Universidade Federal do Ceará - MDCC
Geometria Computacional 2022
Prof. Joaquim Bento Cavalcante Neto
Aluno: Matheus Santos Araújo

Tarefa 2

Questões

Comando de execuções: Comando execução: g++ -o nomeprograma nomeprograma.cpp -static-libgcc

1 Pseudo-ângulo Orientado $\theta(a, b)$

Em vários casos e especificamente no caso da ordenação polar, precisamos comparar o valor dos ângulos e portando podemos substituir as funções trigonométricas inversas (e custosas) por uma função relacionada. Chamamos os valores dados por essa função de pseudo-ângulos.

1.1 a - Em termos de $\theta(a)$ e $\theta(b)$

As funções de pseudo-ângulo $\theta(a)$ e $\theta(b)$ é definida no quadrado unitário e toma valores no intervalo $[0, 8]$. Tal função pode ser implementada com 3 comparações, 1 soma e 1 divisão. O ângulo é substituído pelo comprimento do percurso nas arestas do quadrado partindo do ponto (1,0) tanto para o ponto definido a quanto b.

No algoritmo é verificado a qual quadrante o ponto pertence e se o vetor que vai do centro do quadrado até o ponto a ser classificado intercepta a aresta do quadrado antes ou depois do vértice pertencente ao mesmo quadrante do ponto. A partir dessas informações é possível saber, em unidades inteiras, parte do pseudo-ângulo, a outra parte é calculada por semelhança de triângulos usando-se apenas uma divisão.

```
// Algoritmo do cálculo do pseudoangulo
    int pseudoang(int x,int y){
if (y >= 0){
         if (x >= 0){
             if (x >= y){
                 return y/x;
             }
         return 2 - x/y;
         }

     if (-x <= y){
         return 2 + (-x)/y;
     }
     return 4 - y/(-x);
}

if (x < 0){
     if (-x >= -y){
```

```
        return 4 + (-y)/(-x);
    }
    return 6 - (-x)/(-y);
}

if (x <= -y){
    return 6 + x/(-y);
}
return 8 - (-y)/x;
}
```

Para saber os ângulos de a e b orientados ao plano passamos o ponto a e b na função anterior:

```
  // Ponto do vetor a
int x1 = 0;
int y1 = -1;

// pseudoangulo(a) - vetor a
start = std::clock();
int resultado1 = pseudoang(x1, y1);
std::cout << "Tempo do pseudoangulo a: " << (std::clock() - start) / (double)(CLOCK
std::cout << "pseudoangulo(a): " << resultado1 << "\n";
```

Saída:
Tempo do pseudoangulo a: 0 ms
pseduoangulo(a): 6

```
  // Ponto do vetor b
int x2 = 1;
int y2 = 0;

 // pseudoangulo(b) - vetor b
start2 = std::clock();
int resultado2 = pseudoang(x2, y2);
std::cout << "Tempo do pseudoangulo b: " << (std::clock() - start2) / (double)(CLOC
std::cout << "pseudoangulo(b): " << resultado2 << "\n";
```

Saída:
Tempo do pseudoangulo b: 1 ms
pseduoangulo(b): 0

Para mensurar a distância entre ambos realizamos a subração entre os ângulos de a e b no quadrado únitário, isto é, $a - b$. Isso é possivel devido a característica de função monótona do arco tomado sobre o círculo unitário.

```
   // pseudoangulo(a, b)
int resultado = resultado1 - resultado2;
```

Saída:
pseudoangulo(a,b): 6

1.2 b - a está à esquerda ou direita de b

Para a verificação de posição relativa utilizando pseudo-ângulos realizamos a verificação se a possui pseudoângulo orientado maior (se encontrando a direita) ou menor (se encontrando a esquerda) de b. A partir da orientação definida pseudoângulos maiores estarão a esquerda e menores a direita.

```
// a a direita ou a esquerda de b
if (resultado1 > resultado2){
     std::cout << "a está a esquerda de b";
} else {
     std::cout << "a está a direita de b";
}
```

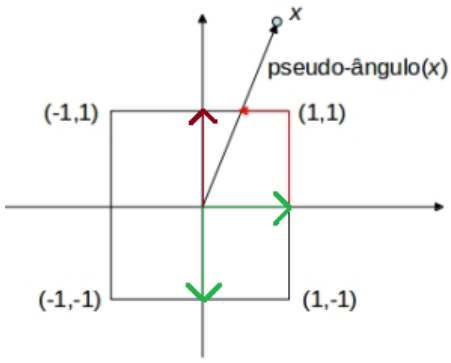
Saída:

a esta a esquerda de b

1.3 c - Ortogonalidade

Se a e b são ortogonais, isto é, geram dois vetores perpendiculares entre si, tal que a está à esquerda de b, podemos afirmar então que $\theta(a, b) = 6$. Isso se deve ao fato que dois vetores a e b conforme apresentando anteriormente o resultado do ângulo tomado pela medida tomada pelo arco entre a e b tal que consiste a no vetor da origem para $(0, -1)$ enquanto b no vetor da origem para $(1, 0)$ caracterizando dois vetores perpendiculares e fazem um ângulo de 90 graus (ortogonais) conforme a imagem 1.

