Geometria Computacional

Wladimir Araújo Tavares [1]

[1]Universidade Federal do Ceará - Campus de Quixadá

31 de maio de 2017

Primitivas geométricas

Ponto : dois números (x, y)
Linha : três números a, b e c $[ax + by = c]$
Segmento : dois pontos
Polígono : uma sequência de pontos
Operações primitivas:

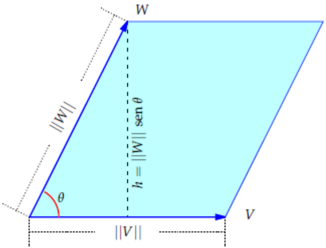
- A área do triângulo formado por três pontos.
- Dados três pontos p_1,p_2 e p_3, determinar se $p_1 - p_2 - p_3$ está no sentido antihorário.
- Determinar se dois segmentos se intersectam.
- Compara os ângulos entre dois segmentos.
- Calcular o perímetro e área de um polígono.
- Determinar se um polígono é convexo.
- Determinar se um ponto está no polígono

Produto Vetorial

- Possui diversas aplicações:
  - Determinar a area de um triângulo.
  - Testar se três pontos são colineares;
  - Determinar a orientação de três pontos (horário, anti-horário).
  - Testar se dois segmentos intersectam.

Produto Vetorial

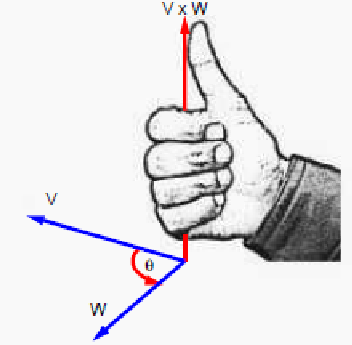
O comprimento de $V \times W$ é a área do paralelogramo definido por V e W.



$$||V \times W|| = ||V||\ ||W||\ sen(\theta) \tag{1}$$

Produto Vetorial

O sentido de $V \times W$ é tal que V, W e $V \times W$, nesta ordem, satisfazem a regra da mão direita.



Produto Vetorial

O produto vetorial 2D pode ser calculado da seguinte maneira:

$$(x_1, y_1) \times (x_2, y_2) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1 \tag{2}$$

O produto vetorial 3D pode ser calculado da seguinte maneira:

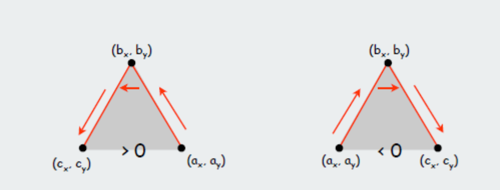
$$\begin{aligned} (x_1, y_1, z_1) \times (x_2, y_2, z_2) &= \begin{vmatrix} x_1 & x_2 & 1 \\ y_1 & y_2 & 1 \\ z_1 & z_2 & 1 \end{vmatrix} \\ &= x_1 y_2 + x_2 z_1 + y_1 z_2 - z_2 x_1 - z_1 y_2 - x_2 y_1 \end{aligned}$$

Área do Triângulo

- Dados três pontos A, B e C.

$$area(\Delta ABC) = \frac{1}{2}||\overrightarrow{AB} \times \overrightarrow{AC}|| \quad (3)$$

  - Se area $>$ 0 então A-B-C está no sentido anti-horário.
  - Se area $<$ 0 então A-B-C está no sentido horário.
  - Se area $==$ 0 então A-B-C são colineares.



Área do Triângulo

```
#include <math.h>
#include <stdio.h>
#include <algorithm>
#include <vector>
#include<iostream>
using namespace std;
const double EPSILON = 1.0e-7;
typedef struct Point{
  int x,y;
  Point(){};
  Point(int _x, int _y) : x(_x), y(_y){};
  Point operator +(const Point &that) const { return Point(x+that.x, y+tha
  Point operator -(const Point &that) const { return Point(x-that.x, y-tha
  Point& operator =(const Point &that){
    x = that.x; y = that.y;
    return *this;
  }
  friend ostream& operator << (ostream& os, const Point &p)
  {
      os << "x: " << p.x << "  y: " << p.y << endl;
      return os;
  }
} Point;
```

