Geometria computazionale

Alessandro Bortolin

~~Volterra~~ Online, 9 febbraio 2024

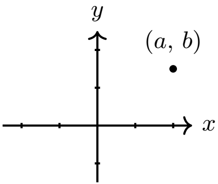
Punti

Punti

Cos'è un punto?

Punti

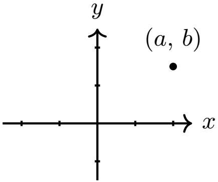
Cos'è un punto?



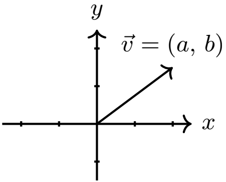
Una coppia di coordinate?

Punti

Cos'è un punto?



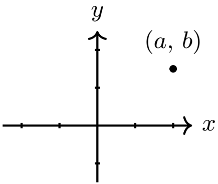
Una coppia di coordinate?



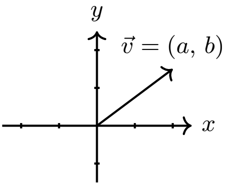
Un vettore?

Punti

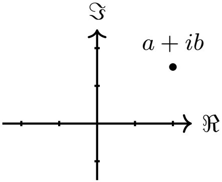
Cos'è un punto?



Una coppia di coordinate?



Un vettore?



Un numero complesso?

Rappresentazione dei punti

Come rappresentare un punto?

Rappresentazione dei punti

Come rappresentare un punto?

Rappresentazione mediante struct

```
struct pt {
    ll x, y;
    pt operator+(pt p) { ... }
    pt operator-(pt p) { ... }
    pt operator*(ll d) { ... }
    pt operator/(ll d) { ... }
    bool operator==(pt p) { ... }
    bool operator!=(pt p) { ... }
};
```

Rappresentazione dei punti

Come rappresentare un punto?

Rappresentazione mediante struct

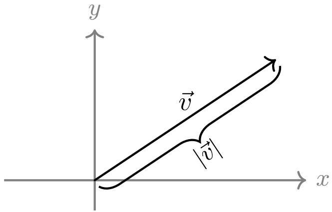
```
struct pt {
    ll x, y;
    pt operator+(pt p) { ... }
    pt operator-(pt p) { ... }
    pt operator*(ll d) { ... }
    pt operator/(ll d) { ... }
    bool operator==(pt p) { ... }
    bool operator!=(pt p) { ... }
};
```

Rappresentazione mediante numeri complessi per persone prigre

```
#include <complex>
typedef std::complex<ll> pt;
#define x real()
#define y imag()
```

Norma di un vettore

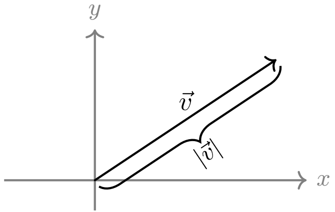
Implementazione



Norma di un vettore

Implementazione

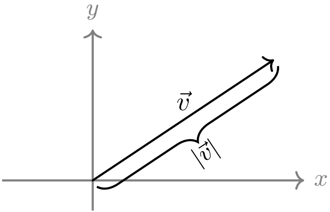
```cpp
ll norm(pt a) {
    return a.x * a.x + a.y * a.y;
}

double abs(pt a) {
    return std::sqrt(norm(a));
}
// oppure
double abs(pt a) {
    // Previene l'overflow
    return std::hypot(a.x, a.y);
}
```



Norma di un vettore

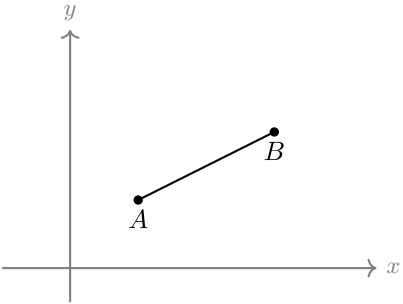
Implementazione

```
ll norm(pt a) {
    return a.x * a.x + a.y * a.y;
}

double abs(pt a) {
    return std::sqrt(norm(a));
}
// oppure
double abs(pt a) {
    // Previene l'overflow
    return std::hypot(a.x, a.y);
}

pt unit(pt a) {
    if (a == pt{0, 0}) return a;
    return a / abs(a); // Richiede i float
}
```



Distanza tra punti

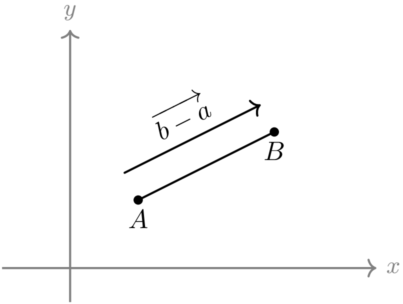
Implementazione



Distanza tra punti

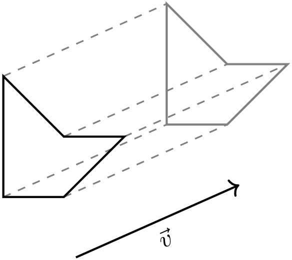
Implementazione

```
double dist(pt a, pt b) {
    return abs(b - a);
}
```



Traslazione

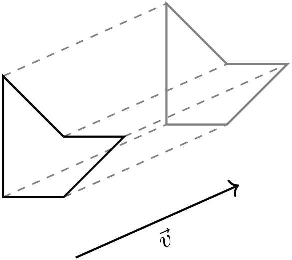
Implementazione



Traslazione

Implementazione

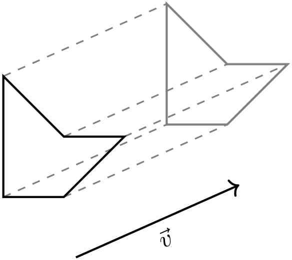
```
pt translate(pt p, pt v) {
    return p + v;
}
```



Traslazione

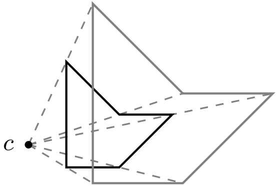
Implementazione

```
pt translate(pt p, pt v) {
    return p + v;
}
pt translate(pt p, pt dir, double dist) {
    return p + unit(dir) * dist;
}
```



Ridimensionare rispetto a un punto (omotetia)

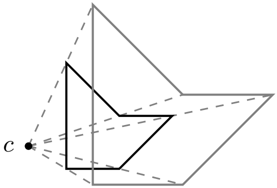
Implementazione



Ridimensionare rispetto a un punto (omotetia)

Implementazione

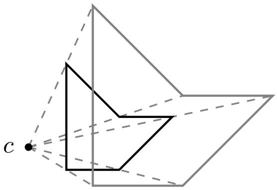
```
pt scale(pt p, ll factor, pt c) {
    return c + (p - c) * factor;
}
```

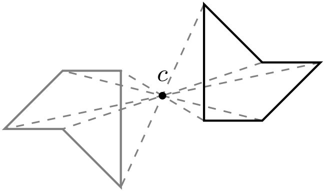


Ridimensionare rispetto a un punto (omotetia)

Implementazione

```
pt scale(pt p, ll factor, pt c) {
    return c + (p - c) * factor;
}
```

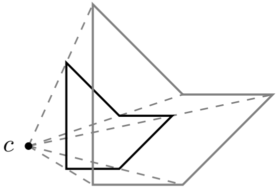


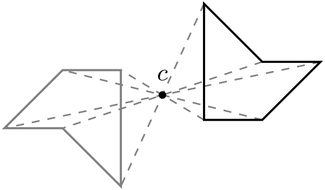


Ridimensionare rispetto a un punto (omotetia)

Implementazione

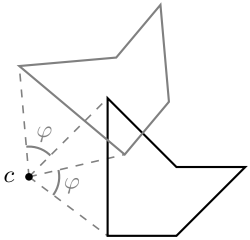
```
pt scale(pt p, ll factor, pt c) {
    return c + (p - c) * factor;
}
pt reflect(pt p, pt c) {
    return scale(p, -1, c);
}
```





Rotazione

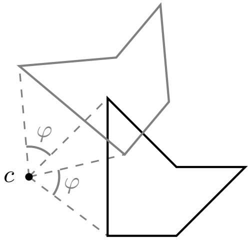
Implementazione



Rotazione

Implementazione

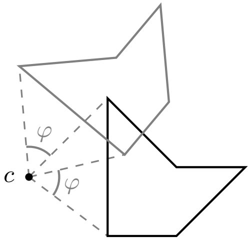
```cpp
pt rotate(pt p, double a) {
    // Richiede i float
    return {
        p.x * std::cos(a) - p.y * std::sin(a),
        p.x * std::sin(a) + p.y * std::cos(a)
    };
}
pt rotate(pt p, double a, pt c) {
    return c + rotate(p - c, a);
}
```



Rotazione

Implementazione

```
pt rotate(pt p, double a) {
    // Richiede i float
    return {
        p.x * std::cos(a) - p.y * std::sin(a),
        p.x * std::sin(a) + p.y * std::cos(a)
    };
}
pt rotate(pt p, double a, pt c) {
    return c + rotate(p - c, a);
}
pt perp(pt p) {
    return {-p.y, p.x};
}
```



Trasformazioni lineari

Trasformazioni lineari

Traslazione

$$\begin{bmatrix}1 & 0 & dx\\ 0 & 1 & dy\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ 1\end{bmatrix}=\begin{bmatrix}x+dx\\ y+dy\\ 1\end{bmatrix}$$