Figura 1: É possível se tomarmos um vetor da origem para $(0, -1)$ e outro para $(1, 0)$ (ambos verdes) que obteremos dois vetores ortogonais com 6 unidades de distância e o vetor vinho $(0, 1)$ que se for a gera $\theta(a, b) = 6$, mas se for b gera $\theta(a, b) = 2$



A partir disso, tomamos também que a está à esquerda de b e temos uma distância no arco de 6 unidades ou faces entre ambos conforme pode ser visto na figura 1, podemo observar que o exemplo se estende para quaisquer vetores ortogonais, desde que a esteja a esquerda de b. Pois, podemos observar também que a outra possibilidade de vetor ortogonal a $0, 1$ surge se trocarmos o ponto final $a = (0, -1)$ para $a = (0, 1)$ e torná-lo inicial, na verdade a única outra possibilidade. Tomando o vetor gerado pelos pseudoangulo $0, 1$ e $1, 0$ temos que seu $\theta(a, b) = 2$ e não 6. Entretanto, nesse caso o primeiro vetor, no caso $a = 0, 1$ estaria a direita de $(b = 1, 0)$ e não a esquerda, logo não poderia ser avaliado (mas caso fosse b e não a seria válido e estaria a 6 unidades). Portanto, o ângulo a a esquerda de b o qual faz 90 graus gera sempre 6 faces/unidades de distância.

2 Outros Tipos de Pseudo-ângulos

2.1 Comparação

A primeira implementação é do pseudoangulo original descrito no livro e detalhada na questão anterior. Entretanto outras implementações foram desenvolvidas.

2.2 Pseudo Ângulo Original

De posse da primitiva ângulo orientado, o problema de ordenação polar é facilmente resolvido. Para cada vetor xi determinamos ângulo (xi) e ordenamos o conjunto de valores reais resultante utilizando Mergesort. Resulta daí um algoritmo O(n log n) para ordenação polar.

Saída do Pseudo Ângulo Original:

```
    ponto: 50,18
Tempo do pseudoangulo: 3 ms
pseudoangulo: 0
ponto: 42,6
Tempo do pseudoangulo: 0 ms
pseudoangulo: 0
ponto: 3,3
Tempo do pseudoangulo: 0 ms
pseudoangulo: 1
ponto: 9,44
Tempo do pseudoangulo: 1 ms
pseudoangulo: 2
ponto: 60,10
Tempo do pseudoangulo: 0 ms
pseudoangulo: 0
ponto: 80,27
Tempo do pseudoangulo: 1 ms
pseudoangulo: 0
ponto: 23,12
Tempo do pseudoangulo: 1 ms
pseudoangulo: 0
ponto: 12,89
Tempo do pseudoangulo: 0 ms
pseudoangulo: 2
ponto: 34,1
Tempo do pseudoangulo: 0 ms
pseudoangulo: 0
ponto: 5,45
Tempo do pseudoangulo: 1 ms
pseudoangulo: 2
Array ordenado dos angulos pseudo:
0 0 0 0 0 0 1 2 2 2
Tempo de ordenar do pseudoangulo: 9 ms
```

2.3 Atan2

A função Atan é uma que calcula o valor (em radianos) do ângulo cuja tangente é igual ao número especificado (o inverso da função tangente).

```
    //  Algoritmo do cálculo do pseudoangulo atan2
int atan2(int dx, int dy){
    int p = dx/(abs(dx)+abs(dy));
    if (dy < 0) {
        return p - 1;
    } else {
        return 1 - p;
    }
}
```

Saída do Atan2:

```
    ponto: 50,18
atan2: 1
Tempo do atan2: 0 ms
ponto: 42,6
atan2: 1
Tempo do atan2: 0 ms
ponto: 3,3
atan2: 1
Tempo do atan2: 0 ms
ponto: 9,44
atan2: 1
Tempo do atan2: 0 ms
ponto: 60,10
atan2: 1
Tempo do atan2: 15 ms
ponto: 80,27
atan2: 1
Tempo do atan2: 0 ms
ponto: 23,12
atan2: 1
Tempo do atan2: 0 ms
ponto: 12,89
atan2: 1
Tempo do atan2: 0 ms
ponto: 34,1
atan2: 1
Tempo do atan2: 0 ms
ponto: 5,45
atan2: 1
Tempo do atan2: 0 ms
Array ordenado do angulos atan2:
1 1 1 1 1 1 1 1 1 1
Tempo de ordenar do atan2: 0 ms
```

