UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN: Ingegneria Informatica

Analisi, implementazione e confronto di algoritmi per l'inviluppo convesso

Relatore: Ch.mo Prof. Geppino Pucci

Laureando: Christian Pastò

ANNO ACCADEMICO 2021 – 2022

Data di laurea 19/09/2022

Indice generale

1 Introduzione

La geometria computazionale è una branca dell'informatica che studia gli algoritmi per risolvere problemi di natura geometrica e la loro implementazione.
Le tipologie di problemi trattati sono molteplici, vengono riportati alcuni tra i più comuni:

- **Inviluppo convesso** (argomento di questa tesi): trovare il più piccolo poligono convesso che racchiuda tutti i punti di un insieme di punti finito dato in ingresso. In questa tesi verranno trattati diversi algoritmi (sia nel caso bidimensionale che in un generico caso multidimensionale) per risolvere tale problema e verranno successivamente confrontati tra di loro.
- **Determinare se un insieme di segmenti si interseca**: dato un insieme di semirette, determinare se si verifica almeno un'intersezione tra di esse.
- **Coppia di punti più vicini**: dato un insieme di punti, determinare tramite una procedura efficiente, la coppia di punti più vicini.
- **Triangolazione**: dato un generico poligono, effettuarne una suddivisione poligonale i cui elementi sono triangoli.

Alcuni dei problemi sopracitati potrebbero a prima vista risultare semplici da risolvere, in realtà il ruolo della geometria computazionale (e anche di altre branche dell'informatica) non è solo quello di trovare una soluzione ad un problema ma anche quello di trovare una soluzione nel modo più efficiente possibile.

Nella pratica la geometria computazionale trova spazio nei più disparati ambiti dell'ingegneria (e non solo) come la grafica computerizzata, la robotica, CAD/CAM, sistemi geografici informativi, la statistica e tanti altri ancora.

In genere l'input di un problema di geometria computazionale è un insieme di oggetti geometrici, ad esempio un insieme di punti o di rette, mentre l'output o è una struttura geometrica sull'insieme di input, oppure una risposta ad una domanda dell'output come nel problema di determinare se un insieme di segmenti si interseca.

In questa tesi, come precedentemente citato, verrà trattato il problema dell'inviluppo convesso. Dapprima verranno descritti e analizzati alcuni dei più importanti algoritmi per l'inviluppo convesso in uno spazio 2-dimensionale, denominati **Graham-Scan**, **Jarvis-march** e **quick-hull,** successivamente verrà esteso l'algoritmo quick-hull per trattare il problema del convex hull in uno spazio generico d-dimensionale. Il motivo di voler sviluppare due versioni di quickhull è perché

nella versione per il caso 2-dimensionale verrà utilizzata la nota tecnica del *Divide & Conquer*, mentre per il caso d-dimensionale l'approccio *D&C* viene eliminato a favore di un'implementazione iterativa. Prima di entrare nel dettaglio implementativo di tali algoritmi sarà utile dedicare un capitolo ai concetti di geometria lineare utilizzati per la loro realizzazione. La tesina procede con un capitolo dedicato a riportare vari esperimenti eseguiti sugli algoritmi, in particolare riguardanti l'analisi delle prestazioni ed infine un ultimo capitolo in cui verranno riportate la conclusioni sul lavoro svolto per produrre questa tesi.

1.1 Inviluppo convesso

Dato un insieme di punti *S*, si definisce inviluppo convesso (in inglese Convex Hull) di *S*, **CH**(*S*) il più piccolo poligono convesso tale per cui tutti i punti di *S* si trovano al suo interno o sul suo perimetro.

Più formalmente dato $S \in \mathbb{R}^k$, l'inviluppo convesso di *S* è l'insieme di tutte le combinazioni convesse di punti di *S.* Se l'insieme di partenza è finito allora **CH**(*S*) è un politopo, definito come:

$$\mathbf{CH}(S) = \left\{ \sum_{j=0}^{n-1} \alpha_j x_j \middle| \sum_{j=0}^{n-1} \alpha_j = 1 \right\}$$

dove α_j sono numeri reali positivi e x_j i punti di *S*.

I vertici di **CH**(*S*) formano un sottoinsieme di punti di *S* i cui elementi sono ritornati in un ordine ben definito: vedremo che gli algoritmi per il caso 2-dimensionale restituiscono i punti dell'inviluppo convesso in un determinato ordine (antiorario in genere anche se è possibile con una semplice modifica ottenere i punti in senso orario).

Una definizione più informale per definire un poligono convesso, è che un poligono convesso è tale se tutti i suoi angoli interni sono $\leq 180°$. Questa definizione può essere utile per comprendere al meglio la logica utilizzata dagli algoritmi per l'inviluppo convesso.

Il problema dell'inviluppo convesso nel caso monodimensionale è banale in quanto consiste nel trovare il minimo e il massimo elemento in un array, operazione che può essere effettuata in un tempo *O(n)* tramite una semplice scansione, dove *n* è la cardinalità dell'insieme, pertanto il problema del convex hull inizia a diventare interessante dalle due dimensioni in poi.

Figura 1.2: Esempio di inviluppo convesso in una dimensione.

Un caso particolare si verifica quando i vertici dell'insieme di partenza sono i punti di un poligono convesso, in questa situazione l'inviluppo convesso dell'insieme di input è l'insieme stesso. Vedremo che questo caso sarà di notevole importanza nello studio delle prestazioni di alcuni tipi di algoritmi per l'inviluppo convesso in quanto il numero dei vertici appartenenti all'inviluppo convesso è uno dei criteri principali per la scelta dell'algoritmo da utilizzare.

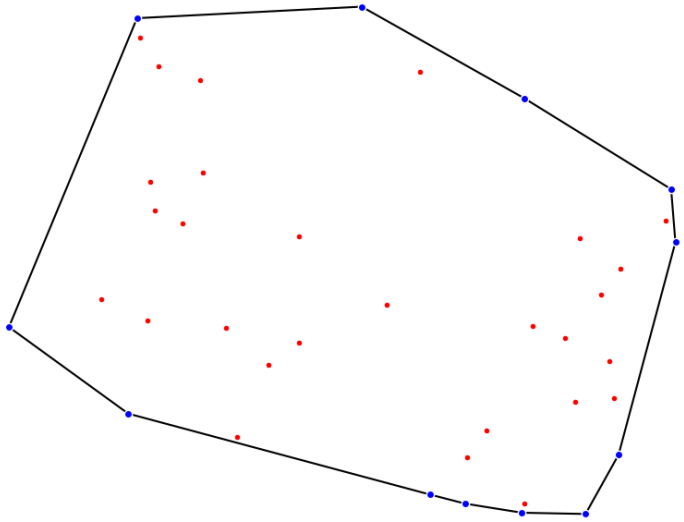

Figura 1.2: esempio di inviluppo convesso nel caso 2-dimensionale.

Un approccio pratico per comprendere il concetto di inviluppo convesso è quello di immaginare i punti di *S* come dei chiodi che sporgono da una tavola, l'inviluppo convesso si può intendere come un elastico che circonda tutti i chiodi.

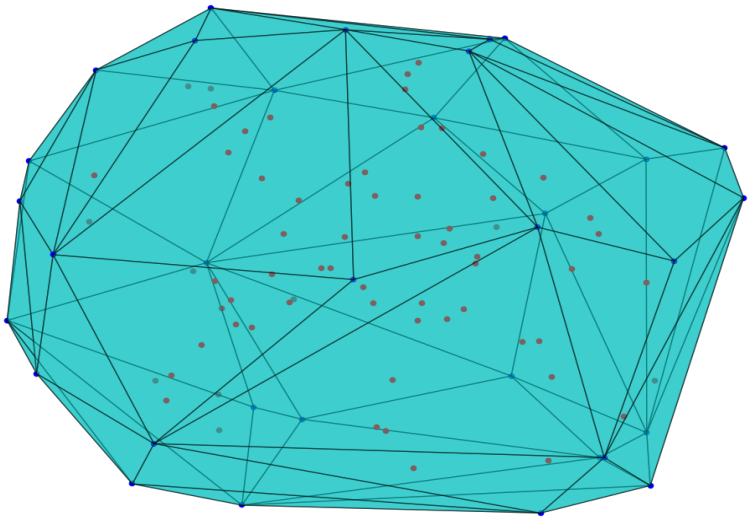

Figura 1.3: Esempio di inviluppo convesso in 3 dimensioni.

L'inviluppo convesso trova applicazioni principalmente in **matematica e in geometria discreta** dove è spesso ricorrente in molte dimostrazioni.

Ovviamente anche in informatica il concetto dell'inviluppo convesso non è fine a se stesso ma è utilizzato come strumento per risolvere alcuni problemi di geometria computazionale, ad esempio un problema di interesse è quello di calcolare la coppia di punti più distanti in un insieme di vertici P di n elementi. La procedura consiste nel calcolare l'inviluppo convesso di P e successivamente trovare la coppia di punti più distanti. È dimostrato che questi due vertici fanno sicuramente parte di **CH**(P), per cui è sufficiente calcolare l'inviluppo convesso di P, ad esempio utilizzando l'algoritmo Graham-scan in un tempo $O(n\ logn)$ e successivamente trovare la coppia di punti più distanti in **CH**(P), operazione che può essere effettuata in un tempo $O(n)$ poiché è possibile eseguire tale operazione in tempo lineare su un poligono convesso. Alla fine si otterrà come complessità temporale $O(n\ logn)$ invece di $O(n^2)$ ottenuto tramite un approccio brute-force in cui si calcola la distanza di ogni possibile coppia di punti.

Un altro campo applicativo importante per l'inviluppo convesso è quello di calcolare la collision mesh nei **motori fisici**, ossia data una mesh tridimensionale calcolare il suo inviluppo convesso che fungerà come mesh per le collisioni dell'oggetto. L'inviluppo convesso viene usato come mesh per le collisioni solamente nel caso in cui non sia richiesta una elevata precisione. Questa procedura viene eseguita perché i motori fisici eseguono più efficientemente le collisioni mesh-to-mesh con modelli 3d convessi.

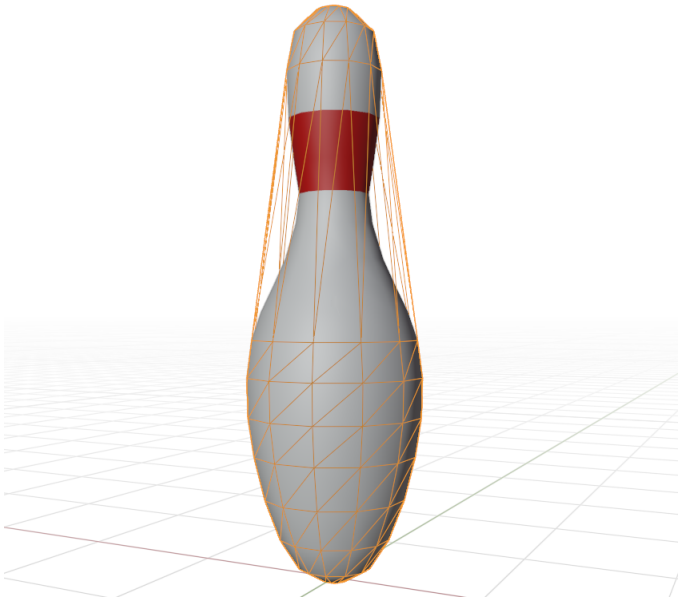

Figura 1.4: Collision mesh (in giallo) dell'oggetto che coincide con il suo inviluppo convesso.

Risulta chiaro anche da questo esempio che il problema dell'inviluppo convesso non si può limitare al caso 2-dimensionale ma è necessario spingersi in dimensioni maggiori di due, in particolare nella terza.

Un'altra applicazione importante riguarda l'impiego del concetto di inviluppo convesso nei problemi di **rilassamento**, in particolare per problemi di **Programmazione Lineare Intera** (PLI), ossia problemi di programmazione lineare dove le variabili possono assumere soltanto valori interi. In genere questo tipo di problemi non sono risolvibili in tempo polinomiale, ma un'eccezione si verifica quando un PLI gode della **proprietà di integralità**.

Un problema di Programmazione Lineare Intera viene definito come:

$$\begin{cases} min\, c\, \boldsymbol{x} \\ \boldsymbol{A}\, \boldsymbol{x} \geq b \\ \boldsymbol{x} \in \mathbb{Z}^n \end{cases}$$

Dove in ordine di altezza rispettivamente sono definite la funzione obiettivo, i vincoli e il dominio delle variabili.

Se in un PLI si eliminano i vincoli $\boldsymbol{x} \in \mathbb{Z}^n$ si ottiene un problema di programmazione lineare detto **rilassamento continuo**. Se accade che le soluzioni ottime del problema di programmazione lineare intera sono uguali a quelle del suo rilassamento, si dice che il problema gode della proprietà di integralità.

Più formalmente un PLI gode della proprietà di integralità se la regione ammissibile del suo rilassamento continuo coincide con il suo inviluppo convesso. Se la regione ammissibile del PLI originale viene definita come:

$$P = \{x \in \mathbb{Z}^n : Ax \geq b\}$$

Il problema gode della proprietà di integralità, se:

$$\{x \in \mathbb{R}^n : Ax \geq b\} = \boldsymbol{CH}(P)$$

dove a sinistra è indicata la regione del rilassamento continuo del PLI, e a destra l'inviluppo convesso dell'insieme P.