Área do Triângulo

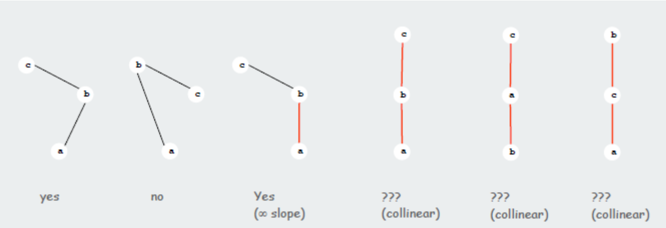
```
int cross_product(int x1, int y1, int x2, int y2){
  return x1*y2 - x2*y1;
}

double signed_area_triangulo(Point p0, Point p1, Point p2){
  int dx1 = p1.x - p0.x;
  int dy1 = p1.y - p0.y;
  int dx2 = p2.x - p0.x;
  int dy2 = p2.y - p0.y;
  return 0.5*cross_product(dx1,dy1,dx2,dy2);
}

double area_triangulo(Point p0, Point p1, Point p2){
  return fabs( signed_area_triangulo(p0,p1,p2) );
}
```

Direção

- Dado três pontos A, B e C
- Se $\overrightarrow{AB} \times \overrightarrow{AC} > 0$ então $\overrightarrow{AC}$ está à esquerda em relação a $\overrightarrow{AB}$ (sentido anti-horário).
- Se $\overrightarrow{AB} \times \overrightarrow{AC} < 0$ então $\overrightarrow{AC}$ está à direita em relação a $\overrightarrow{AB}$ (sentido horário).
- Se $\overrightarrow{AB} \times \overrightarrow{AC} == 0$ então os pontos A, B e C são colineares.



Direção

```
int DIRECTION(Point p0, Point p1, Point p2){
  int dx1 = p1.x - p0.x;
  int dy1 = p1.y - p0.y;
  int dx2 = p2.x - p0.x;
  int dy2 = p2.y - p0.y;

  int val = cross_product(dx1,dy1,dx2,dy2);
  if( val > 0) return 1;
  else if( val < 0) return -1;
  else return 0;
}
```

Checar se dois segmentos se cruzam

- Um segmento $\overrightarrow{p_1 p_2}$ intercepta um outro segmento se o ponto p_1 reside de um lado do segmento e o ponto p_2 reside do outro lado.
- Um caso limite surge se p_1 e p_2 residem sobre a linha (pontos colineares).
- Dois segmentos se cruzam se somente se:
  - Cada segmento intercepta a linha que contém o outro.
  - Uma extremidade de um segmento reside no outro segmento.

Determina se dois segmentos se cruzam

```
bool INTERSECT(Point p1, Point p2, Point p3, Point p4){
  int d1 = DIRECTION(p1,p2,p3);
  int d2 = DIRECTION(p1,p2,p4);
  int d3 = DIRECTION(p3,p4,p1);
  int d4 = DIRECTION(p3,p4,p2);

  if( d1*d2 < 0 && d3*d4 < 0) return true;
  else if(d1==0 && ON_SEGMENT(p1,p2,p3) ) return true;
  else if(d2==0 && ON_SEGMENT(p1,p2,p4) ) return true;
  else if(d3==0 && ON_SEGMENT(p3,p4,p1) ) return true;
  else if(d4==0 && ON_SEGMENT(p3,p4,p2) ) return true;
  else return false;
}
```

Determina se dois segmentos se cruzam

```
int ccw(Point p0, Point p1, Point p2){
  int dx1 = p1.x - p0.x;
  int dy1 = p1.y - p0.y;
  int dx2 = p2.x - p0.x;
  int dy2 = p2.y - p0.y;
  /*Comparando os angulos dy1/dx1 e dy2/dx2*/
  if( dx1*dy2 > dy1*dx2) return 1;
  if( dx1*dy2 < dy1*dx2) return -1;
  /*pontos colineares*/
  /*ponto p0 está entre p1 e p2*/
  if( (dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
  /*ponto p1 está mais próximo de p0
  está no sentido anti-horário*/
  if( dx1*dx1 + dy1*dy1 <  dx2*dx2 + dy2*dy2 ) return 1;
  return 0;
}
```

Determina se dois segmentos se cruzam

```
/*p1,p2,p3 e p4 nao podem ser colineares*/
bool intersect(Point p1, Point p2, Point p3, Point p4){
  return (ccw(p1,p2,p3)*ccw(p1,p2,p4) <=0)
         && (ccw(p3,p4,p1)*ccw(p3,p4,p2) <=0)
}
```

Ordenação polar

- Escreva um algoritmo $O(n \ log \ n)$ para ordenar uma sequência $< p_1, p_2, \ldots, p_n >$ de n pontos de acordo com seus ângulos polares com relação a um determinado ponto de origem p_0.