2.4 Acos

Podemos, por exemplo, utilizar a própria função cosseno ou, para manter o sentido das desigualdades, $f(\theta) = 1\breve{}cos\theta(0 \leq \theta \leq \pi)$, que define uma primitiva que podemos chamar de pseudo-ângulo. Ou utilizar o arco cosseno diretamente para obter o comprimento do arco correspondente no círculo unitário, orientado no sentido anti-horário e tomado a partir do eixo horizontal.

```
    int acosine(int a, int b){
    int ang = acos(a/sqrt(a^2 + b^2));

    if (b < 0){
       ang = ang * -1;
    }

    return ang;
}
```

Saída do Acos:

```
    ponto: 50,18
acosseno: -2147483648
Tempo do acosseno: 14 ms
ponto: 42,6
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 3,3
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 9,44
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 60,10
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 80,27
acosseno: -2147483648
Tempo do acosseno: 12 ms
ponto: 23,12
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 12,89
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 34,1
acosseno: -2147483648
Tempo do acosseno: 0 ms
ponto: 5,45
acosseno: 0
```

```
Tempo do acosseno: 0 ms
Array ordenado dos angulos acosine:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2
Tempo de ordenar do acosseno: 0 ms
```

2.5 Basic

A versão normalmente encontrada no Google. aqui chamada de básica, é uma função que retorna um valor entre 1 e 5. Quando usada como chave de classificação, produz o mesmo comportamento que a chave $mod(atan2(y, x) + 1, 5 * \pi, 2. * \pi)$. Portanto, não é um substituto direto para atan2() se é relvante saber quais ângulos são considerados os menores/maiores, no entanto, ele classifica na "direção"correta.

```
    // Algoritmo básico do pseudoangulo
int basic(int y, int x){
   int r = y / (abs(x) + abs(y));
    if (x < 0){
        return 2. - r;
     } else {
        return 4. + r;
     }
}
```

Saída do Basic:

```
    ponto: 50,18
basic: 4
Tempo do basic: 0 ms
ponto: 42,6
basic: 4
Tempo do basic: 16 ms
ponto: 3,3
basic: 4
Tempo do basic: 0 ms
ponto: 9,44
basic: 4
Tempo do basic: 0 ms
ponto: 60,10
basic: 4
Tempo do basic: 0 ms
ponto: 80,27
basic: 4
Tempo do basic: 0 ms
ponto: 23,12
basic: 4
Tempo do basic: 12 ms
ponto: 12,89
basic: 4
Tempo do basic: 0 ms
```

```
ponto: 34,1
basic: 4
Tempo do basic: 0 ms
ponto: 5,45
basic: 4
Tempo do basic: 0 ms
Array ordenado dos angulos basic:
4 4 4 4 4 4 4 4 4 4
Tempo de ordenar do basic: 0 ms
```

A comparação demonstrou que para alguns pontos atan2 e acos levam bastantes milisegundos para computar o pseudo ângulo enquanto em basic e no pseudo ângulo original os tempos tendem a ser inferiores.

3 Operações com Vetores

Soma:

```
    int sum(int a[], int b[], int n){

   int i;
   int *c;

    for (i = 0; i < n; i++)
      c[i] = a[i] + b[i];

    return *c;
}
```

Subtração:

```
int sub(int a[], int b[], int n)
    int i; int *c;
    for (i = 0; i < n; i++) c[i] = a[i] - b[i];
    return *c;
    Norma:

    double norm(int a[], int n){
  double accum = 0.;
    for (int i = 0; i < n; ++i) {
        accum += a[i] * a[i];
    }
    double norm = sqrt(accum);

    return norm;
}
```

Produto por escalar:

```
    int dotscalar(int a[], int b, int n){
    int p = 0;

    for (int i = 0; i < n; i++)
        p = p + a[i] * b;
    return p;
}
```

Produto escalar

```
    int dot(int a[], int b[], int n){
    int p = 0;

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

        p = p + a[i] * b[i];
    return p;
}
```

Similiaridade do cosseno:

```
    double cosinesimi(int a[], int b[], unsigned int n)
{
    double dot = 0.0;
     for(unsigned int i = 0u; i < n; ++i) {
        dot += a[i] * b[i] ;
    }
    return dot / (norm(a, n) * norm(b, n)) ;
}
```