Avendo verificato tale proprietà allora un PLI può essere risolto tramite metodi semplici ed efficienti (come l'algoritmo del simplesso), alcuni dei problemi di programmazione lineare intera che godono della proprietà di integralità sono il problema del flusso di costo minimo e di flusso massimo su un grafo di flusso e il problema di accoppiamento di costo minimo.

2 Conoscenze preliminari

In questo capitolo verranno trattati gli argomenti di geometria lineare che permetteranno di implementare correttamente gli algoritmi per il convex hull. I primi concetti tratteranno solo argomenti di geometria in due dimensioni e sono quindi applicabili agli algoritmi per l'inviluppo convesso 2-dimensionale, mentre i successivi tratteranno argomenti applicabili in qualsiasi spazio d-dimensionale.

2.1 Angolo polare

L'**angolo polare** di un punto rispetto ad un asse e un polo in $\mathbb{R}^2$, è un concetto elementare della geometria, in particolare servirà per l'algoritmo di Graham e di Jarvis per ottenere una lista di punti ordinati in base al loro angolo polare rispetto a un punto di riferimento, specificando un asse.

Dati due punti $p_0=(p_{0x}, p_{0y})$, $p_1=(p_{1x}, p_{1y}) \in \mathbb{R}^2$, l'angolo polare che forma il segmento $\overline{p_0 p_1}$ è definito come l'angolo che forma tale segmento rispetto ad un vettore unitario che indica l'asse di riferimento di tale valore. Tale valore può assumere un valore compreso nell'intervallo $[0, 2\pi)$, così da considerare una rotazione completa verso un'unica direzione. Nel caso degli algoritmi in questione verrà considerato l'angolo tra un segmento orientato che congiunge due punti p_0 e p_1 e un vettore unitario rispetto all'asse di riferimento, ad esempio se bisogna considerare l'angolo polare di p_1 rispetto all'asse x positivo il vettore unitario a tale asse di riferimento sarà (1, 0).
Il procedimento segue la definizione di prodotto scalare tra due vettori, ossia dati due punti p_0 e $p_1 \in \mathbb{R}^2$ l'angolo polare θ rispetto a un asse di riferimento t è uguale all'angolo compreso tra il vettore $\boldsymbol{x}=p_1-p_0$ e il vettore unitario rispetto all'asse di riferimento $\boldsymbol{t_n}=(t_x, t_y)$.
Più formalmente:

$$\theta = \begin{cases} \arccos\left(\dfrac{\boldsymbol{x}\cdot\boldsymbol{t_n}}{\|\boldsymbol{x}\|\cdot\|\boldsymbol{t_n}\|}\right) & \textit{se} \quad p_{0y} \geq p_{1y} \\ 2\pi \;-\; \arccos\left(\dfrac{\boldsymbol{x}\cdot\boldsymbol{t_n}}{\|\boldsymbol{x}\|\cdot\|\boldsymbol{t_n}\|}\right) & \textit{altrimenti} \end{cases}$$

Il secondo caso della funzione garantisce che θ assuma valori compresi nell'intervallo $[0,2\pi)$ così da poter ordinare i punti in un ordine specificato dall'asse t (di solito orario o antiorario).

In pratica si rende p_0 il polo del nuovo asse e si calcola l'angolo tra il vettore $\boldsymbol{x}$ e $\boldsymbol{t_n}$ applicando la classica definizione di angolo tra due vettori.

Ordinare un insieme di punti in senso antiorario rispetto all'angolo polare con un vertice di riferimento

Esistono diversi metodi per effettuare questa operazione, uno dei quali può essere banalmente applicare la definizione appena descritta per ciascun punto di un insieme finito rispetto ad un vertice di riferimento.

In questo sottoparagrafo viene descritto un metodo computazionalmente più efficiente, in quanto non richiede di calcolare direttamente il valore dell'angolo polare utilizzando funzioni complesse (in termini di prestazioni), ma il tutto viene effettuato attraverso operazioni elementari.

Dato il punto di origine p_0 ed un generico punto dell'insieme originario p, la sequenza di punti si può ordinare in senso antiorario rispetto all'angolo polare del punto p, avendo p_0 come polo rispetto all'asse x positiva usando come chiave di ordinamento tale funzione:

$$key(p_0, p) = \frac{p_x - p_{0x}}{\|p_0 - p\|}$$

Utilizzando questa chiave si ha eliminato la funzione *arccos,* dalla quale dipendeva la maggior parte del tempo di esecuzione, in più si hanno eliminato due prodotti scalari.

In questo esempio i punti vengono ordinati rispetto all'angolo polare positivo in senso antiorario, per ordinare in base all'angolo polare negativo è sufficiente invertire p_0 con p.

Il motivo per cui questo metodo funziona è molto semplice: poiché le funzioni trigonometriche sono monotone globalmente, basta ordinare i punti in base all'argomento della funzione *arccos* senza calcolarne direttamente il valore assunto nel caso in cui si fosse interessati solo ad ordinare i punti.

2.2 Area con segno del parallelogramma

Rimanendo sempre nell'ambito delle due dimensioni, questo paragrafo illustra una tecnica per calcolare l'area con segno del parallelogramma generato da due vettori, questa concetto verrà utilizzata per tutti gli algoritmi dell'inviluppo convesso in due dimensioni, oltre ad essere utilizzato ampiamente anche in altri algoritmi di geometria computazionale che eseguono operazioni su

segmenti. Il vantaggio principale di questa operazione è che consente di rivelare parecchie informazioni su oggetti geometrici, con un costo computazionalmente minimo.

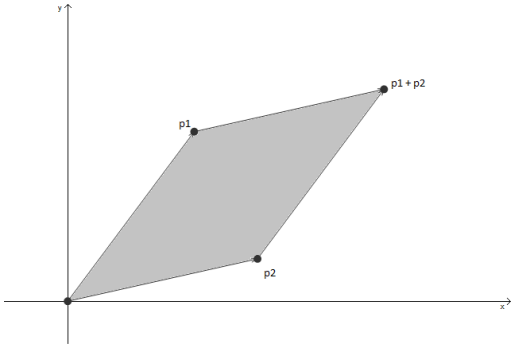

Figura 2.1 : in grigio l'area con segno del parallelogramma generato da $\boldsymbol{p_1}$ e $\boldsymbol{p_2}$.

Dati due vettori $\boldsymbol{p_1}=(p_{1x}, p_{1y})$, $\boldsymbol{p_2}=(p_{2x}, p_{2y}) \in \mathbb{R}^2$, si definisce l'area con segno del parallelogramma generato da $\boldsymbol{p_1}$ e $\boldsymbol{p_2}$, ossia il parallelogramma avente coordinate (0,0), p_1, p_2 e $p_1 + p_2$:

$$\boldsymbol{p_1} \times \boldsymbol{p_2} = p_{1x} p_{2y} - p_{2x} p_{1y}$$

Un equivalente metodo per effettuare tale operazione consiste nel calcolare il determinante della matrice **A**:

$$\boldsymbol{p_1} \times \boldsymbol{p_2} = det(\boldsymbol{A}) = \begin{pmatrix} p_{1x} & p_{1y} \\ p_{2x} & p_{2y} \end{pmatrix}$$

Se $\boldsymbol{p_1} \times \boldsymbol{p_2}$ è positivo allora $\boldsymbol{p_2}$ segue in senso orario $\boldsymbol{p_1}$, se negativo viceversa mentre se è nullo allora $\boldsymbol{p_1}$ e $\boldsymbol{p_2}$ sono collineari, ossia giacciono sulla stessa retta.

A volte con abuso di notazione tale operazione viene nominata anche prodotto vettoriale. Come è noto però, il prodotto vettoriale è definito solo per la terza dimensione, e rappresenta un vettore ortogonale a $\boldsymbol{p_1}$ e $\boldsymbol{p_2}$ secondo la regola della mano destra. Il risultato ottenuto tramite l'operazione spiegata in questo paragrafo può essere interpretato come la coordinata *z* del vettore risultante dal prodotto vettoriale $\boldsymbol{p_1} \times \boldsymbol{p_2}$ aventi come coordinate in tre dimensioni: $\boldsymbol{p_1}=(p_{1x}, p_{1y}, 0)$, $\boldsymbol{p_2}=(p_{2x}, p_{2y}, 0)$.

Ciò che interesserà maggiormente di questo prodotto è il segno, in quanto esso ci può indicare, oltre all'area del parallelogramma, anche la direzione in cui svoltano due segmenti consecutivi, ossia in quale senso ruota l'angolo tra i due segmenti.

Dati due segmenti consecutivi $\overline{p_0 p_1}$ e $\overline{p_1 p_2}$, si rende p_0 l'origine dei due vettori quindi si esegue $(p_2 - p_0) \times (p_1 - p_0)$, se il segno di questo prodotto è negativo si ha una svolta a sinistra nel punto p_1, a destra se il segno è maggiore di zero mentre se nullo vuol dire che i tre punti sono collineari.

2.3 Punto all'interno di un politopo convesso

In questa sezione si iniziano a trattare i primi argomenti di geometria in un generico spazio d-dimensionale. In particolare, descriveremo una tecnica, implementata negli algoritmi quick-hull per scartare dei punti dell'insieme iniziale, verificando se un punto si trova all'interno di un politopo convesso. Questa operazione è fondamentale in quanto alla fine di ogni iterazione del ciclo principale di quick-hull multidimensionale è necessario scartare i punti che si trovano all'interno di un temporaneo inviluppo convesso.

Se dato un poligono convesso $\boldsymbol{P}$ composto dalle facce $<f_1, f_2, \ldots, f_m>$, ciascuna delle quali ha associato un baricentro $<b_1, b_2, \ldots, b_m>$ calcolato come punto medio dei punti estremi delle facce, e un vettore normale $<\boldsymbol{n}_1, \boldsymbol{n}_2, \ldots, \boldsymbol{n}_m>$ che punta fuori da $\boldsymbol{P}$. Sia p il punto che si vuole verificare se è interno o esterno rispetto a $\boldsymbol{P}$.

Definiamo $x_i = \boldsymbol{n_i} \cdot (p - b_i)$ il prodotto scalare tra la normale della i-esima faccia e il vettore che congiunge il baricentro della faccia i al punto. Il punto è interno a $\boldsymbol{P}$ se e solo se:

$$x_i > 0, \forall i \in \{1, 2, \ldots, m\}$$

Se invece almeno uno dei prodotti scalari è negativo, allora il punto è esterno, il caso limite si verifica se $x_k = 0$: in questo caso il punto giace sulla superficie di f_k.

2.4 Punto visibile da una faccia

Questo concetto viene utilizzato solo nell'algoritmo quick-hull nel caso d-dimensionale, tassello fondamentale per le applicazioni di rendering 3d, nelle quali è importante non operare sulle facce delle mesh che non sono visibili, al fine di migliorare le prestazioni.

Una faccia f di un poligono convesso $\boldsymbol{P}$ viene definita visibile rispetto ad un punto p se esiste un segmento che congiunge p ad un punto qualsiasi contenuto in f senza intersecare altre facce, altrimenti tale faccia viene definita non visibile rispetto a p.

Ad un certo punto dell'algoritmo quick-hull ci si troverà ad avere un temporaneo inviluppo convesso che dovrà essere aggiornato, aggiungendo un punto che si trova all'esterno. Qui entra in gioco il concetto di questo paragrafo, perché bisognerà separare le facce dell'inviluppo convesso tra quelle visibili e non visibili dal punto, in modo tale da ottenere un nuovo politopo convesso che sarà il politopo precedente al quale verranno aggiunte delle nuove facce che avranno come vertice il punto esterno e $d - 1$ punti dell'inviluppo convesso precedente.

Per implementare nell'algoritmo quick-hull la definizione bisogna dapprima avere un solido convesso le cui normali delle facce puntino fuori da esso. Dato un punto p esterno ad un poligono avente una faccia f , la quale possiede un vettore normale $\boldsymbol{n}$ che punta fuori dal poligono, e un baricentro c, allora p è visibile da f se e solo se:

$$\boldsymbol{n}\cdot(\boldsymbol{p}-\boldsymbol{c})>0$$

Se tale risultato è uguale a zero allora vuol dire che tale punto giace sulla superficie di f e quindi, essendo complanare, è già incluso nell'inviluppo convesso.

2.5 Distanza tra un punto e un iperpiano

Una procedura che servirà per l'algoritmo quick-hull multidimensionale è la seguente: dato un iperpiano $\boldsymbol{\Sigma} \in \mathbb{R}^d$ di equazione $\boldsymbol{a} \cdot \boldsymbol{X} + b = 0$, dove $\boldsymbol{a}$ è un vettore ortogonale al piano, trovare la distanza da un punto $p=(p_1, p_2, \ldots, p_d) \in \mathbb{R}^d$ dall'iperpiano.
Si definisce quindi una retta passante per il piano $\boldsymbol{\Sigma}$ e per il punto p, tale retta avrà equazione parametrica $\boldsymbol{X} = t\boldsymbol{a} + p$. Allora il punto di intersezione della retta e del piano si può scrivere nella forma $p_0 = t_0\boldsymbol{a} + \boldsymbol{p}$, dove $t_0=\frac{\boldsymbol{a}\cdot p+b}{\|a\|}$, da cui:

$$p_0=\frac{-(\boldsymbol{a}\cdot p+b)}{\|a\|}+p$$

La distanza dal punto p dall'iperpiano $\boldsymbol{\Sigma}$ risulta essere:

$$d(p,\Sigma)=p-p_0=\frac{-(\boldsymbol{a}\cdot p+b)}{\|a\|}$$

Si ricorda che questo concetto è applicabile in un qualsiasi spazio d-dimensionale, pertanto per questo motivo la sua applicazione risulta adeguata nel caso dell'algoritmo dell'inviluppo convesso nel caso multidimensionale.