Trasformazioni lineari

Traslazione

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix}$$

Ridimensionare

$$\begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} cx \\ cy \\ 1 \end{bmatrix}$$

Trasformazioni lineari

Traslazione

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix}$$

Ridimensionare

$$\begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} cx \\ cy \\ 1 \end{bmatrix}$$

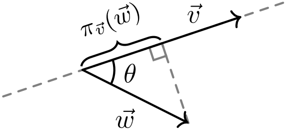
Rotazione

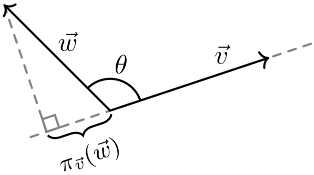
$$\begin{bmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos(a) - y\sin(a) \\ x\sin(a) + y\cos(a) \\ 1 \end{bmatrix}$$

Angoli

Dot product (prodotto scalare)

Implementazione



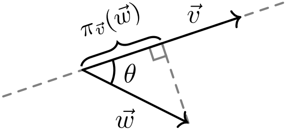


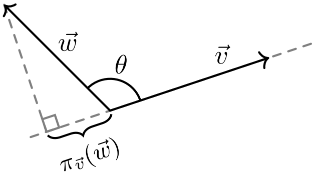
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Implementazione

```
ll dot(pt v, pt w) {
    return v.x * w.x + v.y * w.y;
}
```

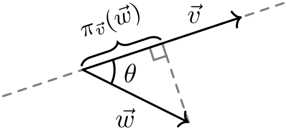


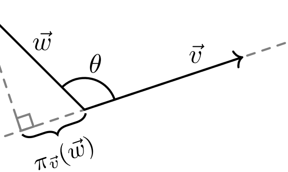


$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà



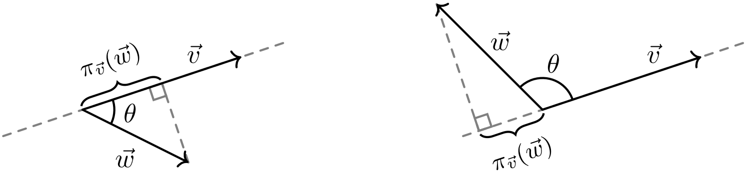


$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà

- $\vec{v} \cdot \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ sono perpendicolari: $\theta = \frac{\pi}{2}$.

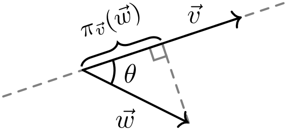


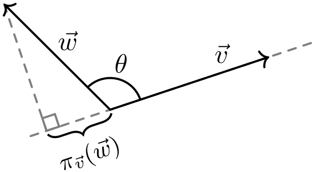
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà

- $\vec{v} \cdot \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ sono perpendicolari: $\theta = \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} > 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è acuto: $\theta < \frac{\pi}{2}$.



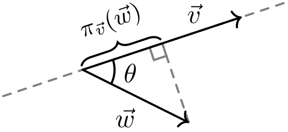


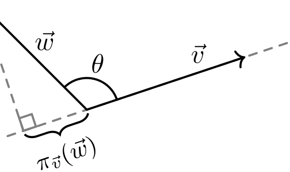
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà

- $\vec{v} \cdot \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ sono perpendicolari: $\theta = \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} > 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è acuto: $\theta < \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} < 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è ottuso: $\theta > \frac{\pi}{2}$.



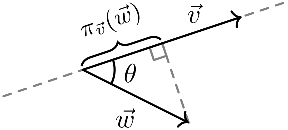


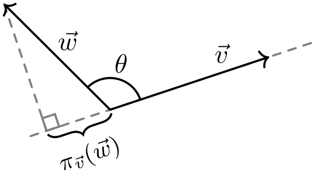
$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}| \pi_{\vec{v}}(\vec{w})$$

Dot product (prodotto scalare)

Proprietà

- $\vec{v} \cdot \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ sono perpendicolari: $\theta = \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} > 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è acuto: $\theta < \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} < 0$ se l'angolo tra $\vec{v}$ e $\vec{w}$ è ottuso: $\theta > \frac{\pi}{2}$.
- $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$.

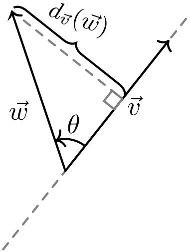


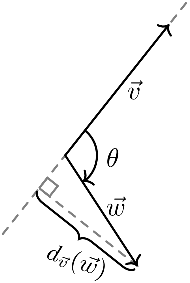


$$\vec{v} \cdot \vec{w} = |\vec{v}| \cdot |\vec{w}| \cos\theta = |\vec{v}|\pi_{\vec{v}}(\vec{w})$$

Cross product (prodotto vettoriale)

Implementazione



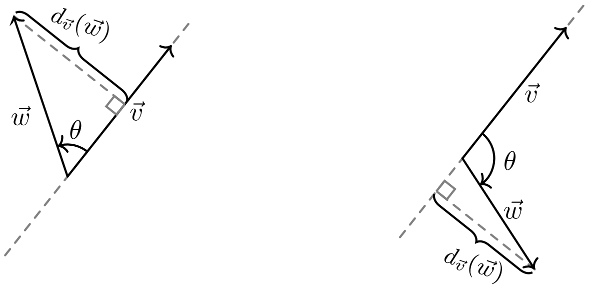


$$\vec{v} \times \vec{w} = |\vec{v}| \cdot |\vec{w}| \sin\theta = |\vec{v}| d_{\vec{v}}(\vec{w})$$

Cross product (prodotto vettoriale)

Implementazione

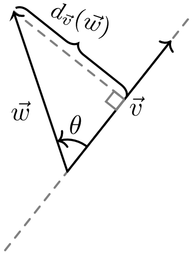
```
ll cross(pt v, pt w) {
    return v.x * w.y - v.y * w.x;
}
```

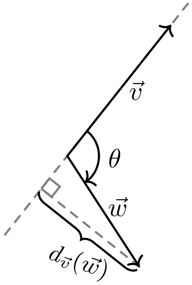


$$\vec{v} \times \vec{w} = |\vec{v}| \cdot |\vec{w}| \sin\theta = |\vec{v}| d_{\vec{v}}(\vec{w})$$

Cross product (prodotto vettoriale)

Proprietà

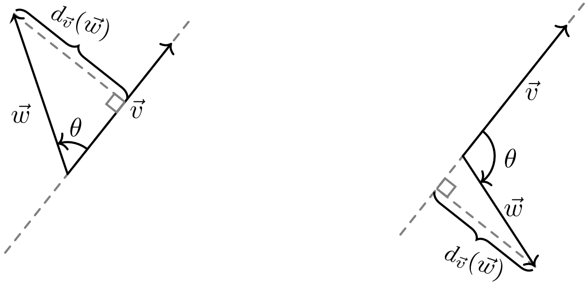




Cross product (prodotto vettoriale)

Proprietà

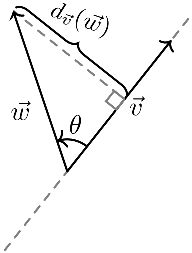
- $\vec{v} \times \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ paralleli.

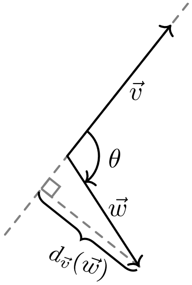


Cross product (prodotto vettoriale)

Proprietà

- $\vec{v} \times \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ paralleli.
- $\vec{v} \times \vec{w} > 0$ se $\vec{w}$ è a "sinistra" di $\vec{v}$.

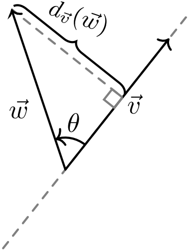


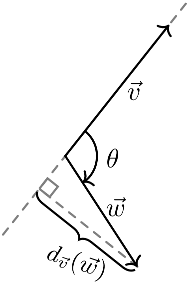


Cross product (prodotto vettoriale)

Proprietà

- $\vec{v} \times \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ paralleli.
- $\vec{v} \times \vec{w} > 0$ se $\vec{w}$ è a "sinistra" di $\vec{v}$.
- $\vec{v} \times \vec{w} < 0$ se $\vec{w}$ è a "destra" di $\vec{v}$.

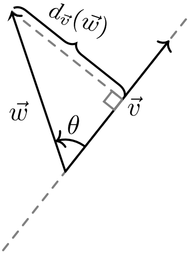


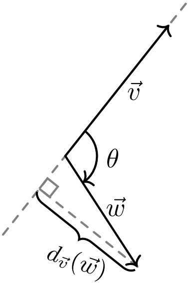


Cross product (prodotto vettoriale)

Proprietà

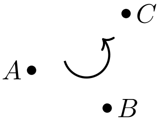
- $\vec{v} \times \vec{w} = 0$ se $\vec{v}$ e $\vec{w}$ paralleli.
- $\vec{v} \times \vec{w} > 0$ se $\vec{w}$ è a “sinistra” di $\vec{v}$.
- $\vec{v} \times \vec{w} < 0$ se $\vec{w}$ è a “destra” di $\vec{v}$.
- $\vec{v} \times \vec{w} = -\vec{w} \times \vec{v}$, l'ordine è importante!



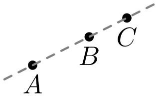


Orientare i punti

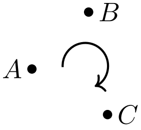
Implementazione



Senso antiorario.



Punti allineati.