Foram testados 1 caso de vetor aleatório e 3 casos patológicos sendo eles vetores colineares, vetores idênticos e vetor nulo. Saída:

```
Vetor a: 41 18467
Vetor b: 6334 26500
Soma: 6375 44967
Subtracao: -6293 -8033
Norma de a: 18467
Norma de b: 27246.5
produto por escalar(a,10): 185080
Produto  por escalar(b,10): 328340
Produto escalar: 489635194
Distancia entre a e b: 2
Similaridade do cos: 0.973117

Vetor a: 0 2
Vetor b: 2 2
Soma: 2 4
Subtracao: -2 0
```

```
Norma de a: 2
Norma de b: 2.82843
produto por escalar(a,10): 20
Produto  por escalar(b,10): 40
Produto escalar: 4
Distancia entre a e b: 2
Similaridade do cos: 0.707107

Vetor a: 2 2
Vetor b: 2 2
Soma: 4 4
Subtracao: 0 0
Norma de a: 2.82843
Norma de b: 2.82843
produto por escalar(a,10): 40
Produto  por escalar(b,10): 40
Produto escalar: 8
Distancia entre a e b: 2
Similaridade do cos: 1

Vetor a: 2 2
Vetor b: 0 0
Soma: 2 2
Subtracao: 2 2
Norma de a: 2.82843
Norma de b: 0
produto por escalar(a,10): 40
Produto  por escalar(b,10): 0
Produto escalar: 0
Distancia entre a e b: 2
Similaridade do cos: nan
```

4 Produto Vetorial, Interseção de Segmentos e Área Orientada

4.1 Produto Vetorial

O produto vetorial é uma operação sobre dois vetores em um espaço vetorial tridimensional e é denotado por $\times$. Dados dois vetores independentes linearmente a e b, o produto vetorial $a \times b$ é um vetor perpendicular ao vetor a e ao vetor b e é a normal do plano contendo os dois vetores.

Podemos usar esta propriedade para calcular o percurso no espaço 2D estendendo nossos pontos e vetores correspondentes para o caso 3D. Então, vamos definir vetores que correspondem à direção escolhida acima e estendê-los para o caso 3D:

Dependendo do valor da coordenada Z, os pontos originais foram percorridos no sentido anti-horário (se for negativo), no sentido horário (se for positivo) ou estão na mesma linha (se o valor for 0).

Figura 2: Dedução para obter o produto vetorial de 3 pontos a partir do sinal de z

$$\overrightarrow{PQ} = (Q_x - P_x, Q_y - P_y, 0),$$

$$\overrightarrow{QR} = (R_x - Q_x, R_y - Q_y, 0)$$

Then we calculate cross product of these vectors:

$$\vec{n} = \overrightarrow{PQ} \times \overrightarrow{QR} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ Q_x - P_x & Q_y - P_y & 0 \\ R_x - Q_x & R_y - Q_y & 0 \end{vmatrix} =$$

$$= -((Q_x - P_x) \cdot (R_y - Q_y) - (Q_y - P_y) \cdot (R_x - Q_x)) \cdot \vec{k} =$$

$$= ((Q_y - P_y) \cdot (R_x - Q_x) - (Q_x - P_x) \cdot (R_y - Q_y)) \cdot \vec{k}$$

É um fato bem conhecido que o produto vetorial de vetores pode ser usado para calcular sua orientação no espaço. A implementação do produto vetorial utilizou a dedução do cálculo do determinante considerando os segmentos $\overrightarrow{ab} = (b_x - a_x, b_y - a_y)$ e $\overrightarrow{bc} = (c_x - b_x, c_y - b_y)$.

```
      //Produto Vetorial  ------------------------------
  int cross(Point a, Point b, Point c){

    int p = (b.y-a.y)*(c.x-b.x) - (b.x-a.x)*(c.y-b.y);

    return p;
}
```

Teste:

```
    Point p1 = {0, 0}, p2 = {4, 4}, p3 = {1, 2};
    int c = cross(p1, p2, p3);
    cout << "Prod Vetorial: " << c << "\n";
```