2.6 Il più piccolo poligono convesso

In questo paragrafo viene argomentato da quanti vertici e facce è composto il più piccolo poligono convesso in un generico spazio d-dimensionale. Tale concetto servirà per l'algoritmo quick-hull

multidimensionale per sapere quanti vertici utilizzare per creare un iniziale politopo convesso che sarà composto dal minor numero di vertici possibile.

Sia $\mathbb{R}^d$ un generico sottospazio d-dimensionale, Si definisce simplesso d-dimensionale il politopo d-dimensionale con il minor numero di vertici. Il simplesso d-dimensionale è composto da $d + 1$ vertici. Essendo un politopo, il simplesso d-dimensionale ha facce che sono a loro volta simplessi di dimensione $d - 1$.

Si noti che il simplesso di dimensione 1 è un segmento; quello di dimensione 2 è un triangolo; quello di dimensione 3 un tetraedro; quello di dimensione 4 un ipertetraedro, ecc.

Il simplesso d-dimensionale viene definito dai suoi vertici $[x_1,\ldots,x_{d+1}]$ ed è l'inviluppo convesso di un generico spazio euclideo $\mathbb{R}^d$. Per $n \leq d$, il numero delle facce n-dimensionali di un simplesso d-dimensionale è dato da tutti i possibili d diversi sottoinsiemi composti da n vertici ciascuno, ossia utilizzando il coefficiente binomiale:

$$n.facce = \binom{d+1}{n+1}$$

Nel nostro caso saremo interessati solo a sapere quante facce di dimensione $d - 1$ ci sono, quindi:

$$n.facce = \binom{d+1}{d} = d+1$$

Si può concludere quindi che il poligono convesso composto dal minor numero di punti in $\mathbb{R}^d$ è composto da $d + 1$ punti e $d + 1$ facce. Le facce sono composte a loro volta da lati che sono simplessi di dimensione $d - 1$.

3 Descrizione e analisi degli algoritmi

In questa sezione vengono presentati gli algoritmi per l'inviluppo convesso, dapprima in 2 dimensioni e successivamente nel caso d-dimensionale. Verranno in primis descritti gli algoritmi e i paradigmi utilizzati per la loro realizzazione, con una successiva implementazione in pseudocodice e un'analisi della loro correttezza e complessità, sia temporale che spaziale.

3.1 Graham-scan

L'algoritmo eredita il suo nome dal suo ideatore Ronald Graham, il quale pubblicò l'algoritmo nel 1972. Esso utilizza due tecniche di programmazione: la prima è chiamata ***metodo incrementale***, che consiste nel costruire l'inviluppo convesso un punto alla volta alla fine di ogni iterazione, dopo aver ordinato i punti in una determinata maniera rispetto ad un punto di riferimento, producendo una sequenza di punti dell'inseme di partenza $<p_1, p_2, ..., p_n>$. All'i-esima iterazione verrà generato l'inviluppo convesso dei punti $<p_1, p_2, ..., p_i>$. La seconda è chiamata ***backtracking***, nella quale l'algoritmo appena scopre di aver trovato una soluzione errata, "torna indietro", scartando l'ultimo vertice trovato e proseguendo la generazione dell'inviluppo convesso partendo con il vertice precedente a quello eliminato.

L'idea su si basa Graham-scan è che percorrendo un poligono convesso in senso antiorario tutte le coppie di segmenti consecutivi non effettuano mai svolte a destra.

Dato un insieme finito di punti P differenti, tale che $|P| \geq 3$, l'algoritmo Graham-scan produce l'inviluppo convesso di P, $\mathbf{CH}(P)$ in un tempo $O(n \log n)$, dove n è la cardinalità di P, producendo una lista di punti $<p_{i0}, p_{i1}, p_{i2}, ..., p_{im}>$ che sono i vertici dell'inviluppo convesso di P elencati in senso antiorario rispetto all'angolo polare composto con p_{i0} rispetto all'asse x positiva, ossia definita la funzione angolo polare tra due punti pol_ang(), i punti sono ordinati sulla base di:

$$pol\_ang\left(p_{i_0}, p_{i_k}\right), \ \forall k \in \left\{i_1, i_2, ..., i_m\right\}$$

L'elemento cardine dell'esecuzione di Graham-scan è uno stack s che avrà al suo interno i punti di un temporaneo inviluppo convesso, riferito al sottoinsieme di $P=<p_1, p_2, ..., p_i>$, $\mathbf{CH}(P_i)$ durante l'i-esima iterazione. Si vedrà in seguito che gli elementi contenuti in s in un determinato istante sarà la base della dimostrazione della correttezza.

Il primo passo dell'algoritmo consiste nel trovare in P uno dei due cosiddetti **punti estremi** rispetto alla coordinata x. Un punto $p_z=(p_{z1}, p_{z2}, \ldots, p_{zn}) \in \mathbb{R}^n$ contenuto in un inseme finito di punti P viene definito estremo rispetto ad un'asse t se:

$$(p_{zt} \leq p_{kt}) \vee (p_{zt} \geq p_{kt}), \forall\, p_k \in P, p_k \neq p_z$$

La determinazione di tale punto è semplice e si basa sulla definizione. Vedremo che anche per i successivi algoritmi il concetto di punto estremo è essenziale.

Definito p_0 il punto estremo a sinistra rispetto alla coordinata y di P più in basso, se molteplici punti condividono la stessa coordinata y più piccola si prende quello con la coordinata x minore. In pratica si sceglie p_0 selezionando l'elemento minimo della lista di punti iniziali usando come chiave (p_y, p_x), ossia ordinando lessicograficamente la tupla contenente la coordinata y come primo elemento e coordinata x il secondo. Il punto trovato farà sicuramente parte dell'inviluppo convesso.
Il secondo step dell'algoritmo consiste nell'ordinare i punti in maniera crescente rispetto all'angolo polare con p_0, come descritto nel paragrafo **2.1**, in modo tale da ottenere una sequenza di punti $<p_1, p_2, \ldots, p_q>$.
Si inseriscono poi in ordine p_0, p_1, p_2 nello stack s: a questo punto abbiamo formato il convex hull dell'insieme di punti $<p_0, p_1, p_2>$.
Per i restanti punti viene eseguito un ciclo che ad ogni i-esima iterazione crea l'inviluppo convesso di P_i.

La procedura in pseudocodice è illustrata di seguito.

Algoritmo 1: Graham_scan

Function Graham_scan(P)
1. Sia p_0 il punto estremo rispetto alla coordinata y più in basso e a sinistra di P
2. sia PO = $<p_1, p_2, \ldots, p_q>$ la lista di punti ordinati in base all'angolo polare con p_0, rimuovendo i vertici collineari con il punto di riferimento eccetto quello più distante
3. $s = stack(p_0, p_1, p_2)$
4. **for** i ← 3 **to** len(PO) **do**
5. **while** $d\_ang(s.next\_to\_top(), s.top(), p_i) \geq 0$ **do**
6. $s.pop()$
7. $s.push(p_i)$
8. **return** s

Innanzitutto bisogna definire le funzioni dello stack. Nella riga 3 viene generato un nuovo stack contenente il punto p_0 in basso, p_1 in mezzo e p_2 in cima, la funzione *top()* restituisce l'elemento più in alto nello stack senza rimuoverlo, *next_to_top()* restituisce l'elemento appena sotto a quello più in alto senza rimuoverlo, ed infine *push()* inserisce un elemento in cima allo stack e *pop()* rimuove l'elemento in cima allo stack e lo restituisce.

Per comprendere la guardia del while bisogna fare riferimento al paragrafo **2.2** dove verso la fine si parla della direzione nella quale svoltano due segmenti consecutivi. In questo caso si parla dei segmenti $\overline{s.next\_to\_top(),\ s.top()}$ e $\overline{s.top(),\ p_i}$. In pratica è richiesto che l'area con segno del parallelogramma generato dai due segmenti si maggiore o uguale a zero, ciò è equivalente a dire che la coppia di segmenti consecutivi effettua una svolta non a sinistra.

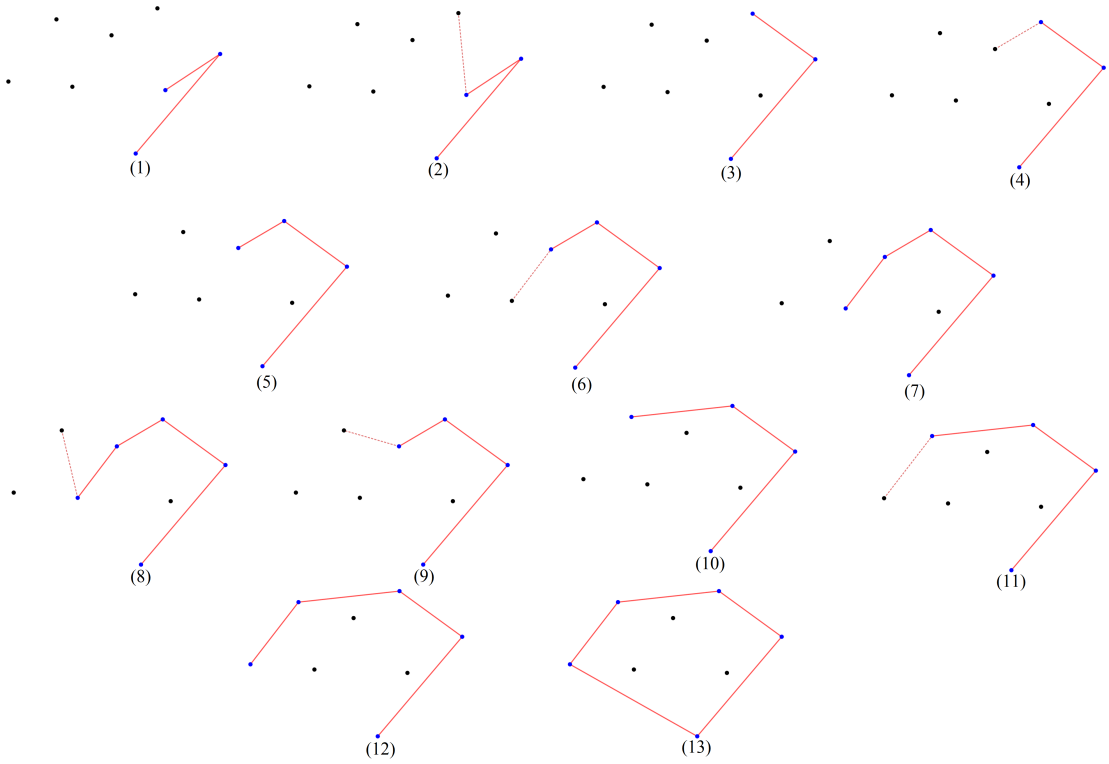

Figura 3.1 : Esempio di esecuzione di Graham_scan.

Correttezza di Graham_scan

Dimostrare la correttezza di Graham_scan significa dimostrare la correttezza della seguente affermazione:

lo stack ritornato dalla funzione Graham_scan contiene i punti dell'inviluppo convesso di un insieme di punti P, con $|P| \geq 3$*, ordinati in senso antiorario.*

Indichiamo con **CH**(P_i) l'insieme dell'inviluppo convesso dell'insieme di punti P_i, ossia $\langle p_1, p_2, \ldots, p_i \rangle$.

La dimostrazione si basa sull'invariante di ciclo, ossia una condizione che è valida sia all'inizio, sia durante e alla fine di un ciclo, in questo caso il ciclo di riferimento sarà il for alla riga 4.
L'invariante di ciclo utilizzata è la seguente:
all'inizio dell'i-esima iterazione lo stack s contiene i vertici di $\mathbf{CH}(P_{i-1})$ *in ordine antiorario.*

Prima di iniziare la dimostrazione dell'invariante di ciclo bisogna dimostrare che i punti p_0, p_1, p_2 fanno parte di $\mathbf{CH}(P_2)$, p_0 chiaramente farà sempre parte dell'inviluppo convesso di qualsiasi sottoinsieme di P, perché è il punto più in basso a sinistra. Se cosi non fosse allora vuol dire che esiste almeno un punto tale per cui $\mathbf{CH}(P)$ contiene al suo interno p_0, ma ciò può accadere solo se quell'altro punto si trova sotto p_0, andando in contraddizione con l'ipotesi di partenza che p_0 è il punto con coordinata y minore. È evidente poi che l'inviluppo convesso dei punti p_0, p_1, p_2 sono i punti stessi perché l'inviluppo convesso d-dimensionale è composto da minimo d + 1 punti e tali punti formano per forza un triangolo che in $\mathbb{R}^2$ è il poligono convesso composto dal minor numero di vertici. Un'eccezione si può verificare nel caso in cui i tre punti siano collineari, ma questa situazione non potrà mai verificarsi, poiché nella creazione della lista PO i punti collineari vengono rimossi ad eccezione di quello più distante da p_0.