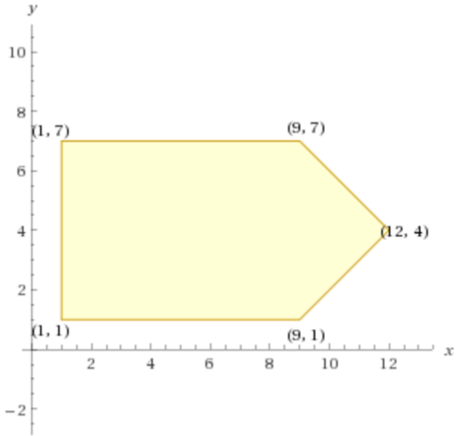
Determina se dois segmentos se cruzam

```
bool compare(const Point & p1, const Point & p2){
 if ( cross_product(p1.x,p1.y,p2.x,p2.y) < 0)
  return false;
 else
  return true;
}

void ordenacao_polar(vector <Point> & P){
  for(int i = 1; i < P.size(); i++){
    P[i] = P[i] - P[0];
  }
  sort(P.begin()+1, P.end(), compare);
  for(int i = 1; i < P.size(); i++){
    P[i] = P[i] + P[0];
  }
}
```

Polígono

Considere o polígono formado pelos pontos (1,1),(9,1),(12,4),(9,7) e (1,7).



Polígono

```
int main(){

  vector <Point> P;

  P.push_back( Point(1,1) );
  P.push_back( Point(1,7) );
  P.push_back( Point(9,1) );
  P.push_back( Point(9,7) );
  P.push_back( Point(12,4) );

  ordenacao_polar(P);

  for(int i = 0; i < P.size(); i++)
    cout << P[i] << endl;
}
```

Polígono

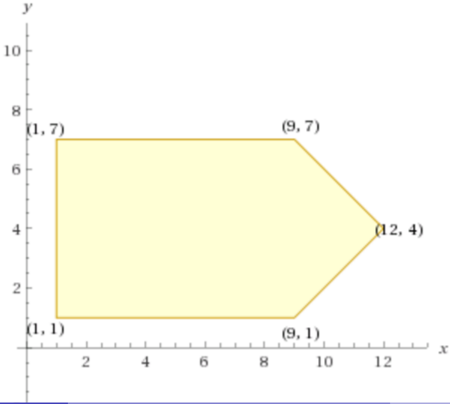
```
x: 1   y: 1

x: 9   y: 1

x: 12   y: 4

x: 9   y: 7

x: 1   y: 7
```



Perímetro

- O perímetro de um poligono (concâvo ou convexo) com n vértices dado alguma ordem dos vértices pode ser computada da seguinte maneira:

```
double dist(Point p1, Point p2){
  int dx = p1.x - p2.x;
  int dy = p1.y - p2.y;
  return hypot(dx,dy);
}
double perimetro(const vector <Point> & P){
  double result = 0.0;
  for(int i = 0; i < P.size(); i++){
    result += dist(P[i],P[(i+1)%P.size()]);
  }
  return result;
}
```

Propriedades

Properties:

edge lengths	$(8 \mid 3\sqrt{2} \mid 3\sqrt{2} \mid 8 \mid 6) \approx (8 \mid 4.24264 \mid 4.24264 \mid 8 \mid 6)$
diagonal lengths	$(\sqrt{130} \mid 10 \mid 6 \mid 10 \mid \sqrt{130}) \approx (11.4018 \mid 10 \mid 6 \mid 10 \mid 11.4018)$
area	57
perimeter	$22 + 6\sqrt{2} \approx 30.4853$
interior angles	$(90° \mid 135° \mid 90° \mid 135° \mid 90°) \approx$ (1.5708 radians | 2.35619 radians | 1.5708 radians | 2.35619 radians | 1.5708 radians)
interior angle sum	$540° = 3\pi$ rad ≈ 9.425 rad

Exemplo

```
int main(){
  vector <Point> P;
  P.push_back( Point(1,1) );
  P.push_back( Point(1,7) );
  P.push_back( Point(9,1) );
  P.push_back( Point(9,7) );
  P.push_back( Point(12,4) );
  ordenacao_polar(P);
  cout << "perimetro " << perimetro(P) << endl;
}
```