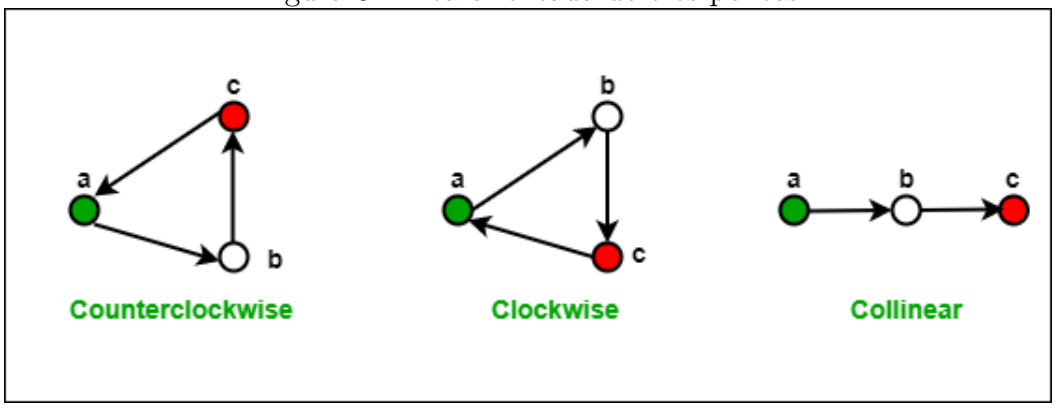
Saída:
Prod Vetorial: -4

4.2 Área Orientada

S é um escalar igual à área orientada de $p_1, p_2 ... p_n$. Isto é, $|S|$ é igual à área do polígono e S é positivo se e somente se $p_1, p_2 ... p_n$, nesta ordem, estão no sentido anti-horário. Para conhecer a orientação verificamos o resultado do produto vetorial e retornarmos o sentido colinear, anti-horario ou horario.

```
    // Área orientada ------------------------------

int orientation(Point a, Point b, Point c) {
   // Produto Vetorial
```

Figura 3: Área orientada de três pontos




```
   int val = cross(a, b, c);
   if (val == 0)
      return 0;     //colinear
   else if(val < 0)
      return 2;    // anti-horario
      return 1;    // horario
}

// Área orientada -----------------------------
```

Teste:

```
 Point p1 = {0, 0}, p2 = {4, 4}, p3 = {1, 2};
 int c = cross(p1, p2, p3);
 cout << "Prod Vetorial: " << c << "\n";
 int o = orientation(p1, p2, p3);
 if (o==0)          cout << "Linear" << "\n";
 else if (o == 1)   cout << "Horario" << "\n";
 else               cout << "Anti horario" << "\n";
```

Saída:
Prod Vetorial: -4
Anti horario

4.3 Interseção de Segmentos

Intersecção entre dois segmentos. Aqui o problema consiste em dizer se um segmento de reta intercepta outro segmento a partir da obtenção de suas orientações.

```
    // Interseção entre segmentos -----------------------------
struct Point {
   int x, y;
};

bool inLine(Point p, Point q, Point r){
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
```

```
            q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
    return false;
}

bool intersect(Point p1, Point q1, Point p2, Point q2){
   // Quatro orientações necessárias
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // Caso geral
    if (o1 != o2 && o3 != o4)
        return true;

    // p1, q1 e p2 são colineares e p2 está no segmento p1q1
    if (o1 == 0 && inLine(p1, p2, q1)) return true;

    // p1, q1 e p2 são colineares e q2 está no segmento p1q1
    if (o2 == 0 && inLine(p1, q2, q1)) return true;

    // p2, q2 e p1 são colineares e p1 está no segmento p2q2
    if (o3 == 0 && inLine(p2, p1, q2)) return true;

    // p2, q2 e q1 são colineares e q1 está no segmento p2q2
    if (o4 == 0 && inLine(p2, q1, q2)) return true;

    return false; // Nenhum dos casos
}
    // Interseção entre segmentos ------------------------------
```

Teste:

```
    struct Point s1 = {1, 1}, q1 = {10, 1};
    struct Point s2 = {1, 2}, q2 = {10, 2};
    intersect(s1, q1, s2, q2)? cout << "Sim\n": cout << "Nao\n";

    s1 = {10, 0}, q1 = {0, 10};
    s2 = {0, 0}, q2 = {10, 10};
    intersect(s1, q1, s2, q2)? cout << "Sim\n": cout << "Nao\n";

    s1 = {-5, -5}, q1 = {0, 0};
    s2 = {1, 1}, q2 = {10, 10};
    intersect(s1, q1, s2, q2)? cout << "Sim\n": cout << "Nao\n";
```