Appurato che prima di iniziare il ciclo for lo stack contiene $\mathbf{CH}(P_2)$, allora l'invariante è soddisfatta all'inizio del ciclo. Ora bisogna provare che l'invariante di ciclo è valida all'inizio della i-esima iterazione. Sia p_k il punto in cima allo stack dopo l'esecuzione del ciclo while all'i-esima iterazione, ma prima che venga inserito p_i, allora in questo momento s contiene gli stessi vertici che conteneva terminata la k-esima iterazione, ossia $\mathbf{CH}(P_k)$. Una volta inserito nello stack p_i, s conterrà i vertici di $\mathbf{CH}(P_k \cup \{p_i\})$: se così non fosse allora l'angolo avrebbe effettuato una svolta non a sinistra.
Ci resta da dimostrare che s contiene gli stessi punti di $\mathbf{CH}(P_i)$. Consideriamo un punto p_j rimosso durante l'iterazione i del ciclo for. Si ha che l'angolo compreso tra p_k, p_j, p_i effettua una svolta non a sinistra. Se si considera il triangolo composto dai vertici p_k, p_i, p_0 allora è facile verificare che p_j è contenuto all'interno di tale triangolo, perciò p_j è un punto da scartare in quanto è all'interno di un triangolo contenuto nell'inviluppo convesso, pertanto non può far parte dell'inviluppo convesso di P. Applicando questo passaggio per tutti i punti da rimuovere durante l'i-esima iterazione si può ricavare una relazione del tipo:

$$\boldsymbol{CH}(P_i - \{p_j\}) = \boldsymbol{CH}(P_i), \forall\, p_j : d\_\text{angle}(p_{j-1}, p_j, p_i) \geq 0$$

Infine si deduce che $\mathbf{CH}(P_i - \{p_j\}) = \mathbf{CH}(P_k \cup \{p_i\}) = \mathbf{CH}(P_i)$.

Manca solo da dimostrare che alla fine del ciclo, *s* contiene i punti di **CH**(*P*) in senso antiorario, questa affermazione risulta essere valida avendo $i = q + 1$, per cui lo stack *s* conterrà i vertici di $\mathbf{CH}(P_q) = \mathbf{CH}(P_n)$, ossia **CH**(*P*), poiché l'inviluppo convesso rimane corretto anche senza i vertici che sono stati scartati nella riga 2, dove *n* è la cardinalità di *P*.

Complessità di Graham_scan

Scegliere il punto estremo più a sinistra rispetto alla coordinata *y* può essere effettuato tramite una semplice scansione della lista iniziale *P* in un tempo *O(n)*. Successivamente bisogna ordinare i punti della lista in senso antiorario rispetto all'angolo polare con p_0, operazione che può essere effettuata tramite i metodi di sorting asintoticamente efficienti in un tempo *O(n logn)*.

Tenendo conto che tutte le operazioni effettuate dallo stack vengono eseguite in un tempo costante *O(1)*, manca solo da valutare la complessità del ciclo for. È evidente che tutti i punti di *P* vengono inseriti nello stack una volta e una parte di essi vengo rimossi, rimane solo da capire quanti sono.

Sia $h = |\mathbf{CH}(P)|$ il numero di vertici dell'inviluppo convesso, quindi il ciclo for sarà composto da *n* push e $(n - h)$ pop, ossia avrà complessità $O(n) + O(n - h) = O(n)$, poiché i punti vengono rimossi al più una volta. Da ciò si deduce che l'algoritmo ha complessità temporale:

$$O(n)+O(n\log n)+O(n)=O(n\log n)$$

Per quanto riguarda la complessità spaziale si ha che la lista dei punti ordinati PO ha dimensione *O(n)*, mentre lo stack *s* ha al più dimensione *O(n)* nel caso, ad esempio, in cui tutti i punti di *P* facciano parte dell'inviluppo convesso, ossia *P* contenga i vertici di un poligono convesso, pertanto la complessità spaziale è *O(n)*.

In conclusione sia la complessità temporale che spaziale rimangono invariate asintoticamente indipendentemente da come sono distribuiti i punti nell'insieme di input, mentre in pratica con alcune distribuzioni particolari (quando un numero elevato di punti dell'insieme di partenza appartiene all'inviluppo convesso) si ottiene un leggero miglioramento delle prestazioni perché vengono effettuate meno rimozioni dei punti.

3.2 Jarvis-march

Algoritmo ideato da Ray A. Jarvis nel 1972, è una procedura per trovare l'inviluppo convesso di una sequenza di punti, che utilizza come paradigma principale il ***Prune-and-search***, ossia, viene creata una porzione dell'inviluppo convesso partendo da un vertice iniziale e procedendo in senso antiorario. Dopo aver creato la prima porzione dell'inviluppo convesso chiamata catena destra, si procede successivamente alla creazione della catena sinistra partendo dal vetrice più in alto della catena destra e finendo dal vertice in cui si è iniziato il processo di costruzione dell'inviluppo convesso.

L'algoritmo viene chiamato anche **gift wrapping algorithm**, in quanto tramite il *Prune-and-search* idealmente il convex hull viene generato "simulando" un impacchettamento con un foglio di carta dell'insieme dei punti di input.

Dato un insieme finito di punti differenti P con $|P| \geq 3$, Jarvis-march trova l'inviluppo convesso di P in un tempo $O(nh)$, dove n è la cardinalità dell'insieme di input e h il numero di vertici facenti parte di $\mathbf{CH}(P)$. Anche in questo caso viene ritornata una lista contenente i punti in maniera ordinata rispetto rispetto al punto estremo rispetto alla coordinata y più a sinistra in senso antiorario.

Nell'algoritmo di Jarvis si inizia prendendo i due punti estremi rispetto alla coordinata y chiamati p_l e p_h, ossia rispettivamente il punto più in basso e il punto più in alto di P (che chiaramente faranno sicuramente parte dell'inviluppo convesso).

A questo punto si inserisce p_l nella lista contenente i punti di $\mathbf{CH}(P)$.

Bisogna definire cosa si intenda per i termini menzionati all'inizio di questo paragrafo, "catena destra" e "catena sinistra":

per "catena destra" si intendono tutti i punti $p \in \boldsymbol{CH}(P)$ che stanno a destra del segmento $\overline{p_l p_h}$, analogamente per catena sinistra.

Da notare che sarebbe stato indifferente se si fossero presi il punto più a destra e il punto più a sinistra di P, allora in questo caso si sarebbe parlato di "catena inferiore" e "catena superiore".

Tramite due cicli vengono generate le due catene in questo modo:

per la catena destra si prende il punto p_c ossia il punto su cui operare, inizialmente p_l, e si aggiunge infondo alla lista di $\mathbf{CH}(P)$ il punto p_m tale per cui:

$$p_m = min_m \{ pol\_\text{ang}(p_c, p_m) \}$$

Il vertice p_c diventerà il punto appena trovato. Si procede la selezione del punto successivo di $\mathbf{CH}(P)$ con questo criterio fino a che $p_m = p_h$.

Invece per la catena sinistra è come prima, ma in questo caso l'angolo da verificare è rispetto all'asse x negativo e il ciclo si ferma quando $p_m = p_l$.

Infine viene ritornata la lista contenente i vertici di **CH**(P) meno l'ultimo elemento perché altrimenti ci sarebbe due volte p_l.

Lo pseudocodice è riportato qui di seguito:

Algoritmo 2: Jarvis_scan

```
Function Jarvis_scan(P)
1.  Siano p_l e p_h i punti estremi rispetto alla coordinata y
2.  hull ← list(p_l)
3.  index ← 1
4.  while hull[index] != p_h do
5.      Sia p_m il punto con angolo polare minore rispetto a hull[index]
6.      hull.add_last(p_m)
7.      index++
8.  while hull[index] != p_l do
9.      Sia p_m il punto con angolo polare negativo minore rispetto a hull[index]
10.     hull.add_last(p_m)
11.     index++
12. hull.remove(index)
13. return hull
```

La riga 2 crea una lista con un solo vertice p_l mentre *index* tiene traccia del punto sul quale si sta lavorando al momento. In questo caso il vertice p_c è rappresentato da *hull*[*index*]. Il contenuto del ciclo nella riga 4 e 8 crea rispettivamente la catena destra e sinistra, l'unica differenza tra i due cicli sono le righe 5 e 9: dove nella prima è richiesto di calcolare l'angolo polare rispetto all'asse x positiva, nella seconda rispetto all'asse x negativa. Alla fine si rimuove l'ultimo elemento della lista, che sarà p_l e si ritorna *hull* in modo tale da non avere tale punto due volte nella lista contenente l'inviluppo convesso.

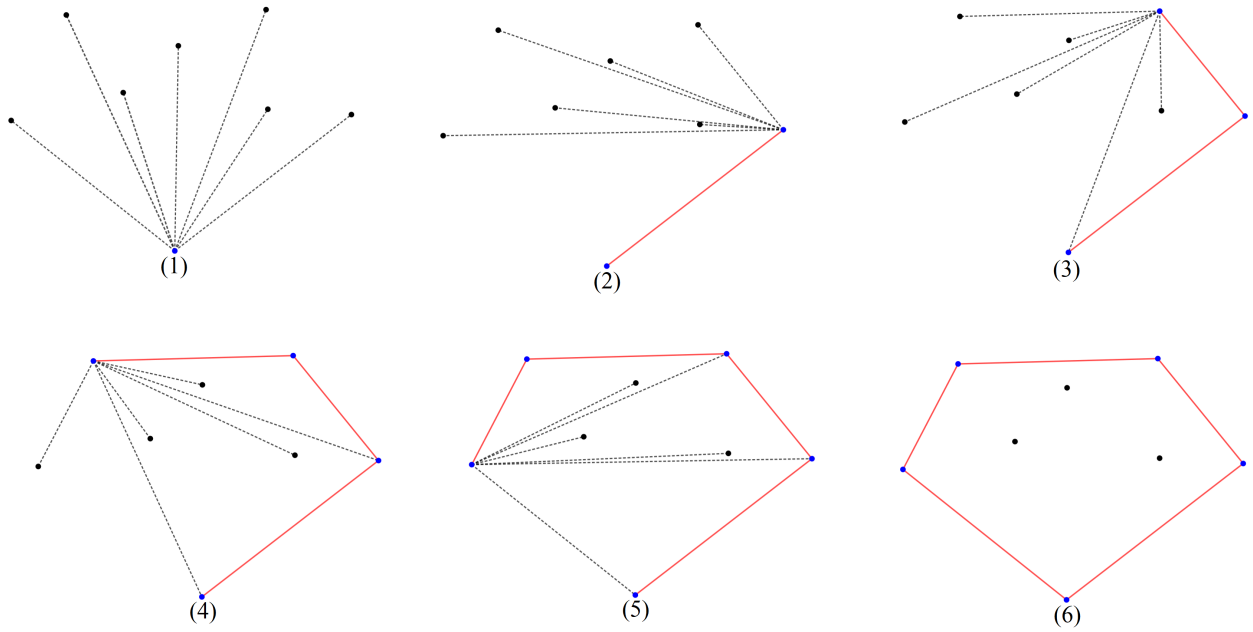

Figura 3.2 : Esempio di esecuzione di Jarvis_scan.

Correttezza di Jarvis-march

Come dimostrato precedentemente risulta evidente che dato un insieme di punti finito P, i due punti estremi rispetto alla coordinata y p_l e p_h fanno parte di **CH**(P), rimane da dimostrare che per ogni punto trovato, partendo da p_l, prendendo i punti con angolo polare minore rispetto al vertice di riferimento si ottiene un insieme uguale a **CH**(P).

Si procede nel dimostrare la correttezza della fase di costruzione della catena destra, per la catena sinistra il procedimento sarà analogo. Partendo dalla costruzione della catena destra fino al punto estremo più in alto rispetto all'asse y, siano p_i il punto su cui si sta cercando il vertice di P con angolo polare minore durante l'i-esima iterazione e p_{i+1} il vertice con angolo polare minore rispetto a p_i, ora bisogna dimostrare la correttezza di:

Dato il segmento $\overline{p_i p_{i+1}}$ *, non esiste alcun punto* p_k *di P, per cui il segmento* $\overline{p_i p_{i+1} p_k}$ *effettua una svolta a destra.*

Risulta evidente che prendendo p_{i+1} con angolo polare minore rispetto a p_i, l'angolo formato dal segmento $p_i p_{i+1} p_k$ è compreso nell'intervallo $[0,\pi)$, perché appena si raggiunge p_h il ciclo per la costruzione della catena destra termina. Quindi il segmento $\overline{p_i p_{i+1} p_k}$ effettua per forza una svolta non a sinistra, se così non fosse allora esiste un punto p_q che giace sulla destra del segmento $\overline{p_i p_{i+1}}$,

ma tale segmento avrebbe angolo polare con p_i minore rispetto a quello con p_{i+1} nel caso in cui $p_{qy} > p_{iy}$ e quindi all'i-esima iterazione l'algoritmo avrebbe scelto p_k al posto di p_{i+1}, invece se $p_{qy} < p_{iy}$ vuol dire che p_i non fa parte di **CH**(P) ma allora p_{i-1} non avrebbe scelto come vertice p_i in quanto p_q avrebbe angolo polare minore rispetto a p_{i-1} che è il punto facente parte dell'inviluppo convesso trovato durante l'iterazione precedente. In questo si giunge alla conclusione che vengono sempre effettuate svolte non a destra per i segmenti consecutivi. In più avendo preso il punto tale per cui l'angolo polare è minimo rispetto al vertice precedente, si è in grado di concludere che non possono esserci vertici a destra della catena destra.
Per quanto riguarda la catena sinistra il procedimento è analogo a quanto riportato precedentemente.