Saída:

```
perimetro 30.4853
```

Área

- A área com sinal de polígono de n vértices dado uma ordem dos vértices pode ser computada pela determinante da matriz abaixo:

$$
\begin{aligned}
A &= \frac{1}{2}\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{vmatrix} \\
&= \frac{1}{2}(x_0y_1 + x_1y_2 + \ldots + x_{n-1}y_0 - y_0x_1 - y_1x_2 - \ldots - y_{n-1}x_0)
\end{aligned}
$$

Área

```
double area(const vector <Point> &P){
  double result = 0.0;
  double x1,y1,x2,y2;
  for(int i = 0; i < P.size(); i++){
    x1 = P[i].x;
    y1 = P[i].y;
    x2 = P[(i+1)%P.size()].x;
    y2 = P[(i+1)%P.size()].y;
    result += (x1*y2 - x2*y1);
  }
  return fabs(result)/2.0;
}
```

Polígono Convexo

- Um polígono é convexo se todos três pontos consecutivos de um polígono estão na mesma direção (todo no sentido horário ou anti-horário). Se encontrarmos pelo menos uma tripla onde esta condição é falsa, então o polígono é côncavo.

Polígono Convexo

```
bool isConvex(const vector <Point> &P){
  int sz = (int) P.size();
  if(sz <= 2) return false;
  bool isLeft = DIRECTION(P[0],P[1],P[2]) > 0;
  for(int i = 1; i < sz-1; i++){
    if( (DIRECTION(P[i],P[i+1],P[(i+2)%sz]) > 0) != isLeft ){
      return false;
    }
  }
  return true;
}
```

Polígono Convexo

```
int main(){
  vector <Point> P;
  P.push_back( Point(1,1) );
  P.push_back( Point(9,1) );
  P.push_back( Point(12,4) );
  P.push_back( Point(9,7) );
  P.push_back( Point(1,7) );
  cout << "isConvex "   << isConvex(P) << endl; //1
  P.clear();
  P.push_back( Point(1,1) );
  P.push_back( Point(3,3) );
  P.push_back( Point(9,1) );
  P.push_back( Point(12,4) );
  P.push_back( Point(9,7) );
  P.push_back( Point(1,7) );
  cout << "isConvex "   << isConvex(P) << endl; //0
}
```

Ângulo entre três pontos

- Dados três pontos P_0, P_1 e $P2$:
- Seja $V = P_1 - P_0$ e $W = P_2 - P_0$
- $V \cdot W = |V||W|cos(\theta)$
- $|V \times W| = |V||W|sen(\theta)$
- $tan(\theta) = \frac{|V \times W|}{V \cdot W}$

Polígono Convexo

```
double dot(Point a, Point b){
  return a.x*b.x + a.y*b.y;
}
double cross(Point a, Point b){
  return a.x*b.y - a.y*b.x;
}
double angle(Point p1, Point p2, Point p3){
  Point v1 = p2-p1;
  Point v2 = p3-p1;
  return acos( dot(v1,v2)/(lenght(v1)*lenght(v2)));
}
double angle2(Point p1, Point p2, Point p3){
  Point v1 = p2-p1;
  Point v2 = p3-p1;
  return atan2( (double)cross(v1,v2), dot(v1,v2) );
}
```

Rotaciona um ponto por um ângulo θ

Um ponto pode ser rotacionado por um ângulo θ no sentido anti-horário usando a matriz de rotação

$$\left| \begin{array}{c} x' \\ y' \end{array} \right| = \left| \begin{array}{cc} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{array} \right| \times \left| \begin{array}{c} x \\ y \end{array} \right| \tag{4}$$

Polígono Convexo

```
double DEG2RAD(double theta){
  return (theta*PI)/180;
}
Point rotate(Point p, double theta){
  double rad = DEG2RAD(theta);
  return Point(p.x*cos(rad) - p.y*sin(rad),
               p.x*sin(rad) + p.y * cos(rad) );
}
```

Ponto no polígono

- O algoritmo winding number permite checar se um ponto *pt* está em um polígono convexo ou côncavo.
- O algoritmo calcula a soma dos ângulos entre três pontos {P[i],pt,P[i+1]} consecutivos de P.
- Se a soma final for igual 2π então *pt* está dentro do polígono;