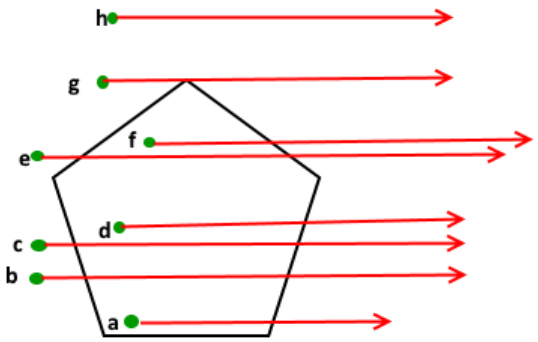
Saída:
Nao
Sim
Nao

5 Ponto em Polígono

5.1 Algoritmo do Tiro

Uma maneira simples de descobrir se o ponto está dentro ou fora de um polígono simples é testar quantas vezes um raio, partindo do ponto e indo em qualquer direção fixa, cruza as arestas do polígono. Se o ponto estiver do lado de fora do polígono, o raio cruzará sua borda um número par de vezes. Se o ponto estiver dentro do polígono, ele cruzará a aresta um número ímpar de vezes.

Figura 4: Algoritmo do Tiro



Implementação:

```
// Algoritmo do tiro ------------------
bool tiro(Point polygon[], int n, Point p) {

// Ao menos 3 vertices
if (n < 3) return false;

    //ponto da linha para o infinito no lado
Point extreme = {INF, p.y};

    // Contador de interseções e loop
int count = 0, i = 0;
do {
//proximo a cada 3
int next = (i+1)%n;
// verifica se a linha do ponto ao extremo faz interseção com a linha de polygon[i] a
if (intersect(polygon[i], polygon[next], p, extreme)){
// Se o ponto p for colinear com o segmento de linha next, então verifica se está na l
           // Se está, inLine retorna true, caso contrário false
if (orientation(polygon[i], p, polygon[next]) == 0)
return inLine(polygon[i], p, polygon[next]);

count++;
}
```

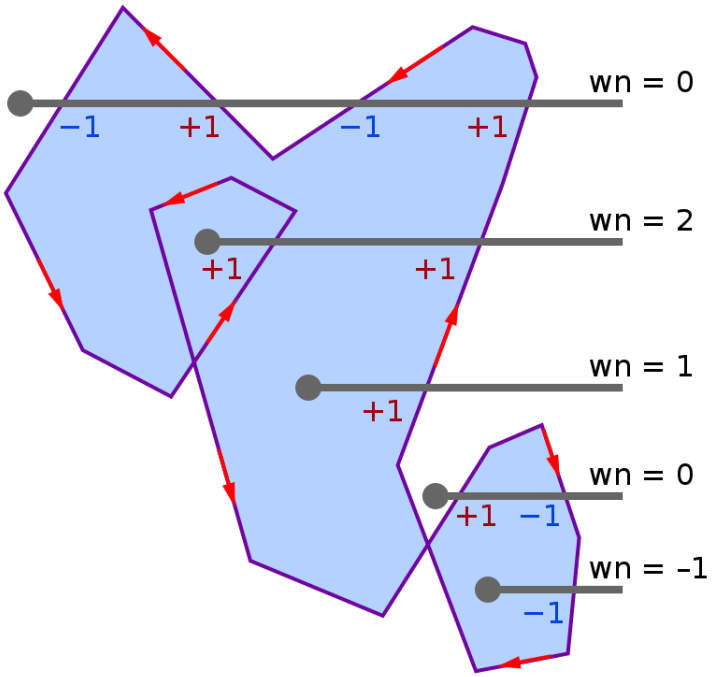
```
i = next;
} while (i != 0);

    // Retorna true se count for ímpar, false caso contrário
return count&1;
}
// Algoritmo do tiro ------------------
```

5.2 Algoritmo do Índice de Rotação

O número do índice de rotação é definido pelo número de vezes que uma curva viaja no sentido anti-horário em torno de um ponto. O algoritmo afirma que para qualquer ponto dentro do polígono o índice de rotação seria diferente de zero. Portanto, também é conhecido como algoritmo de regra diferente de zero.

Para qualquer polígono, encontramos todas as arestas do polígono que cortam a linha que passa pelo ponto de interesse e são paralelas ao eixo y. Para essas arestas verificamos se o ponto de consulta está no lado esquerdo ou direito da aresta ao olhar para todas as arestas no sentido anti-horário. Aumentamos o valor do índice de rotação (k) se o ponto de consulta estiver no lado esquerdo de um cruzamento para cima e diminuimos o k se o ponto de consulta estiver no lado direito de um cruzamento para baixo. Se o número final do índice de rotação for diferente de zero, o ponto estará dentro do polígono.