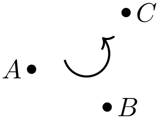


Senso orario.

Orientare i punti

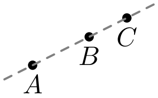
Implementazione

```
ll orient(pt a, pt b, pt c) {
    return cross(b - a, c - a);
}
```



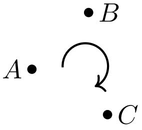
Senso antiorario.

`orient(A, B, C) > 0`



Punti allineati.

`orient(A, B, C) = 0`

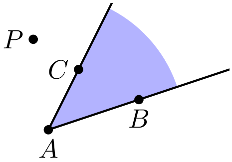


Senso orario.

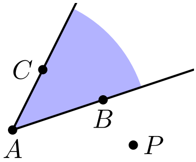
`orient(A, B, C) < 0`

Punti interni a un angolo

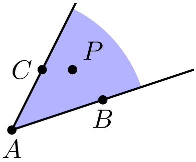
Implementazione



NO



NO

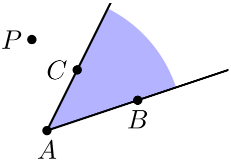


SÍ

Punti interni a un angolo

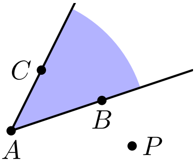
Implementazione

```
bool inAngle(pt a, pt b, pt c, pt p) {
    return orient(a, b, p) >= 0 && orient(a, c, p) <= 0;
}
```



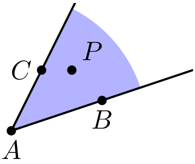
NO

`orient(A, B, P)` ≥ 0
`orient(A, C, P)` > 0



NO

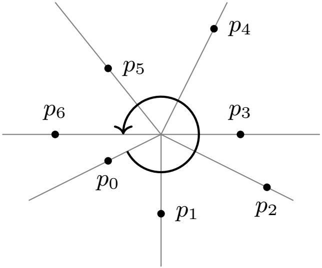
`orient(A, B, P)` < 0
`orient(A, C, P)` ≤ 0



SÍ

`orient(A, B, P)` ≥ 0
`orient(A, C, P)` ≤ 0

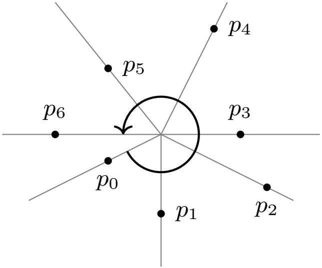
Polar sort



Polar sort

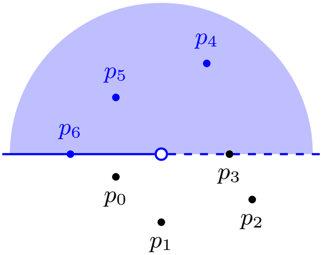
Una versione semplice

```
void polarSort(vector<pt>& P) {
    sort(P.begin(), P.end(), [](pt v, pt w) {
        return atan2l(v.y, v.x) < atan2l(w.y, w.x);
    });
}
```



Polar sort

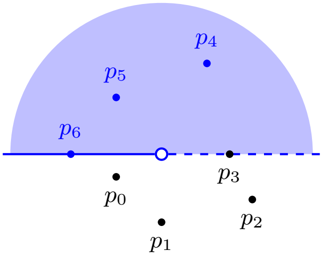
Una versione migliore



Polar sort

Una versione migliore

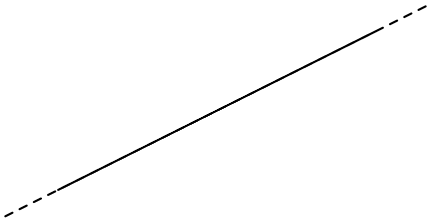
```
bool half(pt p) {
    return p.y > 0 || (p.y == 0 && p.x < 0);
}
void polarSort(vector<pt>& P) {
    sort(P.begin(), P.end(), [](pt v, pt w) {
        return pair(half(v), 0ll) < pair(half(w), cross(v,w));
    });
}
```



Rette e segmenti

Rappresentazione delle rette

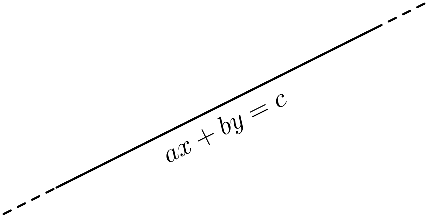
Come rappresentare una retta?



Rappresentazione delle rette

Come rappresentare una retta?

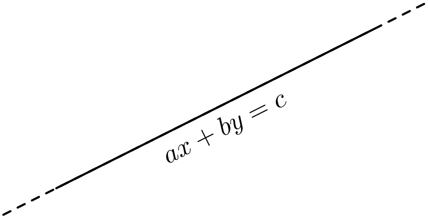
- Forma implicita: $ax + by = c$.



Rappresentazione delle rette

Come rappresentare una retta?

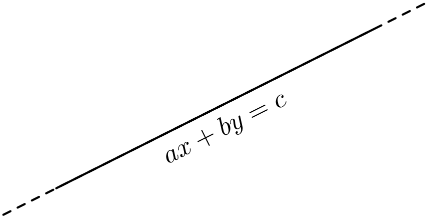
- Forma implicita: $ax + by = c$. Non particolarmente comodo.



Rappresentazione delle rette

Come rappresentare una retta?

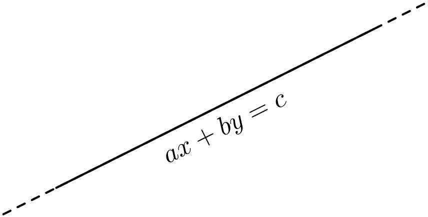
- Forma implicita: $ax + by = c$. Non particolarmente comodo.
- Forma esplicita: $y = mx + q$.



Rappresentazione delle rette

Come rappresentare una retta?

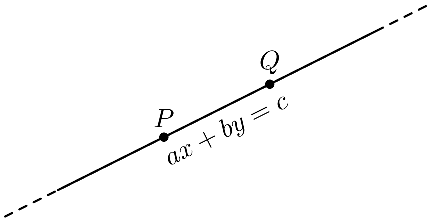
- Forma implicita: $ax + by = c$. Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. Comodo per il convex hull trick.



Rappresentazione delle rette

Come rappresentare una retta?

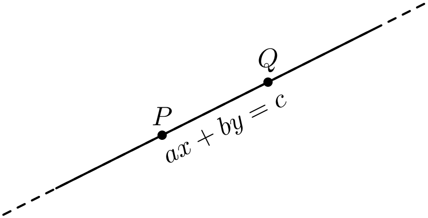
- Forma implicita: $ax + by = c$. Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. Comodo per il convex hull trick.
- Dati due punti: (P, Q).



Rappresentazione delle rette

Come rappresentare una retta?

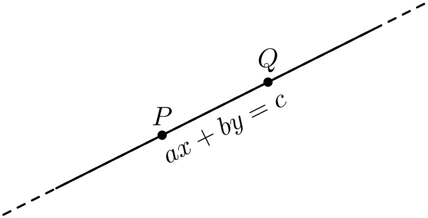
- Forma implicita: $ax + by = c$. Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. Comodo per il convex hull trick.
- Dati due punti: (P, Q). Comodo per rappresentare i segmenti.



Rappresentazione delle rette

Come rappresentare una retta?

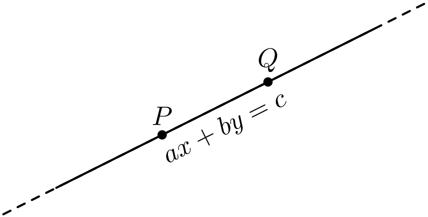
- Forma implicita: $ax + by = c$. — Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. — Comodo per il convex hull trick.
- Dati due punti: (P, Q). — Comodo per rappresentare i segmenti.
- Data la direzione e l'offset: $(\vec{v}, c)$.



Rappresentazione delle rette

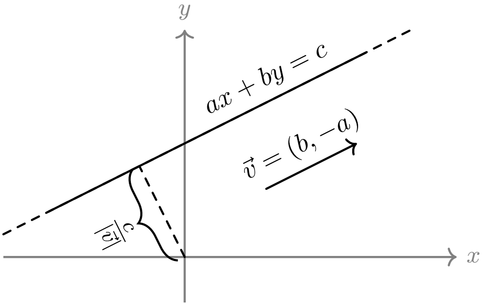
Come rappresentare una retta?

- Forma implicita: $ax + by = c$. Non particolarmente comodo.
- Forma esplicita: $y = mx + q$. Comodo per il convex hull trick.
- Dati due punti: (P, Q). Comodo per rappresentare i segmenti.
- Data la direzione e l'offset: $(\vec{v}, c)$. Comodo per effettuare operazioni.



