0 ;
	return (d>0) ?1 :-1 ;
}
```

<一>点的基本运算

```
// 返回两点之间欧氏距离
double dist(POINT p1,POINT p2)
{
    return( sqrt( (p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y) ) );
}
// 判断两个点是否重合
bool equal_point(POINT p1,POINT p2)
{
    return ( (abs(p1.x-p2.x)<EP)&&(abs(p1.y-p2.y)<EP) );
}

/*(sp-op)*(ep-op)的叉积
r=multiply(sp,ep,op),得到(sp-op)*(ep-op)的叉积
r>0:sp 在矢量 op ep 的顺时针方向;
r=0：op sp ep 三点共线;
r<0: sp 在矢量 op ep 的逆时针方向  */
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));
}

double amultiply(POINT sp,POINT ep,POINT op)
{
    return fabs((sp.x-op.x)*(ep.y-op.y)-(ep.x-op.x)*(sp.y-op.y));
}

/*矢量(p1-op)和(p2-op)的点积
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:   两矢量夹角为锐角;
r = 0：两矢量夹角为直角;
r > 0:   两矢量夹角为钝角  */
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 */
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+dey*dey)*(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;
}
```

<二>线段及直线的基本运算

```
/* 判断点 C 在线段 AB 所在的直线 l 上垂足 P 的与线段 AB 的关系
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 c,LINESEG l)
{
```

```
    LINESEG tl;
    tl.s=l.s;
    tl.e=c;
    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
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*/
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，注意反余弦函数的值域是从 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);
}
```

```
//判断线段 u 和 v 相交(包括相交在端点处)
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));
}

// 判断线段 u 和 v 相交（不包括双方的端点）
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)));
}

// 判断线段 v 所在直线与线段 u 相交
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)
//  注意：atan()返回的是 -PI/2 ~ PI/2
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);
	else return false;
}
```

<三> 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;
}
// 判断顶点是否按逆时针排列
// 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;
}

/*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(POINT q)
{
    int c=0,i,n;
    LINESEG l1,l2;

    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)%vcount];

        double ee= Polygon[(i+2)%vcount].x;
        double ss= Polygon[(i+3)%vcount].y;

        if(online(l2,q))
            return 1;
        if(intersect_A(l1,l2))
            c++;      // 相交且不在端点

         if(online(l1,l2.e)&& !online(l1,l2.s) && l2.e.y>l2.e.y)
            c++;//l2 的一个端点在 l1 上且该端点是两端点中纵坐标较大的那个
```

```
        if(!online(l1,l2.e)&& online(l1,l2.s) && l2.e.y<l2.e.y)
            c++;//忽略平行边
    }
    if(c%2 == 1)
        return 0;
    else
        return 2;
}

//判断点 q 在凸多边形 polygon 内
// 点 q 是凸多边形 polygon 内[包括边上]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)
{
    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 */
            return false；
    }
    return true；
}

/*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;
}
// 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;
}

/*所谓凸多边形的直径，即凸多边形任两个顶点的最大距离。下面的算法
仅耗时 O(n)，是一个优秀的算法。 输入必须是一个凸多边形，且顶点
必须按顺序（顺时针、逆时针均可）依次输入。若输入不是凸多边形
而是一般点集，则要先求其凸包。 就是先求出所有跖对，然后求出每
个跖对的距离，取最大者。点数要多于 5 个*/
void Diameter(POINT ch[],int n,double &dia)
{
   int znum=0,i,j,k=1;
   int zd[MAXV][2];
   double tmp;
   while(amultiply(ch[0],ch[k+1],ch[n-1]) > amultiply(ch[0],ch[k],ch[n-1])-EP)
      k++;
   i=0;
   j=k;
   while(i<=k && j<n)
   {
      zd[znum][0]=i;
     zd[znum++][1]=j;
     while(amultiply(ch[i+1],ch[j+1],ch[i]) > amultiply(ch[i+1],ch[j],ch[i]) – EP
            && j< n-1)
     {
         zd[znum][0]=i;
         zd[znum++][1]=j;
         j++;
      }
      i++;
   }
   dia=-1.0;
   for(i=0;i<znum;i++)
   {
       printf("%d %d\n",zd[i][0],zd[i][1]);
      tmp=dist(ch[zd[i][0]],ch[zd[i][1]]);
      if(dia<tmp)
         dia=tmp;
   }
}
```