Figura 5: Algoritmo do índice de rotação



Implementação:

```
// Algoritmo do índice de rotação ------------------
int isLeft(const Point &a, const Point &b, const Point &point) {
    return ((b.x - a.x) * (point.y - a.y) -
```

```
            (point.x - a.x) * (b.y - a.y));
}

bool rot(const Point points_list [], int n, const Point &point) {
    // contador do indice
    int winding_number = 0;
int i;

    for (i = 0; i < n; ++i) {
        Point point1(points_list[i]);
        Point point2;

        if (i == (n - 1)) {
            point2 = points_list[0];
        } else {
            point2 = points_list[i + 1];
        }

        if (point1.y <= point.y) {
            if (point2.y > point.y) {
                if (isLeft(point1, point2, point) > 0) {
                    ++winding_number;
                }
            }
        } else {
            if (point2.y <= point.y) {
                if (isLeft(point1, point2, point) < 0) {
                    --winding_number;
                }
            }
        }
    }

    return winding_number;
}
// Algoritmo do índice de rotação ------------------
```

Foram considerados para testes do algoritmo do tiro e do algoritmo do índice de rotação os casos patológicos onde a linha cruza uma aresta do polígono, cruza a "quina"ponto na fronteira do polígono e um caso com polígono côncavo.

```
// Caso  patológico - ponto de vértice
Point polygon1[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
int n = sizeof(polygon1)/sizeof(polygon1[0]);
Point p = {0, 0};
tiro(polygon1, n, p)? cout << "Tiro: " << "Sim \n":  cout << "Tiro: " << "Nao \n";
rot(polygon1, n, p) != 0? cout << "indice: " << "Sim\n \n": cout << "indice: " << "Na

    // Caso  patológico - Ponto na fronteira
```

```
p = {0, 5};
tiro(polygon1, n, p)? cout << "Tiro: " << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon1, n, p) != 0? cout << "indice: " << "Sim \n\n": cout << "indice: " << "Nao

    // Caso  patológico - linha passando na quina
Point polygon2[] = {{0, 0}, {5, 5}, {5, 0}};
p = {0, 5};
n = sizeof(polygon2)/sizeof(polygon2[0]);
tiro(polygon2, n, p)? cout << "Tiro: " << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon2, n, p) != 0? cout << "indice: " << "Sim \n\n": cout << "indice: " << "Nao

// Pontos fora
p = {7, 1};
tiro(polygon2, n, p)? cout << "Tiro: " << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon2, n, p) != 0? cout << "indice: " << "Sim \n\n": cout << "indice: " << "Nao

    // Ponto fora
p = {8, 1};
tiro(polygon2, n, p)? cout << "Tiro: " << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon2, n, p) != 0? cout << "indice: " << "Sim \n\n": cout << "indice: " << "Nao

    // Caso  patológico - Polígono concavo
Point polygon3[] = {{0, 0}, {10, 0}, {5,5}, {10, 10}, {0, 10}};

// Ponto na quina
p = {0,5};
n = sizeof(polygon3)/sizeof(polygon3[0]);
tiro(polygon3, n, p)? cout << "Tiro:" << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon3, n, p) != 0? cout << "Indice:" << "Sim \n\n": cout << "indice: " << "Nao

    // Ponto dentro
p = {4, 3};
n = sizeof(polygon3)/sizeof(polygon3[0]);
tiro(polygon3, n, p)? cout << "Tiro:" << "Sim \n": cout << "Tiro: " << "Nao \n";
rot(polygon3, n, p) != 0? cout << "Indice:" << "Sim \n\n": cout << "indice: " << "Nao
```

Saída:

```
Tiro: Sim
indice: Sim

Tiro: Sim
indice: Sim

Tiro: Nao
indice: Nao

Tiro: Nao
indice: Nao
```

```
Tiro: Nao
indice: Nao

Tiro:Sim
Indice:Sim

Tiro:Sim
Indice:Sim
```

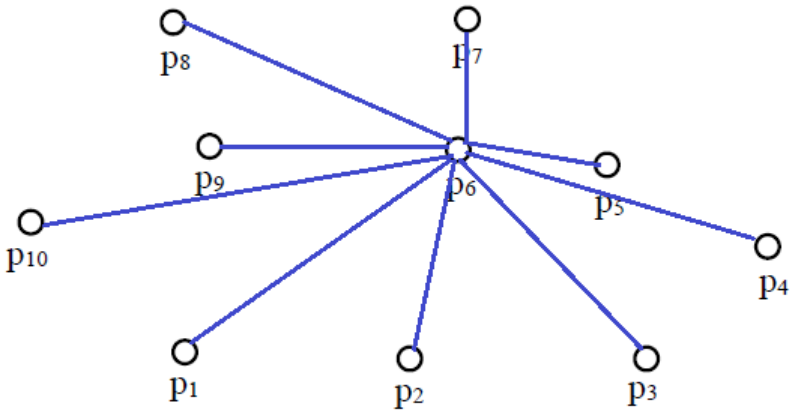
Os algoritmos obtiveram resultados iguais para os pontos testados. Retornando a resposta certa em casos como ponto estando no vértice, na fronteira, na quina e fora e colinear a uma quina e dentro de um polígono côncavo.