Complessità di Jarvis_march

La scelta dei punti estremi rispetto alla coordinata y viene effettuata tramite una semplice scansione della lista P e richiede un tempo $O(n)$, sia h il numero di vertici di **CH**(P), allora i cicli while, che per praticità potrebbero essere condensati in un unico ciclo, richiedono un tempo $O(nh)$, mentre tutte le operazioni sulla lista vengono eseguite in un tempo costante $O(1)$.
Per cui il tempo di esecuzione di Jarvis_scan è:

$$O(n)+O(nh)=O(nh)$$

Si può notare che, a differenza di Graham_scan, l'algoritmo di Jarvis non fornisce sempre le stesse prestazioni asintotiche, infatti è facile da dimostrare che se P è un insieme contenente i punti di un poligono convesso (ad esempio tutti i punti contenuti nella stessa circonferenza), quindi se $n = h$, l'algoritmo ottiene prestazioni quadratiche, per questo motivo Jarvis_scan appartiene a una famiglia di algoritmi detta **output-sensitive**.

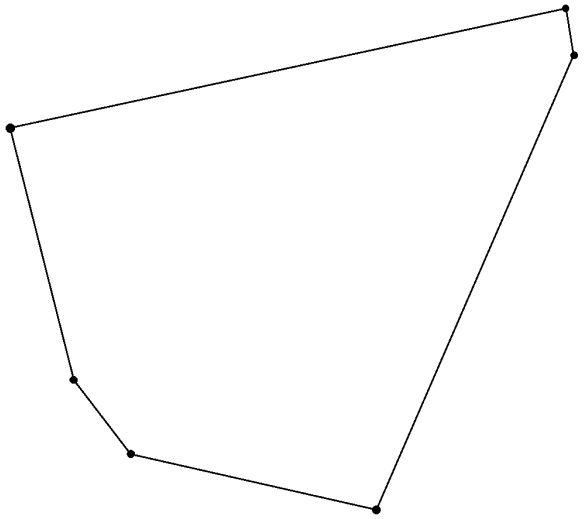

Figura 3.3 : esempio di worst-case di Jarvis_scan,
tutti i punti fanno parte di **CH**(*P*).

Per quanto riguarda l'occupazione in memoria, l'algoritmo oltre a p_l e p_h, mantiene la lista *hull* che conterrà i vertici di **CH**(*P*) in senso antiorario, tale lista conterrà solo i punti dell'inviluppo convesso, ossia avrà dimensione *O(n)*, il che vuol dire che nei casi più "fortunati" *hull* conterrà una piccola frazione dei vertici di *P*.

3.3 Quick-hull bidimensionale

Il più recente dei tre algoritmi per il convex hull in 2d presentati in questa tesi, ideato nel 1990 da Jonathan Scott Greenfield, utilizza come paradigma, la tecnica del ***divide-and-conquer***, ossia è basato nel ridurre ricorsivamente ad ogni passo la taglia del problema (fase di divide) fino ad arrivare ad un punto in cui la taglia è talmente piccola da poter risolvere il sottoproblema con una procedura elementare (caso di base). Nella fase di conquer si combinano le soluzioni ai sottoproblemi generati precedentemente dalla fase di divide.

Quick_hull è un algoritmo che si può facilmente estendere a più dimensioni, come vedremo in seguito. È importante specificare che la tecnica del *divide-and-conquer* viene utilizzata solo nel caso bidimensionale: il motivo di tale scelta implementativa sta nel fatto che in un generico spazio d-dimensionale eseguire determinate operazioni risulta più complesso rispetto al caso 2d in cui alcune operazioni sono più semplici da eseguire. In più, per quick-hull la ricorsione è facile da eliminare in quanto, come si potrà vedere, sarà una ricorsione di coda.

L'idea su cui si basa l'algoritmo Quick-hull è quella di creare l'inviluppo convesso attraverso un'espansione triangolare. In pratica l'inviluppo convesso viene generato partendo da un triangolo iniziale, avente come vertici punti dell'insieme di partenza, i cui lati possono a loro volta far parte di un altro triangolo avente come vertici i due estremi del lato di quel triangolo e un altro vertice dell'insieme di partenza. Un segmento in comune a due triangoli viene rimosso. La fine del processo di espansione si verifica quando non ci sono più vertici esterni al poligono generato. L'operazione di espansione triangolare produce l'inviluppo convesso di un insieme di punti finito. Tale concetto per via della sua facilità nell'estensione in più dimensioni fa si che l'algoritmo quick-hull possa essere adattato anche ad un contesto multidimensionale.

In questo caso l'algoritmo, come molti algoritmi ricorsivi, consiste in due procedure: la prima *quick_hull*, che rappresenta la prima invocazione del metodo e consiste nel preparare l'insieme di partenza P per la procedura ricorsiva *find_hull*.

La procedura inizia prendendo i punti estremi rispetto alla coordinata x p_l e p_r (anche in questo caso è indifferente scegliere la coordinata rispetto alla quale prendere i punti estremi). Tali punti, come già detto faranno sicuramente parte dell'inviluppo convesso e quindi dovranno essere inseriti nella lista contenente i punti di $\mathbf{CH}(P)$. Dopodiché si traccia una linea che congiunge i vertici p_l e p_r e si creano due liste disgiunte di punti: la prima contiene i punti che si trovano sopra il segmento $\overline{p_l p_r}$ la seconda i punti che si trovano sotto. Se un punto si trova sulla linea chiaramente non farà parte di nessuno dei due insiemi perché collineare e contenuto in un segmento i cui estremi fanno parte di un inviluppo convesso, pertanto si può scartare a prescindere. Per effettuare questa operazione è sufficiente verificare in che direzione ruota l'angolo $p_l p_r p$, dove p è un generico punto di P diverso da p_l e p_r, utilizzando la tecnica descritta a fine del paragrafo **2.2**. Alla fine di tale funzione si invocano due chiamate alla procedura *find_hull*, una riferita alla catena superiore del convex hull, l'altra riferita alla catena inferiore.

Per ciascuno dei due insiemi si trova il punto a massima distanza dal segmento di riferimento p_f, tale punto, che farà sicuramente parte di $\mathbf{CH}(P)$, viene inserito nella lista dei punti di $\mathbf{CH}(P)$ nella posizione successiva a quella del punto sinistro del segmento. A questo punto abbiamo formato un triangolo composto dai vertici $p_l p_r p_f$ e si creano tre insiemi, il primo s_1, contiene i punti che stanno all'interno del triangolo $p_l p_r p_f$, il secondo s_2, i punti che stanno alla sua sinistra, mentre il terzo s_3, quelli che stanno alla sua destra. Si procede nel trovare ricorsivamente i punti di $\mathbf{CH}(P)$ in s_2 e s_3, usando come segmento iniziale rispettivamente per l'insieme s_2 p_l e p_f, mentre per s_3 p_f e p_l, mentre è possibile scartare a prescindere i punti all'interno del triangolo $p_l p_r p_f$ perché sono punti interni ad un

sottoinsieme convesso di **CH**(P). Si procede nella ricorsione fino a che non ci sono punti rimasti fuori all'inviluppo convesso generato.

La procedura in pseudocodice è illustrata qui di seguito:

Algoritmo 3: Quick_hull2d

Function quick_hull(P)
1. Siano p_l e p_r i punti estremi rispetto alla coordinata x
2. *hull* ← list(p_l)
3. Sia s_1 l'insieme di punti che si trova sopra al segmento p_l e p_r
4. Sia s_2 l'insieme di punti che si trova sotto al segmento p_l e p_r
5. find_hull(s_1, p_l, p_r, *hull*)
6. *hull*.add_last(p_r)
7. find_hull(s_2, p_r, p_l, *hull*)
8. **return** *hull*

Function find_hull(s, p_l, p_r, *hull*)
1. **if** len(s) == 0 **then return**
2. Sia p_f il punto più lontano dal segmento $\overline{p_l p_r}$
3. Sia s_0 il sottoinsieme di s contenente i punti che stanno all'interno del triangolo $p_l p_f p_r$
4. Sia s_1 il sottoinsieme di s contenente i punti alla sinistra del triangolo $p_l p_f p_r$
5. Sia s_2 il sottoinsieme di s contenente i punti alla destra del triangolo $p_l p_f p_r$
6. find_hull(s_1, p_l, p_f, *hull*)
7. *hull*.add_last(p_f)
8. find_hull(s_2, p_f, p_r, *hull*)

Come primo appunto, si noti che nelle righe 5-6 della procedura *quick_hull*, l'ordine dei punti p_l e p_r è invertito, questo perché utilizzando la regola dell'area con segno del parallelogramma invertendo i punti si possono determinare i vertici sopra il segmento. In pratica è come se nella seconda chiamata l'insieme di punti ruotasse di 180° (quindi p_l si trova a destra e p_r a sinistra).

Nella riga 2 di *find_hull*, si prende il punto più lontano dal segmento, in modo tale da massimizzare la probabilità che un punto si trovi all'interno del triangolo $p_l\ p_f\ p_r$, effettuare una scansione dell'intero insieme s per trovare il punto con distanza maggiore dal segmento $p_l p_r$ risulta più efficiente che prendere un punto a caso, in più p_f è garantito che farà parte dell'inviluppo convesso e quindi è sufficiente inserirlo nella lista contenente i punti dell'inviluppo convesso nella posizione corretta. Infine nella riga 9 di *find_hull* si inserisce p_f in *hull* nella posizione successiva a p_l in modo tale da poter ritornare i punti ordinati in senso orario, chiaramente è possibile ritornare i punti in senso antiorario, come negli atri due algoritmi, con semplici modifiche nell'inserimento dei punti in *hull*.

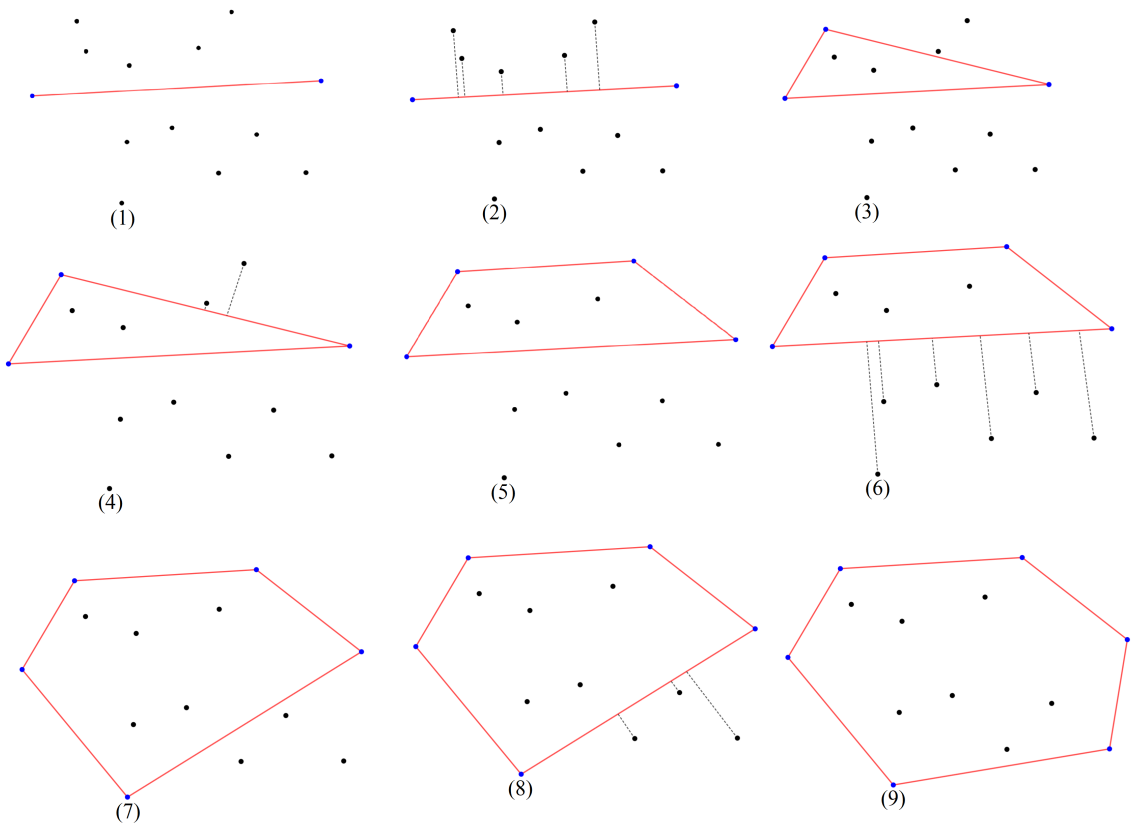

Figura 3.4: Esempio di esecuzione di quick_hull in 2d.