Rappresentazione delle rette

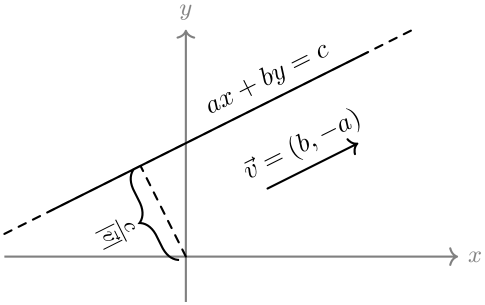
Implementazione



Rappresentazione delle rette

Implementazione

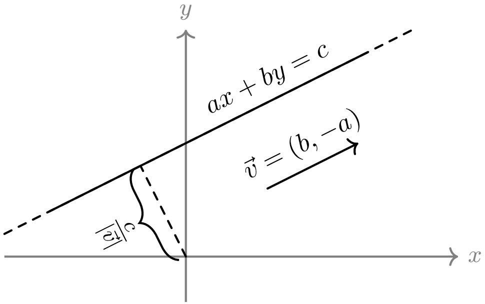
```
struct line {
    pt v; ll c;


};
```



Rappresentazione delle rette

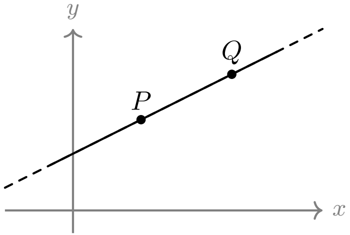
Implementazione

```
struct line {
    pt v; ll c;
    line(pt v, ll c) : v(v), c(c) {}                    // (v⃗, c)
    line(ll a, ll b, ll c) : v({b, -a}), c(c) {}      // ax + by = c
};
```



Retta per due punti

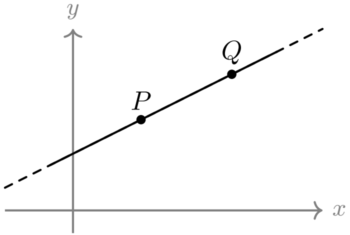
Definizione



Retta per due punti

Definizione

$$\vec{v} = \vec{q} - \vec{p}$$

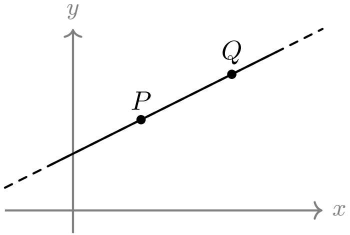


Retta per due punti

Definizione

$$\vec{v} = \vec{q} - \vec{p}$$

$$\overrightarrow{(b,-a)} \times \overrightarrow{(x,y)} = ax + by = c$$

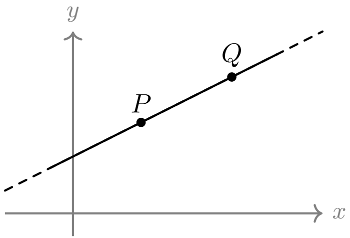


Retta per due punti

Definizione

$$\vec{v} = \vec{q} - \vec{p}$$

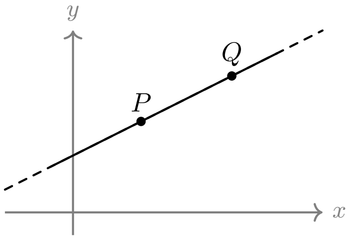
$$\overrightarrow{(b, -a)} \times \overrightarrow{(x, y)} = ax + by = c \quad \Longrightarrow \quad c = \vec{v} \times \vec{p}$$



Retta per due punti

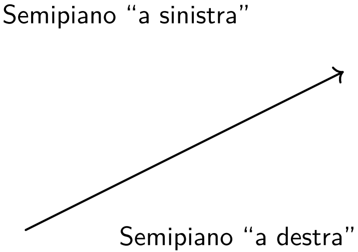
Implementazione

```
struct line {
    ...
    line(pt p, pt q) : v(q - p), c(cross(v, p)) {}
};
```



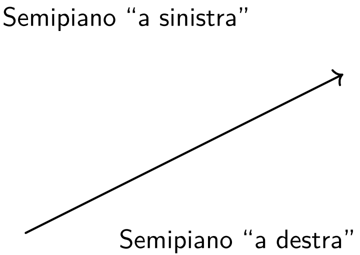
Punto-retta

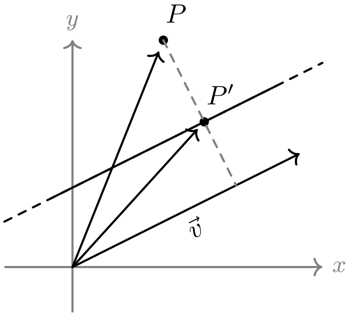
Implementazione



Punto-retta

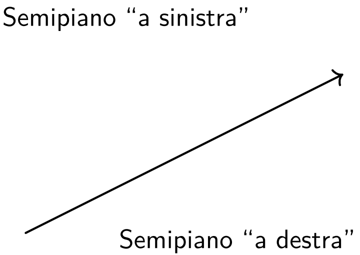
Implementazione

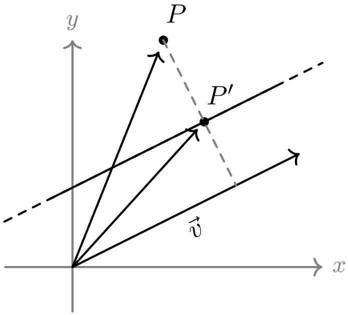




Punto-retta

Implementazione





$$\vec{v} \times \vec{p} \overset{?}{\lesseqgtr} \vec{v} \times \vec{p'}$$

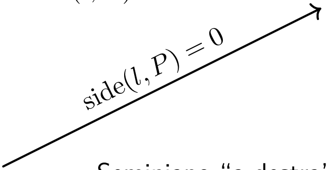
Punto-retta

Implementazione

```
ll side(line l, pt p) {
    return cross(l.v, p) - l.c;
}
```

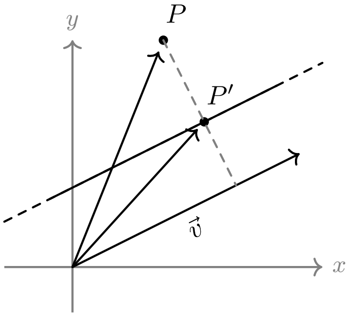
Semipiano "a sinistra"

$$\mathrm{side}(l, P) > 0$$



Semipiano "a destra"

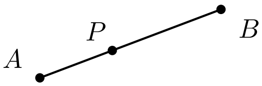
$$\mathrm{side}(l, P) < 0$$



$$\vec{v} \times \vec{p} \overset{?}{\lessgtr} \vec{v} \times \vec{p'}$$

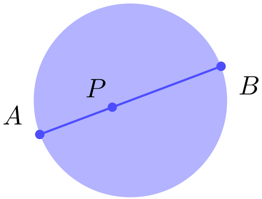
Punto-segmento

Implementazione



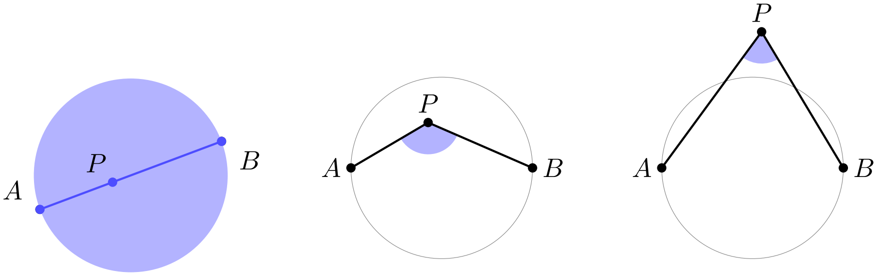
Punto-segmento

Implementazione



Punto-segmento

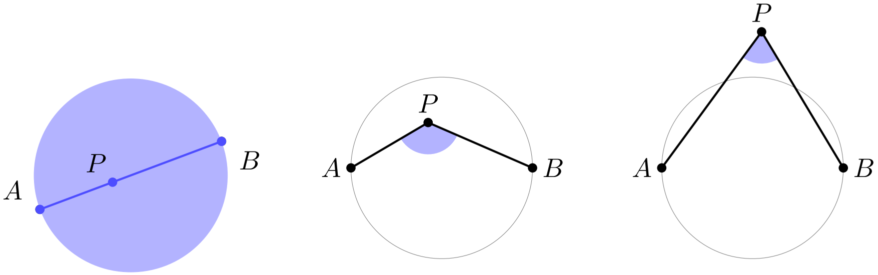
Implementazione



Punto-segmento

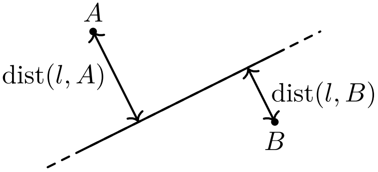
Implementazione

```
bool onSegment(pt a, pt b, pt p) {
    return orient(a, b, p) == 0 && dot(a - p, b - p) <= 0;
}
```



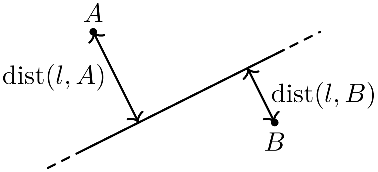
Distanza punto-retta

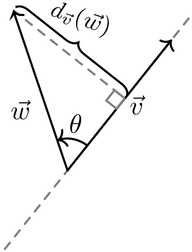
Implementazione



Distanza punto-retta

Implementazione



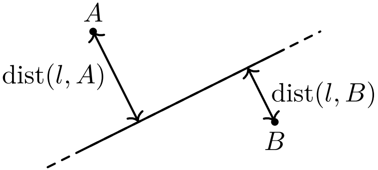


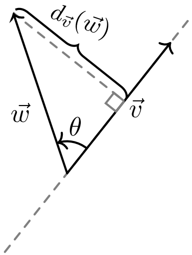
$$\vec{v} \times \vec{w} = |\vec{v}| d_{\vec{v}}(\vec{w})$$

Distanza punto-retta

Implementazione

```cpp
double dist(line l, pt p) {
    return std::abs(side(l, p)) / abs(l.v);
}
```

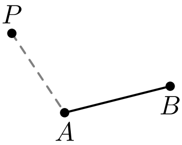




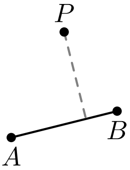
$$\vec{v} \times \vec{w} = |\vec{v}| d_{\vec{v}}(\vec{w})$$

Distanza punto-segmento

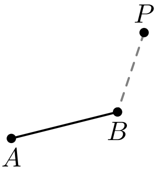
Implementazione



Più vicino a A.