6 Polígono Dado Pelos Pontos

6.1 a - Polígono Estrelado

Um polígono P tem formato de estrela quando existe um ponto p no interior de P tal que qualquer outro ponto (vértice) na fronteira seja visível para p. Isto é, um polígono P tem formato de estrela se existe um ponto z tal que para cada ponto p de P o segmento zp esteja inteiramente dentro de P. O conjunto de todos os pontos z com esta propriedade é chamado de kernel de P.

Figura 6: Polígono estrelado dos pontos dados, já ordenados com $z = p_6$



6.2 b - Se o ponto no centroide está dentro do polígono

Ao testar com os algoritmos do tiro e do índice de rotação, ambos detalhados matematicamente anteriormente obtemos o resultado do ponto $z = p_6$ estar dentro do polígono, pois cruza a aresta um número ímpar de vezes. O índice de rotação também confirma o resultado de ponto interior, pois o resultado é diferente de zero.

Figura 7: Se o centroide p_6 está dentro do polígono pelo algoritmo do tiro

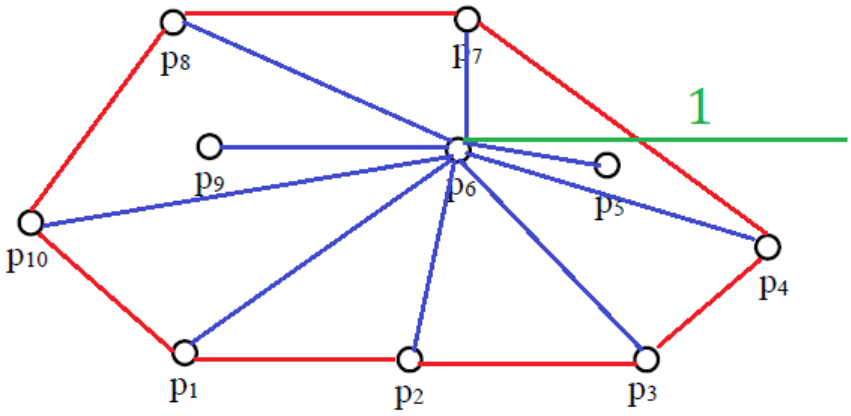
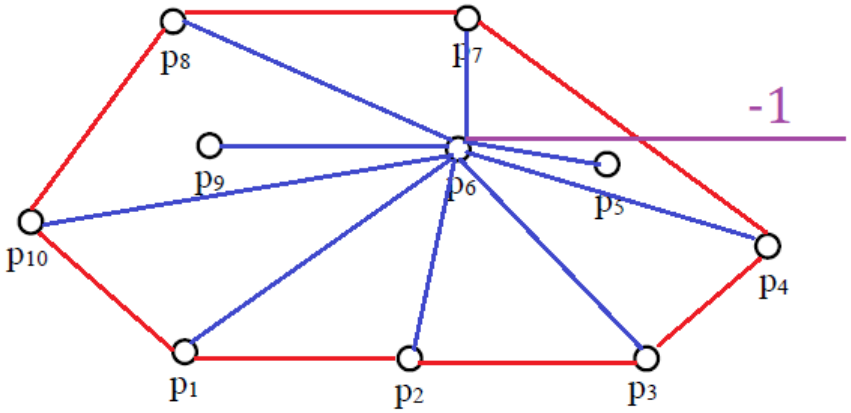


Figura 8: Se o centroide p_6 está dentro do polígono pelo algoritmo do índice de rotação



6.3 c - Complexidade de cada um dos algoritmos

Testar se um polígono tem formato de estrela e encontrar um único ponto no kernel pode ser resolvido em tempo linear formulando o problema como um programa linear e aplicando técnicas de programação linear de baixa dimensão. Cada aresta de um polígono define um semiplano interior, o semiplano cuja fronteira está na linha que contém a aresta e que contém os pontos do polígono em uma vizinhança de qualquer ponto interior da aresta. O núcleo de um polígono é a interseção de todos os seus semiplanos internos. A interseção de um conjunto arbitrário de N semiplanos pode ser encontrada em tempo $\theta(NlogN)$ usando a abordagem de dividir e conquistar.

O algoritmo do tiro tem complexidade de tempo $O(n)$ onde n é o número de vértices no polígono dado. A complexidade de tempo do índice de rotação é $O(n)$ semelhante ao algoritmo de tiro.