Correttezza di quick-hull2d

Prima di iniziare la dimostrazione è bene definire che cosa sono i vertici **primari** e **secondari**. Un vertice primario di un poligono è un vertice che unisce due lati di tale poligono, mentre altri punti del poligono che non sono primari sono secondari. È evidente che per definire un poligono sono necessari i vertici primari, mentre i secondari sono ridondanti e possono essere omessi.
La dimostrazione della correttezza procederà in tre parti:

(1) provare che l'output di *quick_hull* produce solo vertici primari di **CH**(P).

(2) provare che tutti i vertici primari di **CH**(P) fanno parte dell'output.

(3) Provare l'ordine dei vertici dell'output.

Iniziamo dimostrando l'affermazione (1). Si traccia una linea orizzontale l su cui giace almeno un punto di P e tale che tutti gli altri punti dell'insieme di partenza si trovano in basso. Risulta evidente che ogni punto su l fa parte di **CH**(P); tale retta viene chiamata retta di supporto. Da notare che se la retta contiene più di due punti, solo i due estremi sono vertici primari.

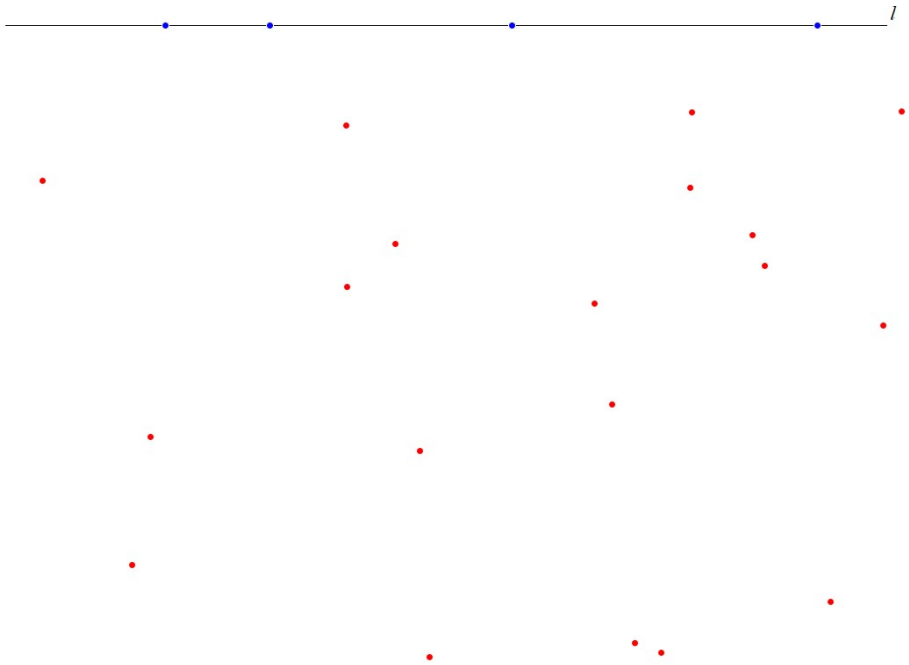

Figura 3.5: retta di supporto per un insieme di punti.

L'algoritmo procede prendendo i punti estremi rispetto alla coordinata x p_l e p_r (è già stato dimostrato che fanno parte di **CH**(P)).

Dopodiché viene invocato l'algoritmo *find_hull*(s_1, p_l, p_r, *hull*). Utilizzando l'induzione matematica sulla cardinalità di s_1, si può usare la seguente invariante su *find_hull*:

1. p_l e p_r sono vertici primari.
2. s_1 è un sottoinsieme di P e si trova a sinistra del segmento p_lp_r.
3. $s_2 = P - s_1$ si trova a destra del segmento p_lp_r.

La prova è la seguente. Per il caso base, se $|s_1| = 0$, è banale e l'algoritmo non restituisce alcun vertice.

Passo induttivo: consideriamo il caso in cui $|s_1| = n$. *find_hull* trova il vertice p_f con maggiore distanza dalla retta p_lp_r; ora sia l una retta parallela a $\overline{p_l p_r}$, è evidente che l è una retta di supporto per s_1.

Essendo che p_f giace su una retta di supporto allora p_f fa parte di **CH**(P) ed è un vertice primario.

Allora $|s_1 - \{p_f\}| = n - 1$, pertanto la cardinalità è sempre ridotta ad ogni chiamata ricorsiva.

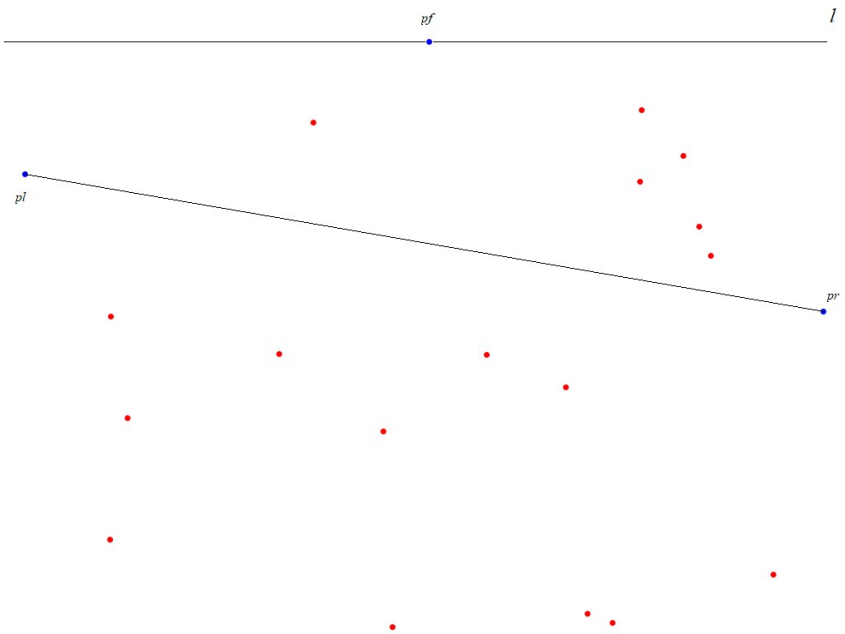

Figura 3.6: il vertice più distante dal segmento formato da p_l e p_r si trova sempre su una retta di supporto.

Bisogna dimostrare che l'invariante è mantenuta per le successive chiamate ricorsive find_hull(s_1, p_l, p_f, *hull*) e find_hull(s_2, p_f, p_r, *hull*). Per quanto riguarda 1), è evidente perché come già dimostrato i punti estremi (che saranno rispettivamente p_l, p_f e p_f, p_r) sono punti estremi dei sottoinsiemi di *s*. Per ciò che riguarda la 2), è banale poiché il nuovo sottoinsieme si trova sempre a sinistra della retta che congiunge uno dei punti estremi al punto più distante. Infine per la 3) si può dimostrare per assurdo, supponendo che ci sia un punto con coordinate appartenente all'insieme $s_1 - s_{11}$, dove s_{11} è l'insieme dei punti a destra di p_l p_f. Allora tale punto ha coordinate *x* e *y* minore di p_l, ma ciò non è possibile in quanto p_l è punto estremo. Il procedimento è analogo per l'insieme s_2.

Si procede ora a dimostrare (2), ossia che tutti i vertici primari di **CH**(*P*) fanno parte dell'output. Si utilizza l'induzione matematica: ad ogni chiamata di *find_hull* viene sempre ridotto l'insieme su cui si sta lavorando, in due modi:

- Nel caso in cui il punto è il più lontano dal segmento dei punti estremi, come già dimostrato tale punto appartiene all'output ad è un vertice primario.
- Nel caso in cui il punto è interno al triangolo formato durante una chiamata di *find_hull*, chiaramente il vertice non è primario e non viene restituito dall'output.

L'algoritmo termina solo quando tutti i vertici sono stati rimossi e l'output contiene solo punti del caso 1, quindi solo vertici primari.

Infine si prova (3), anche qui si procede per induzione. Per il caso base: se l'insieme è vuoto non viene ritornato nulla, e quindi l'affermazione è soddisfatta. Per il passo induttivo: le chiamate ricorsive a find_hull seguono un determinato ordine, pertanto è sufficiente inserire il punto più lontano dagli estremi in mezzo alle due chiamate ricorsive, in modo tale da garantire l'ordine prestabilito.

Complessità di quick-hull2d

Anche in questo caso si parla di un algoritmo output-sensitive, la cui complessità temporale dipende dall'insieme di input. È bene procedere nell'analisi valutando il caso peggiore e il caso medio.
Il caso peggiore è molto raro e non è detto che si verifichi sempre. Se tutti i vertici dell'insieme di partenza appartengono all'inviluppo convesso, e se durante la creazione di s_1 e s_2 uno è vuoto e l'altro contiene $|s| - 1$ punti, vengono eseguite il maggior numero possibile di chiamate a *find_hull*, operazione che peggiorerà le prestazioni in assoluto. Quello che si verifica è che ad ogni chiamata di *find_hull* uno dei due insiemi s_1 o s_2 è vuoto, per cui otteniamo un equazione di ricorrenza del tipo:

$$T(n)=\begin{cases}0 & se\ \ n=0\\ T(n-1)+O(n) & altrimenti\end{cases}$$

Dove $n = |P|$, ricordando che la ricerca di un punto e la suddivisione dell'insieme richiedono un tempo lineare. Si può risolvere tale ricorrenza utilizzando vari metodi, ad esempio l'unfolding che consiste nell'applicare ripetutamente la ricorrenza fino ad ottenere una relazione su *T(n)* non ricorsiva:

$$T(n)=T(n-1)+O(n)=T(n-2)+O(n)+O(n)=...=T(n-i)+iO(n)$$

Quando $i = n$ siamo al caso base, per cui:

$$T(n)=nO(n)=O(n^2)$$

Come per Jarvis_scan anche quick_hull ottiene prestazioni quadratiche al caso peggiore.
Il caso medio si verifica quando tutte le chiamate di *find_hull* creano i due sottoinsiemi s_1 e s_2 di dimensioni all'incirca pari,in questo caso si dice che l'esecuzione è **bilanciata**, ossia assumendo per semplicità n del tipo 2^i, si ottiene:

$$T(n)=\begin{cases}0 & se\ \ n=0\\ 2T(n/2)+O(n) & altrimenti\end{cases}$$

Si può evitare di risolvere direttamente tale ricorrenza, utilizzando il **Master-theorem**, in questo caso si è in grado di trovare l'andamento asintotico di T(*n*), che risulta essere:

$$T(n)=O(n\ \ logn)$$

L'andamento asintotico di quick-hull2d perciò non dipende solo da quanti punti fanno parte di **CH**(*P*) ma anche da quanto è bilanciata la sua esecuzione. In particolare, se *h* è il numero di vertici dell'inviluppo convesso e $h << n$ e l'esecuzione di quick-hull2d è bilanciata, si avranno *h* chiamate a *find_hull* ciascuna delle quali richiede un tempo non maggiore di *O(n)*. Si può considerare un altro caso in cui vengono rimossi la maggior parte dei vertici perché rientrano nel triangolo iniziale, allora rimangono *O(h)* vertici e assumendo i punti distribuiti uniformemente si avrà come complessità totale *O(n + h logh)*. In genere se i punti sono generati secondo una distribuzione circolare si può assumere $h = O(n^{1/3})$, per cui in questo caso fortunato si avrà una complessità *O(n)*. Nella pratica dato che le prestazioni dipendono non solo da *h* ma anche da come sono distribuiti i punti nell'insieme iniziale si considera come caso medio *O(n logn)* e caso peggiore $O(n^2)$.

Tali equazioni di ricorrenze ricordano il noto algoritmo di ordinamento quick_sort. In effetti i due algoritmi hanno molto in comune, prima di tutto condividono la complessità al caso medio e peggiore e in più condividono la causa del loro worst-case, dovuta al caso in cui uno dei due sottoinsiemi su cui eseguire le chiamate ricorsive sia vuoto e l'altro contiene i rimanenti punti.

Per quanto riguarda la complessità spaziale, si ha che ad ogni chiamata di *find_hull* si generano tre insiemi con dimensione *O(n)*, se $h = |\mathbf{CH}(P)|$ si hanno *O(h)* chiamate, per cui la complessità spaziale è *O(n)*.

Algoritmo	**Complessità al caso medio**	**Complessità al caso peggiore**	**Complessità spaziale**
Graham_scan	*O(n logn)*	*O(n logn)*	*O(n)*
Jarvis_scan	*O(nh)*	$O(n^2)$	*O(n)*
Quick_hull2d	*O(n logn)*	$O(n^2)$	*O(n)*

Tabella 3.1: Tabella riassuntiva delle prestazioni degli algoritmi per l'inviluppo convesso in due dimensioni.

3.4 Quick-hull d-dimensionale

L'ultimo paragrafo di questo capitolo tratterà un'estensione dell'algoritmo quick_hull nel caso di un generico spazio d-dimensionale, con $d > 1$. L'algoritmo venne sviluppato da C. Bradford Barber, David P. Dobkine e Hannu Huhdanpaa nel 1996.