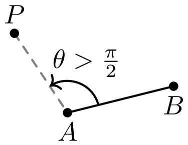
Più vicino alla proiezione.



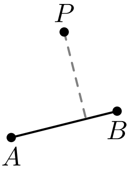
Più vicino a B.

Distanza punto-segmento

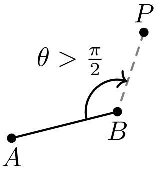
Implementazione



Più vicino a A.



Più vicino alla proiezione.

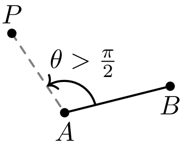


Più vicino a B.

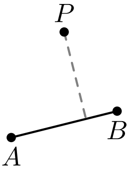
Distanza punto-segmento

Implementazione

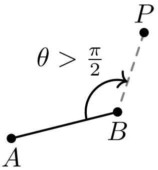
```
double dist(pt a, pt b, pt p) {
    if (dot(p - a, b - a) > 0 && dot(p - b, a - b) > 0) {
        return dist({a, b}, p);
    } else {
        return std::min(dist(p, a), dist(p, b));
    }
}
```



Più vicino a A.



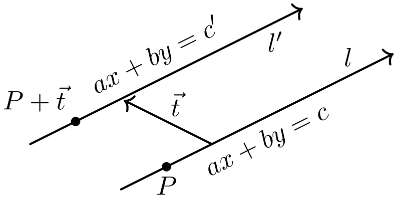
Più vicino alla proiezione.



Più vicino a B.

Traslazione

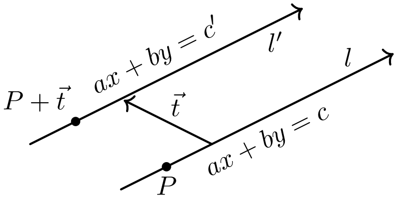
Implementazione



Traslazione

Implementazione

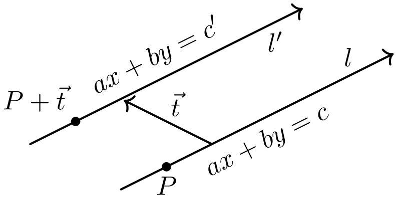
```
line translate(line l, pt t) {
    return {l.v, l.c + cross(l.v, t)};
}
```

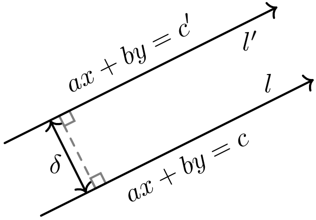


Traslazione

Implementazione

```
line translate(line l, pt t) {
    return {l.v, l.c + cross(l.v, t)};
}
```

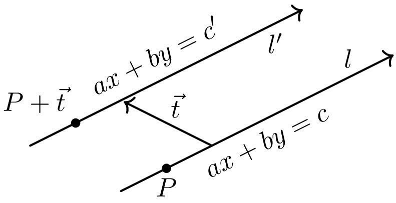


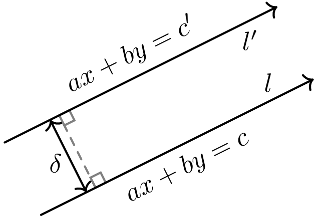


Traslazione

Implementazione

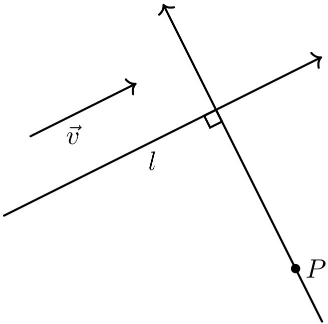
```
line translate(line l, pt t) {
    return {l.v, l.c + cross(l.v, t)};
}
line translate(line l, double dist) {
    return {l.v, l.c + dist * abs(l.v)}; // Richiede i float
}
```





Retta parallela e perpendicolare

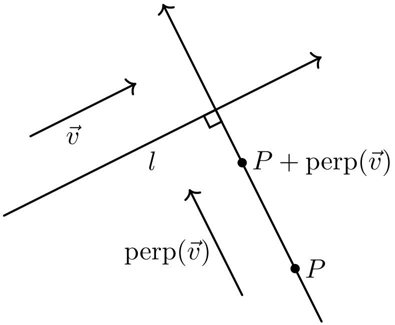
Implementazione



Retta parallela e perpendicolare

Implementazione

```
line paral(line l, pt p) {
    return {p, p + l.v};
}
line perp(line l, pt p) {
    return {p, p + perp(l.v)};
}
```



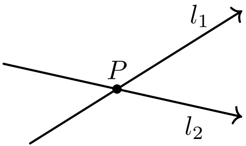
Altre operazioni

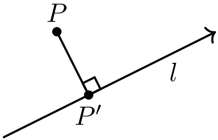
Intersezione tra due rette

```
// Richiede i float
std::optional<pt> intersect(line l1, line l2) {
    double d = cross(l1.v, l2.v);
    if (std::abs(d) < EPS) return {};
    return (l2.v * l1.c - l1.v * l2.c) / d;
}
```

Proiezione di un punto su una retta

```
// Richiede i float
pt project(line l, pt p) {
    return p - perp(l.v) * side(l, p) / norm(l.v);
}
```

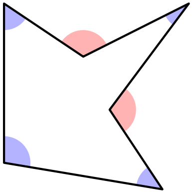


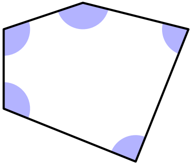


Poligoni

Convessità

Implementazione

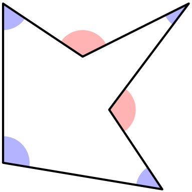

Non convesso


Convesso

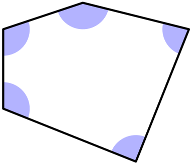
Convessità

Implementazione

```
bool isConvex(int N, vector<pt> P) {
    bool hasPos = false, hasNeg = false;
    for (int i = 0; i < N; i++) {
        ll o = orient(P[i], P[(i + 1) % N], p[(i + 2) % N]);
        hasPos += (o > 0);
        hasNeg += (o < 0);
    }
    return !(hasPos && hasNeg);
}
```



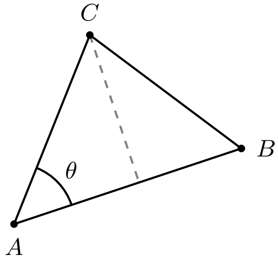
Non convesso



Convesso

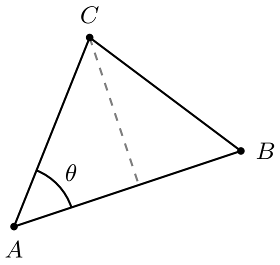
Area di un triangolo

Implementazione



Area di un triangolo

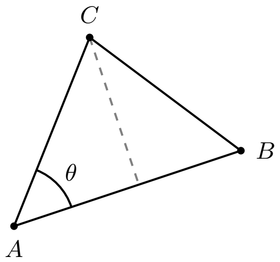
Implementazione



$$|\overset{\triangle}{ABC}| = \frac{1}{2}|AB||AC|\sin\theta$$

Area di un triangolo

Implementazione

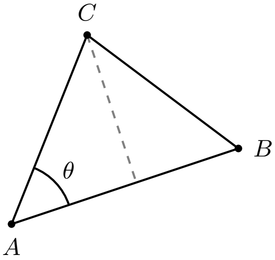


$$|\overset{\triangle}{ABC}| = \frac{1}{2}|AB||AC|\sin\theta = \frac{1}{2}\left|\overrightarrow{AB} \times \overrightarrow{AC}\right|$$

Area di un triangolo

Implementazione

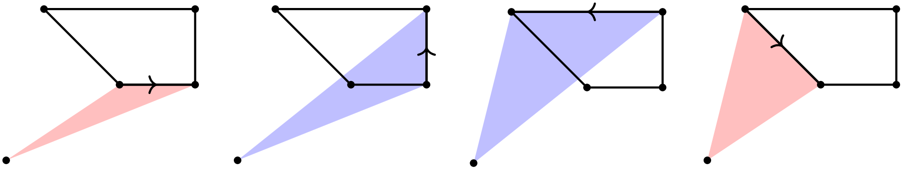
```
double areaTriangle(pt a, pt b, pt c) {
    return std::abs(cross(b - a, c - a)) / 2.0;
}
```



$$|\overset{\triangle}{ABC}| = \frac{1}{2}|AB||AC|\sin\theta = \frac{1}{2}\left|\overrightarrow{AB} \times \overrightarrow{AC}\right|$$