Ponto no polígono

```
bool inPolygon(Point pt, const vector <Point> &P){
  int sz = (int)P.size();
  if( sz <= 2) return false;
  double sum = 0;

  for(int i=0; i < sz; i++){
    sum += fabs( angle(pt,P[i],P[(i+1)%sz]));
  }
  return fabs(sum - 2*PI) < EPSILON;
}
```

Ponto no polígono

```
int main(){
  vector <Point> P;
  P.push_back( Point(1,1) );
  P.push_back( Point(9,1) );
  P.push_back( Point(12,4) );
  P.push_back( Point(9,7) );
  P.push_back( Point(1,7) );
  Point pt(2,2);
  cout << "isConvex "   << isConvex(P) << endl; //1
  cout << "inPolygon " << inPolygon(pt,P) << endl;//1
  P.clear();
  P.push_back( Point(1,1) );
  P.push_back( Point(3,3) );
  P.push_back( Point(9,1) );
  P.push_back( Point(12,4) );
  P.push_back( Point(9,7) );
  P.push_back( Point(1,7) );
  cout << "isConvex "   << isConvex(P) << endl; //0
  cout << "inPolygon " << inPolygon(pt,P) << endl;//0
}
```

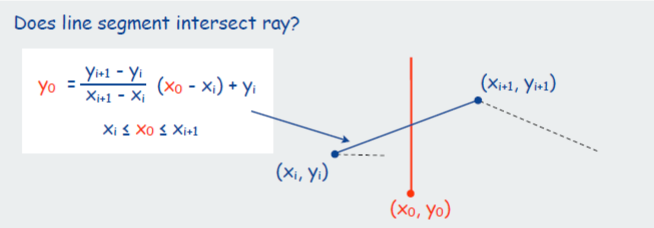
Ponto no polígono

- O que acontece no procedimento *inPolygon* se o ponto pt está em uma aresta do Polígono P ou se pt está no ponto médio entre P[i] e P[i+2]? Como essa situação pode ser corrigida?

Ponto no polígono

```
bool inPolygon2(Point pt, const vector <Point> &P){
  int sz = (int)P.size();
  if( sz <= 2) return false;
  Point extreme(1000000, pt.y);
  int count=0;
  for(int i = 0; i < sz; i++){
    if( INTERSECT(P[i], P[(i+1)%sz], pt, extreme) ){
      if( DIRECTION(P[i], P[(i+1)%sz],pt) == 0)
        return ON_SEGMENT(P[i], P[(i+1)%sz],pt);
      count++;
    }
  }
  return (count%2)==1;
}
```

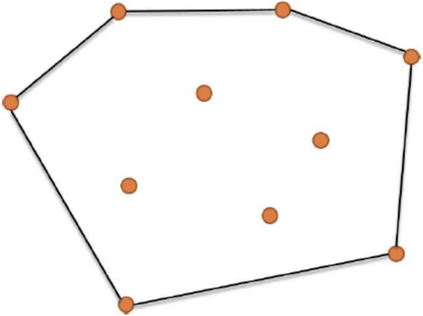
Ponto no polígono




```
bool inPolygon3(Point pt, vector<Point> P){
    int sz = P.size();
    int crossing = 0;
    for(int i = 0; i < sz; i++){
        if( P[(i+1)%sz].x == P[i].x) continue;
        double slope = (P[(i+1)%sz].y-P[i].y)/(P[(i+1)%sz].x-P[i].x);
        bool cond1 = (P[i].x <= pt.x) && (pt.x < P[(i+1)%sz].x);
        bool cond2 = (P[(i+1)%sz].x <= pt.x) && (pt.x < P[i].x);
        bool above = (pt.y < slope*(pt.x-P[i].x) +P[i].y);
        if( (cond1 || cond2) && above ) crossing++;
    }
    return ( (crossing % 2) != 0 );
}
```

Fecho convexo

- Dado n pontos em um plano, encontre o menor polígono convexo contendo todos os pontos.



Algoritmo Simples

- O segmento AB está no fecho convexo se somente se DIRECTION(A,B,C) tem o mesmo sinal para todos os pontos C
- Para cada ponto A e B:
  - Se DIRECTION(A,B,C) > 0 para todo $C \neq A, B$ então insira os pontos A,B

Algoritmo Graham Scan

- Sabemos que o ponto mais à esquerda está no fecho convexo.
- Coloque o ponto mais à esquerda como origem.
- Ordene os pontos usando o produto vetorial.
- Incrementalmente construa o fecho convexo usando uma pilha

Pseudocódigo

- Mantenha uma lista de pontos com a propriedade convexa
- Para cada ponto *i* faça:
  - Se o novo ponto faz um canto concavo remova o vértice que causa isso.
  - Repita até que a lista de pontos tenha a propriedade convexa.