In questo caso l'output dell'algoritmo non saranno più i vertici dell'inviluppo convesso ma saranno ritornate le facce, o meglio i vertici che compongono le facce, che a loro volta sono composte da lati che confinano con un'altra faccia del poligono.

Come già detto, per la versione multidimensionale di quick_hull si rinuncia all'approccio *Divide-and-conquer*, sostituendo la chiamata a *find_hull* con un ciclo while.

Come vincolo statico è richiesto solo che l'insieme dei punti di input P sia composto da almeno $d + 1$ punti differenti tra di loro e non complanari, perché come spiegato nel paragrafo **2.6**, il numero minimo di punti necessari a definire un poligono convesso in uno spazio d-dimensionale è proprio $d + 1$.

Innanzitutto si definisce la classe **ridge** che rappresenta i bordi delle facce. Tale oggetto conterrà i punti che lo compongono e sarà dotato di una funzione in grado di compararlo con altri ridge. Bisogna prestare attenzione per il confronto perché può capitare che due ridge uguali abbiano memorizzato i punti in un ordine differente. In pratica i bordi di una faccia sarebbero, in 2d, un punto e una faccia è composta da due bordi (ovvero è un segmento), in 3d, un segmento e una faccia è composta da tre bordi, eccetera.

L'altra classe fondamentale per l'algoritmo è la classe **facet**, che rappresenta le facce dell'inviluppo convesso, tale oggetto sarà definito tramite i punti che lo compongono, il baricentro, un vettore normale (che punterà fuori dal convex hull), una outside_list che rappresenta i punti visibili da tale faccia esternamente, ed i suoi bordi. L'oggetto facet deve quindi essere dotato di un metodo per definire la outside_list e un metodo di comparazione.

Definite queste due classi, si può proseguire con l'implementazione in pseudocodice:

Algoritmo 4: Multidimensional_quick_hull

Function mq_hull(*P*)
1. dim ← len(P_0)
2. sia *hull* un simplesso composto da *d + 1* punti di *P*
3. sia *outside* la lista dei punti fuori da *s* (da non confondere con le outside_list delle facce)
4. **for** *facet* in *hull* **do**
5. *facet*.set_outside_list()
6. **while** len(*outside*) != 0 **do**
7. facet ← $\text{argmax}_{f \in S}\{\text{distance}(f, p)_{p \in f.outside\_list}\}$
8. sia p_f il punto contenuto in *facet*.outside_list con distanza euclidea maggiore da *facet*
9. *invisibles* ←∅
10. **for** *f* in *hull* **do**
11. **if not** is_visible(*hull*, *f*) **then** *invisibles* = *invisibles* ∪ {*f*}
12. *visibles* ← *hull – invisibles*
13. *horizon* ← compute_horizon(*visibles*, *invisibles*)
14. *New_faces* ← ∅
15. **for** (p_1, p_2) in *horizon* **do**
16. *f* ← facet(p_1, p_2, p_f)
17. *New_faces* = *New_faces* ∪ {*f*}
18. *hull* = *New_faces* ∪ *invisibles*
19. ricalcolo *outside*
20. **for** *facet* in *hull* **do**
21. *facet*.set_outside_list()
22. **return** *hull*

Si inizia prendendo la dimensione del primo punto di *P* (si suppone che tutti i punti abbiano la stessa dimensione), si procede creando un simplesso iniziale di dimensionalità appropriata che sarà il primo inviluppo convesso. Ovviamente è consigliabile prendere il simplesso iniziale più grande possibile, in modo tale da aumentare la probabilità che un punto si trovi all'interno di tale simplesso, così da scartarlo. Il metodo proposto per la creazione è il seguente:

1. Si prendono i punti estremi rispetto ad ogni coordinata.
2. Si calcola la distanza euclidea dei due punti estremi per ciascuna coordinata. Tali punti saranno p_1 e p_2 e si inseriscono in un simplesso iniziale *s*, così da creare un simplesso monodimensionale.
3. Si trova il punto più distante da *s* e lo si aggiunge ad *s*.
4. se *len(s) = d + 1* si termina, altrimenti si torna al punto 3.

Nella pratica calcolare il punto più distante da un politopo che cambia dimensionalità ad ogni iterazione può risultare un'operazione parecchio dispendiosa, poiché bisognerebbe calcolare per ogni faccia del poligono il punto più distante, quindi si può utilizzare un metodo meno preciso ma

più veloce, che consiste nel calcolare il baricentro dei punti che compongono il politopo e trovarne il punto più distante. Successivamente si calcolano i punti fuori dal simplesso tramite il metodo proposto nel paragrafo **2.3**, e si crea la outside_list per ogni faccia del temporaneo inviluppo convesso. Queste due operazioni verranno effettuate alla fine di ogni iterazione del ciclo. La guardia del while controlla che la lista *outside*, che contiene i tutti i punti fuori dall'inviluppo convesso,sia vuota, se così non fosse inizia una nuova iterazione del ciclo, in cui inizialmente viene presa la faccia di *hull* tale per cui è massima la distanza con uno dei punti della sua outside_list, e si salva tale punto. Si sceglie di eseguire questa operazione in modo tale da massimizzare la probabilità che aggiornando l'inviluppo convesso, il maggior numero di punti venga inglobato dal convex hull durante la creazione delle nuove facce. A questo punto si crea l'insieme delle facce visibili e non visibili: chiaramente gli insiemi *visbles* e *invisibles* sono disgiunti, e vale che *visbles* + *invisibles* = *hull*, da cui si ricava *visibles* = *hull* – *invisibles*. Bisogna ora calcolare l'orizzonte, che consiste nei bordi delle facce visibili che si dovranno attaccare a p_f in modo tale da creare le nuove facce dell'inviluppo convesso. Nella pratica l'orizzonte è composto dai bordi in comune tra le facce visibili e non visibili da p_f, meno formalmente è la regione di confine tra le facce visibili e non visibili da p_f. Sia $\mathbf{F_i} \in$ *visibles* e sia $\mathbf{F_j} \in$ *invisibles* ciascuna delle quali contiene dei bordi $\mathbf{E_{ix}}$ e $\mathbf{E_{jx}}$, allora l'insieme dell'orizzonte è definito come:

$$horizon(p_f, hull) = E_{kz} : E_{kz} \in F_i, E_{kz} \in F_j$$

Una volta calcolato l'orizzonte, si creano le nuove facce da aggiungere, generate dai vertici dei bordi dell'orizzonte unite a p_f. Il nuovo convex hull sarà ottenuto come l'unione dell'insieme delle facce non visibili e le facce appena create, ossia:

$$hull = New\_faces \cup invisibles$$

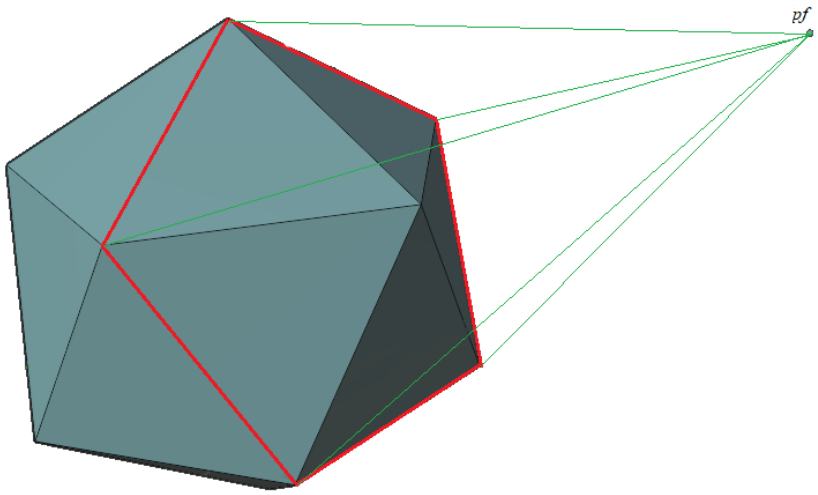

Figura 3.7: in rosso l'orizzonte per il punto p_f.

Alla fine del ciclo si calcolano l'outside set di *hull* e si ricalcolano le outside_list per ciascuna faccia. Il ciclo termina quando non ci sono più punti fuori da *hull*.

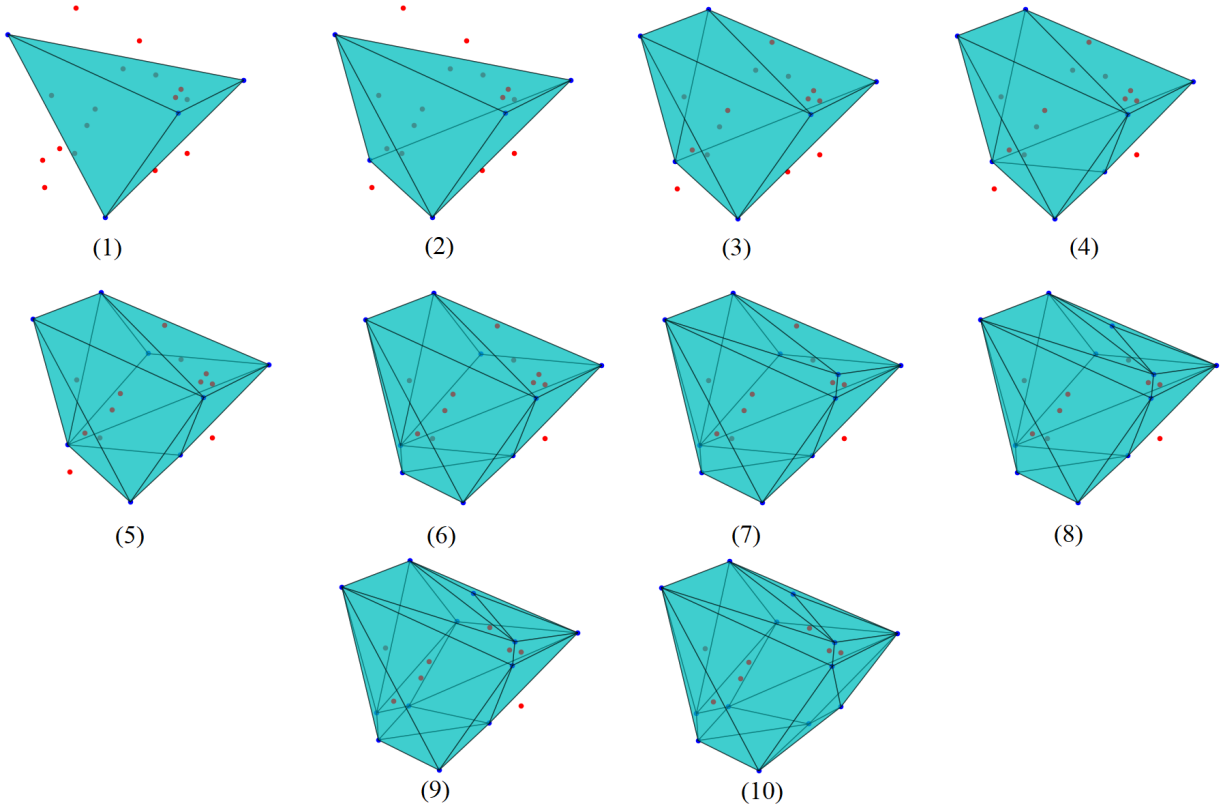

Figura 3.8: Esempio di esecuzione di quick_hull su un insieme di punti tridimensionali.

Correttezza di quick-hull multidimensionale

Avendo già dimostrato la versione ricorsiva di quick_hull in due dimensioni, bisogna avere solo alcuni accorgimenti per la conversione della versione multidimensionale. In particolare è come se il ciclo sostituisse le funzioni *find_hull*, con la differenza che le facce interessate vengono selezionate in base ad un criterio e non in modo "lineare" come avviene nell'algoritmo *D&C*.
Bisogna quindi dimostrare la correttezza di questa affermazione:
multidimensional_quick_hull produce l'inviluppo convesso di un insieme di punti finito in $\mathbb{R}^d$.
L'algoritmo inizia con una creazione di un simplesso d-dimensionale composto da d + 1 vertici. Successivamente, dopo aver creato l'insieme dei punti che si trovano fuori dal simplesso, viene selezionato il vertice più distante da tale politopo. Tale punto farà sicuramente parte dell'inviluppo

convesso. Avendo già dimostrato in due dimensioni che preso un punto con questo criterio, esso appartiene all'inviluppo convesso, per il caso d-dimensionale è uguale, l'unica differenza è che se in 2d si parla di retta di supporto, in questo caso si parlerà di iperpiano di supporto.
Successivamente il punto viene processato in base al seguente teorema:
Sia **CH**(*P*) un inviluppo convesso in $\mathbb{R}^d$ di un insieme di punti finito *P*, e *p* un punto esterno a **CH**(*P*), non facente parte di *P*. Allora la faccia *f* appartiene a **CH**(**CH**(*P*) ∪ {*p*}) se e solo se:

1. *f* è una faccia di **CH**(*P*), e *p* non è visibile da *f*.
2. oppure *f* non è una faccia di **CH**(*P*), ed è composta da *p* ed i vertici di un bordo di un'altra faccia di **CH**(*P*) con una faccia incidente che non vede *p* e l'altra faccia incidente che vede *p*.