Area di un poligono

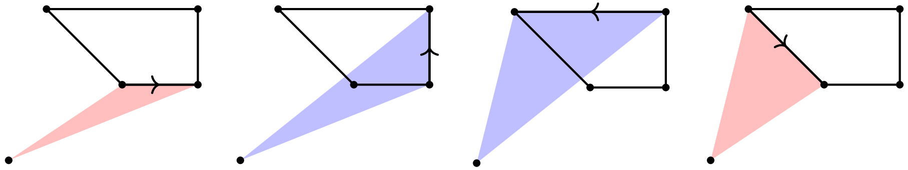
Implementazione



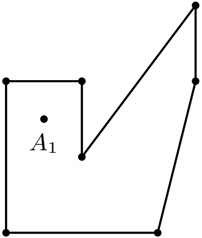
Area di un poligono

Implementazione

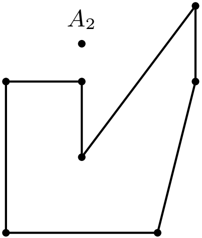
```
double area(vector<pt> P) {
    ll res = 0;
    for (size_t i = 0; i < P.size(); i++) {
        res += cross(P[i], P[(i + 1) % P.size()]);
    }
    return std::abs(res) / 2.0;
}
```



Punto interno a un poligono

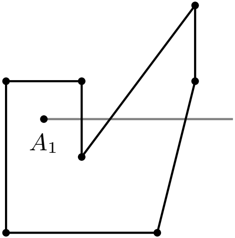


punto interno

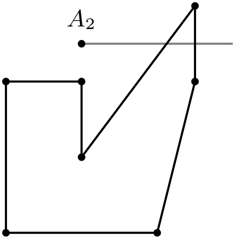


punto esterno

Punto interno a un poligono

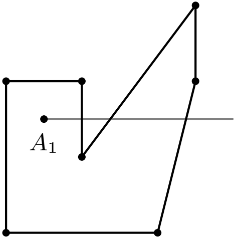


3 intersezioni $\Longrightarrow$ punto interno

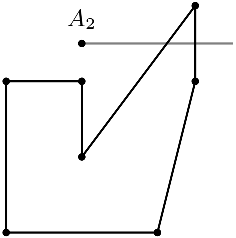


2 intersezioni $\Longrightarrow$ punto esterno

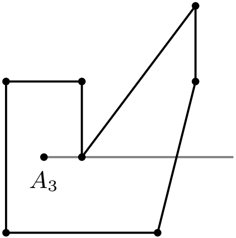
Punto interno a un poligono

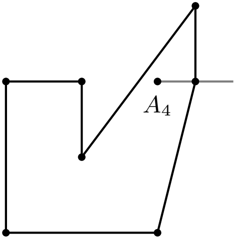


3 intersezioni $\Longrightarrow$ punto interno

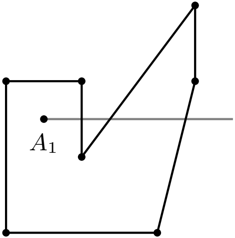


2 intersezioni $\Longrightarrow$ punto esterno

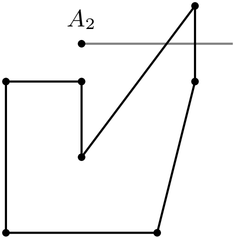




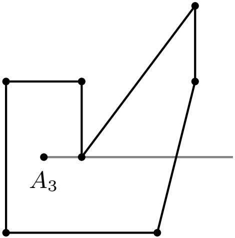
Punto interno a un poligono



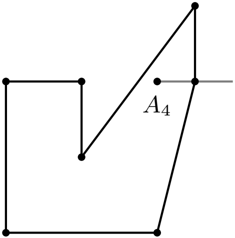
3 intersezioni $\Longrightarrow$ punto interno



2 intersezioni $\Longrightarrow$ punto esterno

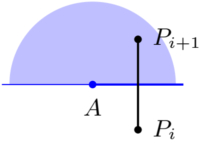


2 intersezioni?

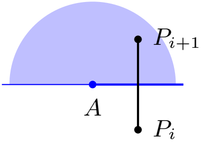


1 intersezione?

Punto interno a un poligono

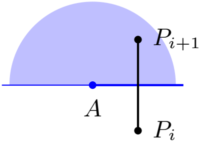


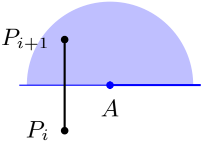
Punto interno a un poligono



- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓

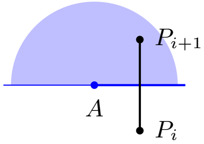
Punto interno a un poligono



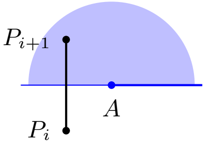


- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓

Punto interno a un poligono

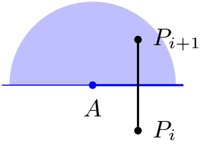


- Intersezione: $\checkmark$
- Semipiani diversi: $\checkmark$
- Segmento valido: $\checkmark$

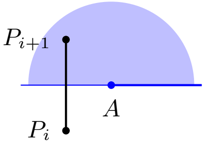


- Intersezione: $\times$
- Semipiani diversi: $\checkmark$
- Segmento non valido: $\times$

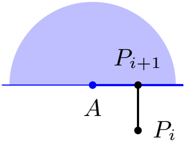
Punto interno a un poligono



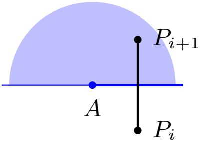
- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓



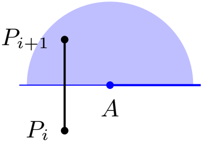
- Intersezione: ×
- Semipiani diversi: ✓
- Segmento non valido: ×



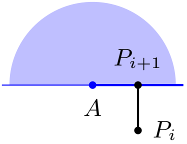
Punto interno a un poligono



- Intersezione: $\checkmark$
- Semipiani diversi: $\checkmark$
- Segmento valido: $\checkmark$

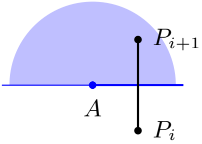


- Intersezione: $\times$
- Semipiani diversi: $\checkmark$
- Segmento non valido: $\times$

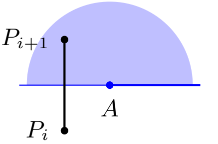


- Intersezione: $\checkmark$
- Semipiani diversi: $\checkmark$
- Segmento valido: $\checkmark$

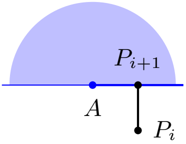
Punto interno a un poligono



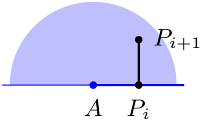
- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓



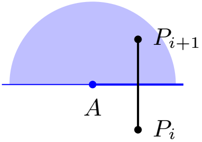
- Intersezione: ×
- Semipiani diversi: ✓
- Segmento non valido: ×



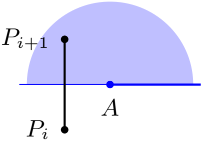
- Intersezione: ✓
- Semipiani diversi: ✓
- Segmento valido: ✓



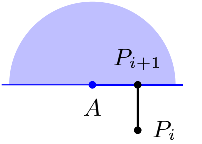
Punto interno a un poligono



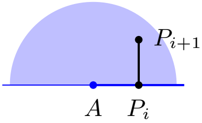
- Intersezione: $\checkmark$
- Semipiani diversi: $\checkmark$
- Segmento valido: $\checkmark$



- Intersezione: $\times$
- Semipiani diversi: $\checkmark$
- Segmento non valido: $\times$



- Intersezione: $\checkmark$
- Semipiani diversi: $\checkmark$
- Segmento valido: $\checkmark$



- Intersezione: $\checkmark$
- Semipiani diversi: $\times$
- Segmento non valido: $\times$

Punto interno a un poligono

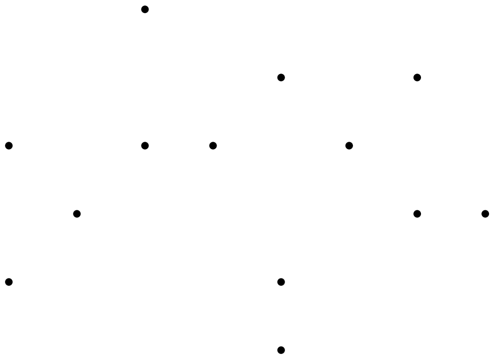
Implementazione

```
bool above(pt a, pt p) {
    return p.y >= a.y;
}
bool crossesRay(pt a, pt p, pt q) {
    return (above(a, q) - above(a, p)) * orient(a, p, q) > 0;
}
bool inPolygon(vector<pt> P, pt a, bool strict = false) {
    int cnt = 0;
    for (size_t i = 0; i < P.size(); i++) {
        if (onSegment(P[i], P[(i + 1) % P.size()], a))
            return !strict;
        cnt += crossesRay(a, P[i], P[(i + 1) % P.size()]);
    }
    return cnt % 2;
}
```

Convex hull

Definizione

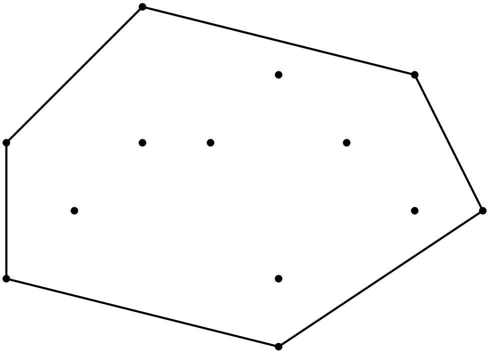
Il convex hull è il più piccolo poligono **convesso** che contiene tutti i punti di un dato insieme.