Fecho Convexo

```
vector<Point> convex_hull(vector <Point> & P){
  int n = (int)P.size();
  if( n <= 2 ){
    if( P[0] == P[1]  ) P.pop_back();
    return P;
  }
  int pivot = 0;
  for(int i = 1; i < n; i++){
    if( P[i].y < P[pivot].y ||
      (P[i].y == P[pivot].y && P[i].x < P[pivot].x))
      pivot = i;
  swap(P[0],P[pivot]);
  ordenacao_polar(P);
  vector <Point> S;
  S.push_back(P[0]); S.push_back(P[1]);S.push_back(P[2]);
  int i = 3;
  while(i < n){
    int j = (int) S.size() - 1;
    if( DIRECTION(S[j-1],S[j],P[i])>0 )
        S.push_back(P[i++]);
    else S.pop_back();
  }
  return S;
}
```

Reta

```
typedef struct Line{
  double a,b,c;
  friend ostream& operator << (ostream& os, const Line &l)
  {
      os << l.a << "x␣+␣" << l.b << "␣y␣+␣" << l.c << "␣=␣0␣" << endl;
      return os;
  }
}Line;

void pointsToLine(PointD p1,PointD p2, Line &l){
  if( fabs(p1.x - p2.x) < EPSILON ){
    l.a = 1.0;
    l.b = 0.0;
    l.c = -p1.x;
  }else{
    l.a = -(double)(p1.y-p2.y)/(p1.x-p2.x);
    l.b = 1.0;
    l.c = -(double)(l.a*p1.x) - p1.y;
  }
}
```

Reta paralelas e coincidentes

```
bool areParallel(Line l1, Line l2){
  return ( fabs(l1.a-l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
}

bool areSame(Line l1, Line l2){
  return areParallel(l1,l2) && ( fabs(l1.c-l2.c) < EPSILON);
}

int main(){
  Line l1,l2,l3;
  pointsToLine( PointD(2,2), PointD(5,5), l1);
  cout << l1 << endl;
  pointsToLine( PointD(1,1), PointD(4,4), l2);
  cout << l2 << endl;
  pointsToLine( PointD(2,3), PointD(5,6), l3);
  cout << l3 << endl;
  if( areSame(l1,l2) )
    cout << "l1 e l2 are same" << endl;
  if( areParallel(l1,l3) )
    cout << "l1 e l3 are parallel" << endl;
}
```

Intersecção entre retas

```
bool areIntersect(Line l1, Line l2, PointD &pt){
  if( areParallel(l1,l2) ) return false;
  /*Resolver um sistema linear com 2 equações*/
  pt.x = (l2.b*l1.c - l1.b*l2.c)/(l2.a*l1.b - l1.a*l2.b);
  /* teste para linhas verticais*/
  if( fabs(l1.b) > EPSILON)
    pt.y = -(l1.a*pt.x + l1.c);
  else
    pt.y = -(l2.a*pt.x + l2.c);
  return true;
}

int main(){
  Line l1,l2,l3;
  pointsToLine( PointD(2,2), PointD(5,5), l1);
  cout << l1 << endl;
  pointsToLine( PointD(2,4), PointD(3,2), l2);
  cout << l2 << endl;
  PointD pt;
  if( areIntersect(l1,l2,pt) )
    cout << pt << endl;
}
```

Ponto mais próximo

```
void pointAndSlopeLine(PointD p, double m, Line &l){
  l.a = -m;
  l.b = 1.0;
  l.c = -(l.a*p.x + l.b*p.y);
}
void closesPoint(PointD p_in, Line l, PointD & p_c){
  Line perp;
  if( fabs(l.b) <= EPSILON){ // reta vertical
    p_c.x = -l.c;
    p_c.y = p_in.y;
    return;
  }
  if( fabs(l.a) <= EPSILON){// reta horizontal
    p_c.x = p_in.x;
    p_c.y = -l.c;
    return ;
  }
  pointAndSlopeLine(p_in, 1.0/l.a, perp);
  areIntersect(l, perp, p_c);
}
```