Convex hull

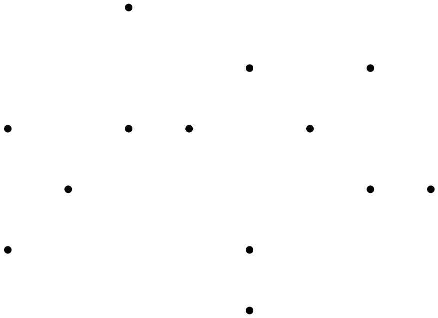
Definizione

Il convex hull è il più piccolo poligono **convesso** che contiene tutti i punti di un dato insieme.



Convex hull

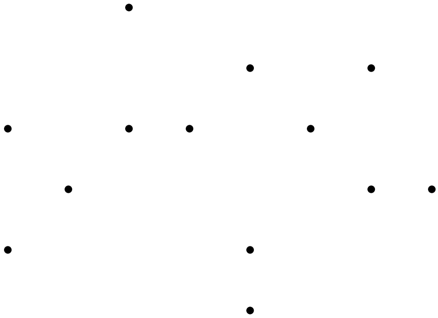
Andrew's algorithm



Convex hull

Andrew's algorithm

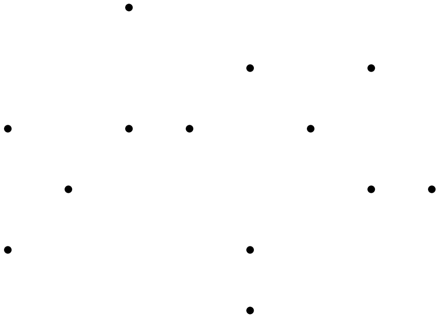
- iteriamo i punti in ordine crescente di coordinata x;



Convex hull

Andrew's algorithm

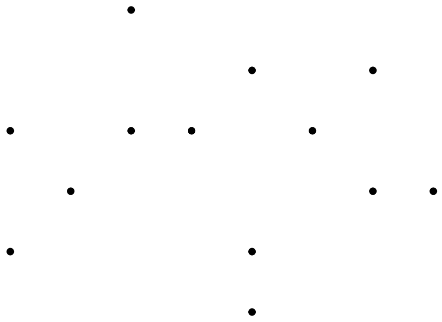
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;



Convex hull

Andrew's algorithm

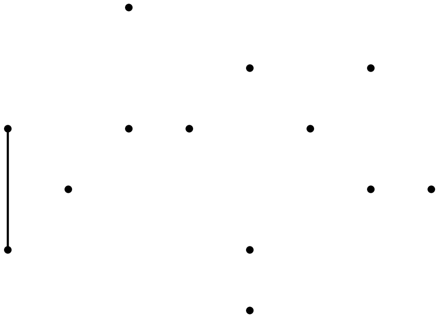
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

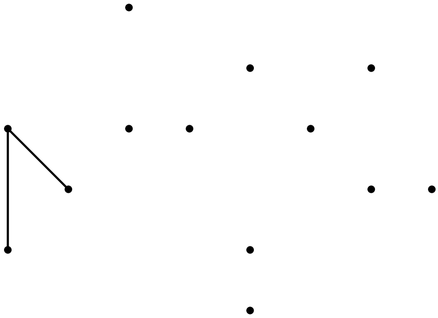
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

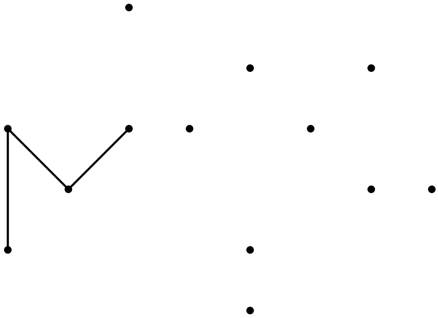
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

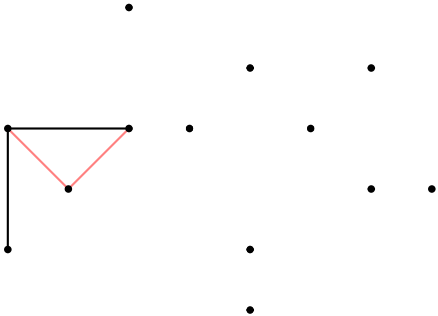
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

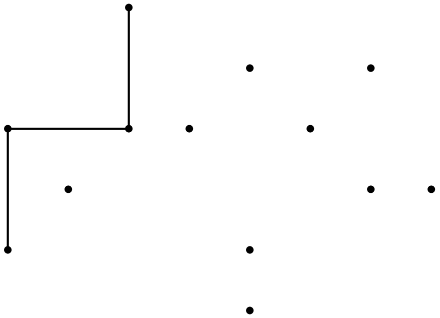
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

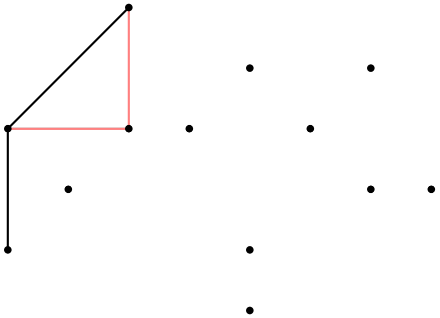
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

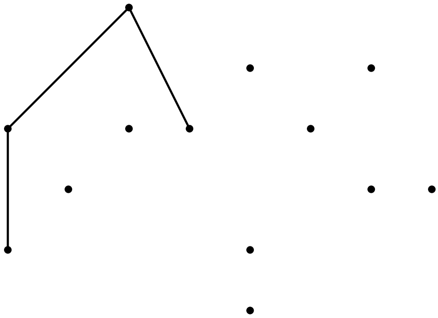
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

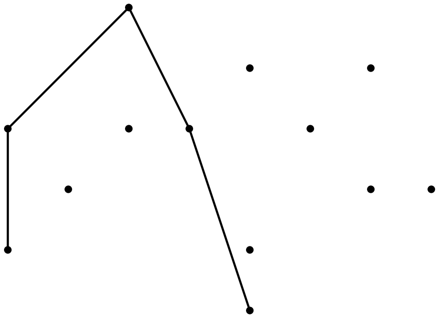
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

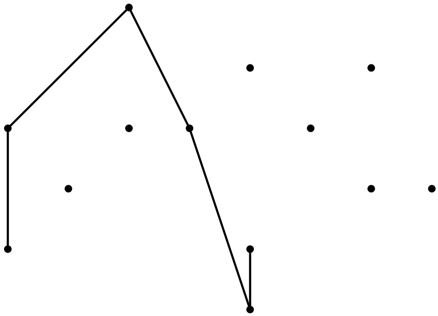
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

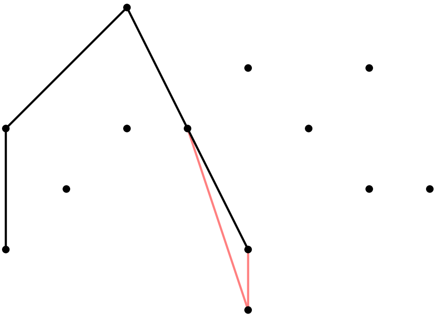
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

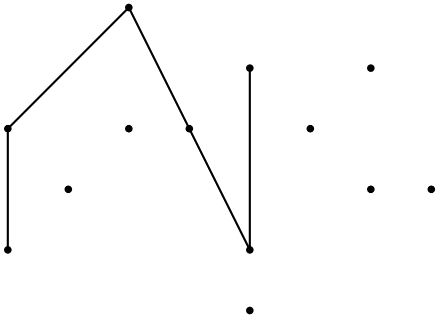
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

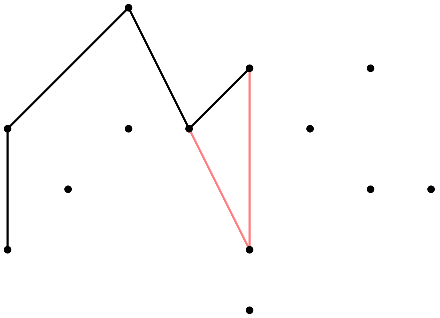
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

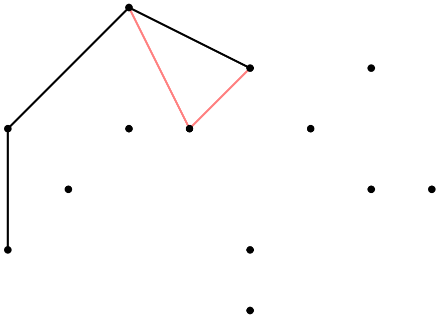
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

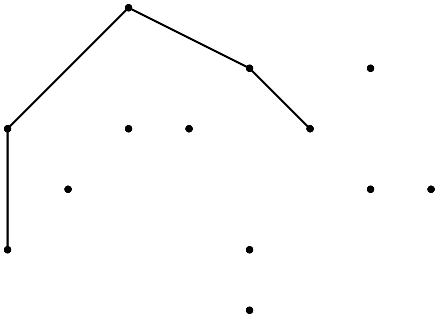
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

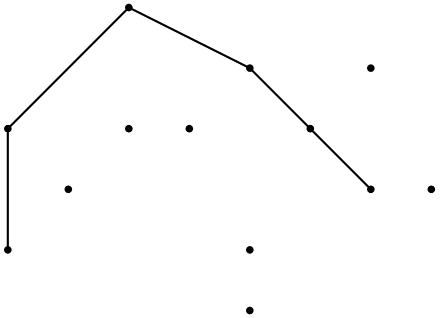
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

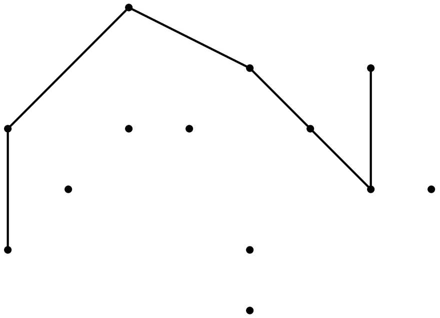
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

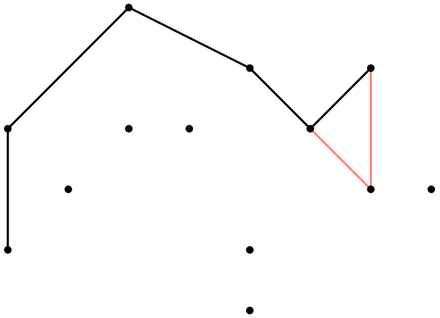
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

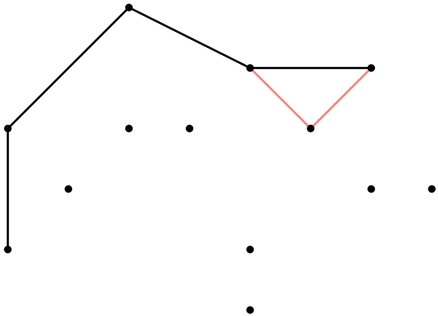
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

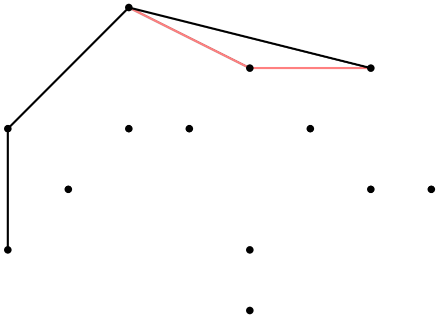
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

Andrew's algorithm

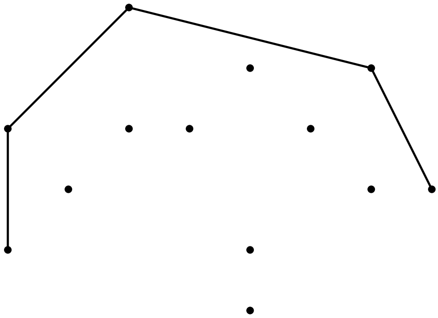
- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

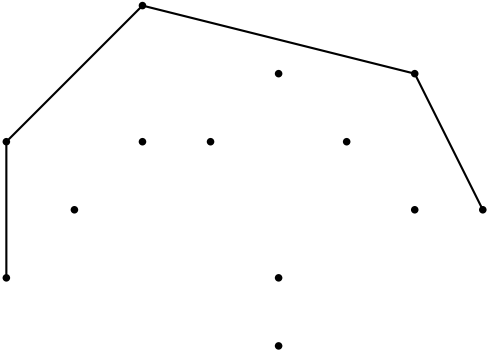
Andrew's algorithm

- iteriamo i punti in ordine crescente di coordinata x;
- aggiungiamo i punti al convex hull uno alla volta;
- dopo aver aggiunto un punto, controlliamo che l'ultimo segmento non "giri" a sinistra, in tal caso lo rimuoviamo;



Convex hull

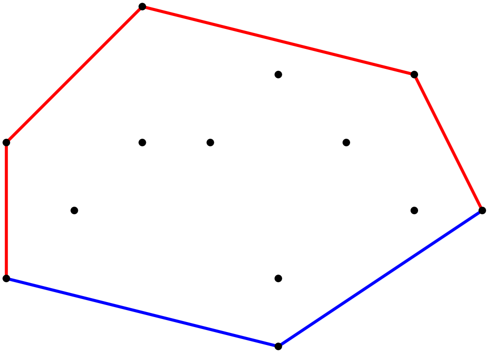
Andrew's algorithm



Convex hull

Andrew's algorithm

- infine ripetiamo il procedimento in ordine decrescente di coordinata x per determinare la metà inferiore del convex hull.



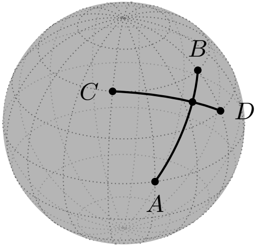
Convex hull

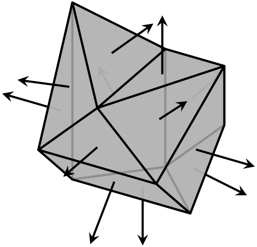
Implementazione

```
vector<pt> convexHull(vector<pt> P, bool all = true) {
    sort(P.begin(), P.end(), [](pt a, pt b) {
        return tie(a.x, a.y) < tie(b.x, b.y);
    });

    vector<pt> ch;
    for (int i = 0; i < 2; i++) {
        for (pt p : P) {
            while (ch.size() >= 2 &&
                    orient(ch[ch.size()-2], ch.back(), p) >= (all ? 0 : 1)) {
                ch.pop_back();
            }
            ch.push_back(p);
        }
        ch.pop_back();
        reverse(P.begin(), P.end());
    }

    return ch;
}
```

Geometria 3D

Per il prossimo stage...





Variabili floating point

Evitare i float

Usare i float solo se necessario

Evitare i float

Usare i float solo se necessario

- Usare i float solo quando è strettamente necessario.

Evitare i float

Usare i float solo se necessario

- Usare i float solo quando è strettamente necessario.
- Cercare sempre di usare una rappresentazione alternativa in cui i valori sono interi ($2x$, x^2, ...).

Quale tipo usare?

Tipo	
`float`	
`double`	
`long double`	
`__float128`	

Quale tipo usare?

Tipo	Precisione	
`float`	24 bit	
`double`	53 bit	
`long double`	64 bit	
`__float128`	113 bit	

Quale tipo usare?

Tipo	Precisione	Errore relativo	
`float`	24 bit	$6.0 \cdot 10^{-8}$	
`double`	53 bit	$1.1 \cdot 10^{-16}$	
`long double`	64 bit	$5.4 \cdot 10^{-20}$	
`__float128`	113 bit	$9.6 \cdot 10^{-35}$	

Quale tipo usare?

Tipo	Precisione	Errore relativo	Valore massimo
`float`	24 bit	$6.0 \cdot 10^{-8}$	$3.4 \cdot 10^{38}$
`double`	53 bit	$1.1 \cdot 10^{-16}$	$1.7 \cdot 10^{308}$
`long double`	64 bit	$5.4 \cdot 10^{-20}$	$1.1 \cdot 10^{4932}$
`__float128`	113 bit	$9.6 \cdot 10^{-35}$	$1.2 \cdot 10^{4932}$

Quale tipo usare?

Tipo	Precisione	Errore relativo	Valore massimo	Velocità
`float`	24 bit	$6.0 \cdot 10^{-8}$	$3.4 \cdot 10^{38}$	veloce
`double`	53 bit	$1.1 \cdot 10^{-16}$	$1.7 \cdot 10^{308}$	decente
`long double`	64 bit	$5.4 \cdot 10^{-20}$	$1.1 \cdot 10^{4932}$	lento
`__float128`	113 bit	$9.6 \cdot 10^{-35}$	$1.2 \cdot 10^{4932}$	molto lento

Consigli

Come usare correttamente i float

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.
- Minimizzare le operazioni che amplificano l'errore ($1/x$, $\sqrt{x}$, ...).

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.
- Minimizzare le operazioni che amplificano l'errore ($1/x$, $\sqrt{x}$, ...).
- Controllare il dominio delle funzioni ($\sqrt{x}$, $\arccos$, $\log$, ...).

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.
- Minimizzare le operazioni che amplificano l'errore ($1/x$, $\sqrt{x}$, ...).
- Controllare il dominio delle funzioni ($\sqrt{x}$, $\arccos$, $\log$, ...).
- Cercare di eseguire operazioni con numeri dello stesso magnitudo.

Consigli

Come usare correttamente i float

- Compensare l'errore durante i confronti.
- Minimizzare le operazioni che amplificano l'errore ($1/x$, $\sqrt{x}$, ...).
- Controllare il dominio delle funzioni ($\sqrt{x}$, $\arccos$, $\log$, ...).
- Cercare di eseguire operazioni con numeri dello stesso magnitudo.
- Usare `-ffast-math` (abilitato da `-Ofast`) può rendere il codice più veloce, ma lo rende non conforme allo standard IEEE.

Esercizi

Problemi

- Sezione "Geometry" di CSES:
  https://cses.fi/problemset/list/
- Aerodynamic:
  https://codeforces.com/contest/1299/problem/B
- Balloon Darts:
  https://codeforces.com/gym/104466/problem/B
- Break a leg!:
  https://codeforces.com/gym/104945/problem/H

Bibliografia

Antti Laaksonen.
Guide to Competitive Programming.
Springer, 2017.

Victor Lecomte.
Handbook of geometry for competitive programmers.
`https://victorlecomte.com/cp-geo.pdf`, 2018.