Nel primo caso l'algoritmo mantiene la facce, nel secondo caso costruisce delle nuove facce. La procedura che crea delle nuove facce utilizzando il concetto di orizzonte garantisce la convessità del politopo anche se *p* è visibile da più di una faccia.
Si continua con questo metodo fino a che non rimangono punti esterni a **CH**(*P*).

Complessità di quick-hull multidimensionale

Prima di dimostrare la complessità di multidimensional_quick_hull è bene definire un concetto fondamentale per valutarne il suo andamento. Sia *d* la dimensione del sottospazio in cui viene eseguito multidimensional_quick_hull, *n* il numero di punti dell'insieme *P*, *h* il numero di punti dell'inviluppo convesso e $f_h = O\left(h^{\lfloor d/2 \rfloor} / \lfloor d/2 \rfloor !\right)$ il numero massimo di facce per *h* vertici. Allora un'esecuzione di multidimensional_quick_hull si dice bilanciata se:

- il numero medio di facce alla i-esimo punto processato è $\frac{d f_i}{i}$.
- il numero medio di punti partizionati per l'i-esimo punto partizionato è $\frac{d(n-i)}{i}$.

La prima cosa che si può notare che la complessità crescerà non solo in funzione di *n* ma anche in funzione di *d*, in quanto a parità di punti dell'insieme di partenza aumentando la dimensione aumentano anche il numero delle facce.

Considerando un'esecuzione bilanciata di multidimensional_quick_hull in uno spazio d-dimensionale, la complessità di tale algoritmo è:

$$T(n,d)=\begin{cases} O(n\log h) & se\ \ d\le 3 \\ O(nf_h/\ \ h) & se\ \ d>3 \end{cases}$$

Bisogna considerare innanzitutto il costo di aggiungere un punto all'inviluppo convesso, tale lavoro è composto da una fase in cui si generano le nuove facce, ossia $O(d^3)$ ciascuna, il numero di facce create è funzione di *fr*, quindi il costo totale di aggiunta di un punto è $O\left(d^3\sum_{i=1}^{h}\frac{df_i}{i}\right)=O(f_h)$.

Il costo di partizionare un punto risulta essere $O\left(d\sum_{i=1}^{h}\frac{d^2(n-i)f_j}{i^2}\right)=O(d^3n\sum_{i=1}^{h}i^{\lfloor d/2\rfloor-2}/\lfloor d/2\rfloor!)$ ed è proporzionale al numero delle nuove facce, se $d \le 3$ la somma risulta essere uguale a $O(n\ logh)$, altrimenti $O(n\ f_h/h)$.

4 Confronto algoritmi per l'inviluppo convesso

In questo capitolo verrà riportata la parte operativa della tesi, nella quale si andranno a confrontare sotto diversi aspetti gli algoritmi per l'inviluppo convesso.
Si inizierà con un primo confronto degli algoritmi in due dimensioni sulla base di diversi dataset che modificheranno ad ogni test il numero di punti facenti parte del convex hull. In seguito si studierà l'andamento di quick_hull nel caso multidimensionale al variare della dimensione, per verificare la dipendenza della dimensione per quanto riguarda la complessità temporale di tale algoritmo.

La generazione dei dataset P per le varie prove consisterà nel creare i punti all'interno di un' ipersfera di raggio unitario in $\mathbb{R}^d$ centrata nell'origine (in 2d si tratterà di un cerchio), definito un parametro α che indica la percentuale di punti di P che giacciono sulla superficie dell'ipersfera generata, si avrà quindi che $\alpha\ n$ punti saranno sulla superficie di tale sfera, mentre $(1 - \alpha)\ n$ si troveranno all'interno di tale sfera, ossia per $(1 - \alpha)\ n$ punti si avrà $x_1^2+x_2^2+...+x_d^2 < 1$ mentre per $\alpha\ n$ punti $x_1^2+x_2^2+...+x_d^2 = 1$. I punti generati sulla superficie della circonferenza in 2d saranno distribuiti uniformemente in modo tale che tra un punto e il successivo ci sia un angolo pari a $2\pi / \alpha n$, così da minimizzare la probabilità che un punto interno al cerchio faccia comunque parte dell'inviluppo convesso.

I vari test saranno effettuati su una macchina avente come processore un i5 8400 e 8 gb di memoria ram, utilizzando algoritmi scritti in linguaggio di programmazione Python 3.9.13.

4.1 Confronto degli algoritmi in due dimensioni

Prima di iniziare il confronto vero e proprio è bene fare alcune considerazioni sull'andamento degli algoritmi in 2d trattati, in particolare per quanto riguarda gli algoritmi output-sensitive, in modo tale da scegliere i valori di α più appropriati.

Per graham_scan, che ha prestazioni $O(n \log n)$, c'è poco da dire in quanto tale algoritmo mantiene sempre le stesse prestazioni asintotiche indipendentemente dall'insieme di punti iniziale, anche se come vedremo ci saranno ugualmente variazioni delle prestazioni (non asintotiche) al variare del numero di vertici facenti parte dell'inviluppo convesso.

Jarvis_march è sicuramente un algoritmo che modifica in maniera importante le sue prestazioni in base all'insieme di partenza, avendo andamento asintotico $O(nh)$, dove h è il numero di punti facenti parte dell'inviluppo convesso. Le sue prestazioni sembrerebbero essere più efficienti di graham_scan nel caso in cui h sia molto limitato, in effetti nel caso dell'andamento di graham_scan si sta parlando di un logaritmo in base 2 ricavato da un ordinamento, per cui bisogna trovare un valore iniziale di α tale per cui h si avvicini il più possibile al $log_2 n$, in modo tale da pareggiare, per quanto possibile, le prestazioni dei due algoritmi.
Infine per quick_hull, che anch'esso presenta prestazioni quadratiche al caso peggiore, ciò dipende sia dal numero di vertici facenti parte del convex hull che in maniera più importante dalla loro distribuzione, il caso in cui tale algoritmo peggiora le sue prestazioni è quando esegue un' esecuzione non bilanciata, in cui a volte può degradare anche in prestazioni quadratiche, ciò rende più complesso confrontare la sua complessità rispetto agli altri due algoritmi.

I test effettuati verranno eseguiti con $\alpha=<0.01, 0.1, 0.25, 0.5>$ e successivamente verranno testate distribuzioni su casi particolari. È importante sottolineare il fatto che il dataset deve contenere un numero elevato di punti, per due motivi principali: innanzitutto, perché in un dataset di piccole dimensioni ci sarebbero differenze troppo piccole in termini di tempo di esecuzione tra gli algoritmi; in secondo luogo, aumentando il numero di punti del dataset si diminuisce la probabilità che un punto interno alla circonferenza faccia parte dell'inviluppo convesso.

I confronti iniziali al variare di α verranno eseguiti solo tra gli algoritmi graham_scan e quick_hull, poiché l'algoritmo di Jarvis ottiene prestazioni quadratiche, ossia $O(\alpha n^2)$ e con un numero elevato di vertici del dataset le prestazioni risulteranno notevolmente peggiori anche con valori di α piccoli. Successivamente verranno analizzati tutti e tre gli algoritmi cercando di enfatizzare le situazioni in cui ottengono prestazioni migliori.

Confronto al variare di α

Iniziando a valutare le prestazioni dei due algoritmi con $\alpha=0.01$, ossia circa l'1% dei vertici del dataset farà parte dell'inviluppo convesso, come si può vedere dal grafico avente in ascissa il numero di punti del dataset e in ordinata il tempo di esecuzione in secondi, si ottiene prestazioni leggermente migliori con quick_hull.

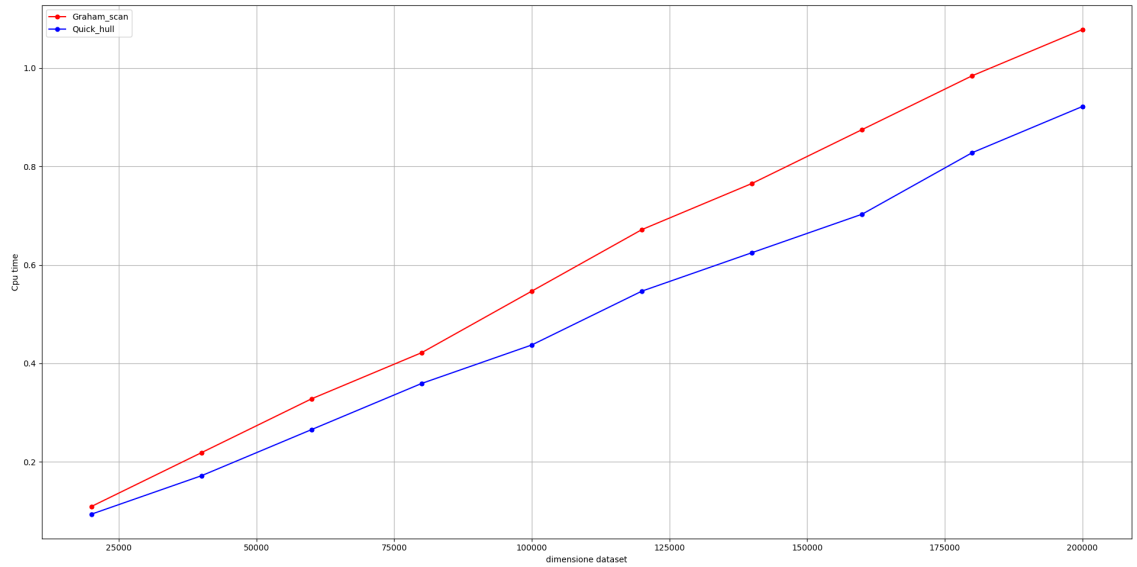

Figura 4.1: esecuzione con α=0.01.

Il motivo del fatto che quick_hull risulta più efficiente rispetto a graham_scan è perché tale algoritmo, in questo caso in cui è possibile controllare $|\mathbf{CH}(P)|$, risulta avere prestazioni $O(n + \alpha n \log(\alpha n)) = O(\alpha n \log(\alpha n))$ per cui come andamento asintotico è lo stesso rispetto a graham_scan ma con coefficienti moltiplicativi minori.

Aumentando il valore di α a 0.1 si può notare un notevole peggioramento delle prestazioni di quick_hull: non c'è da stupirsi, poiché la natura output-sensitive di tale algoritmo gli impone peggioramenti di performace all'aumentare della percentuale di punti facenti parte dell'inviluppo convesso, perché diminuiranno parallelamente il numero di vertici da scartare.

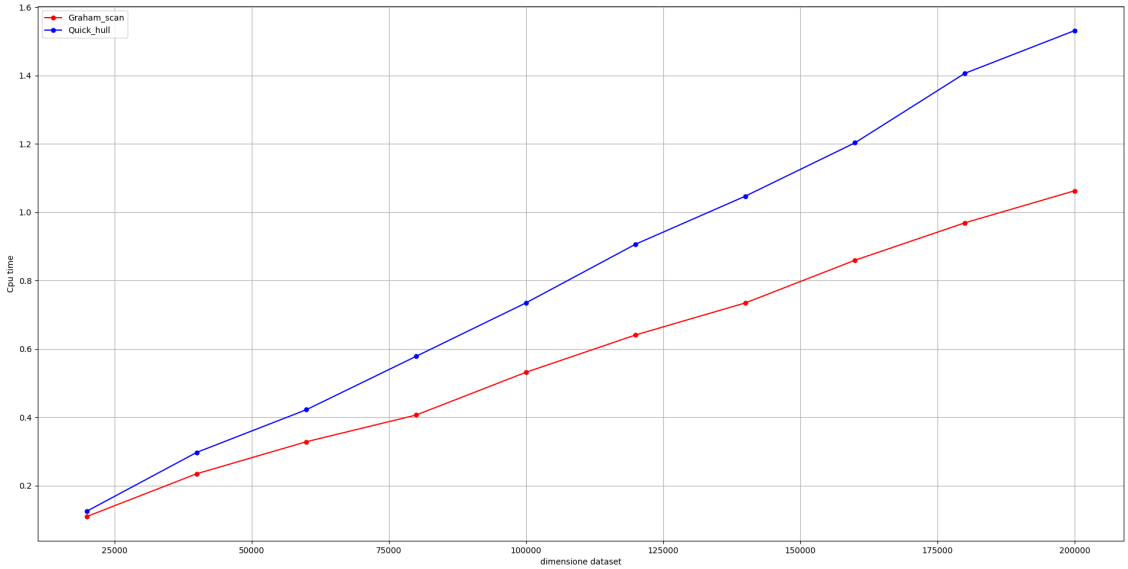

Figura 4.2: esecuzione con α=0.1.

Si può notare che ad ogni modo per dataset di piccole dimensioni (sotto ai 25000 punti) le prestazioni rimangono comunque ottime per entrambi gli algoritmi, in più per qualsiasi cardinalità dell'insieme di punti iniziale la differenza non è significativa: ad esempio con 200000 punti vi è una differenza di circa 450ms.

Le differenze risultano essere significative dal momento in cui α è pari a 0.25, ossia il 25% dei punti farà parte dell'inviluppo convesso. Un valore di α così elevato è sicuramente innaturale nel caso di punti generati casualmente in una distribuzione circolare, poiché in una distribuzione del genere i punti facenti parte dell'inviluppo convesso sono circa $n^{1/3}$ e con dataset di queste dimensioni $n^{1/3} << 0.25n$.

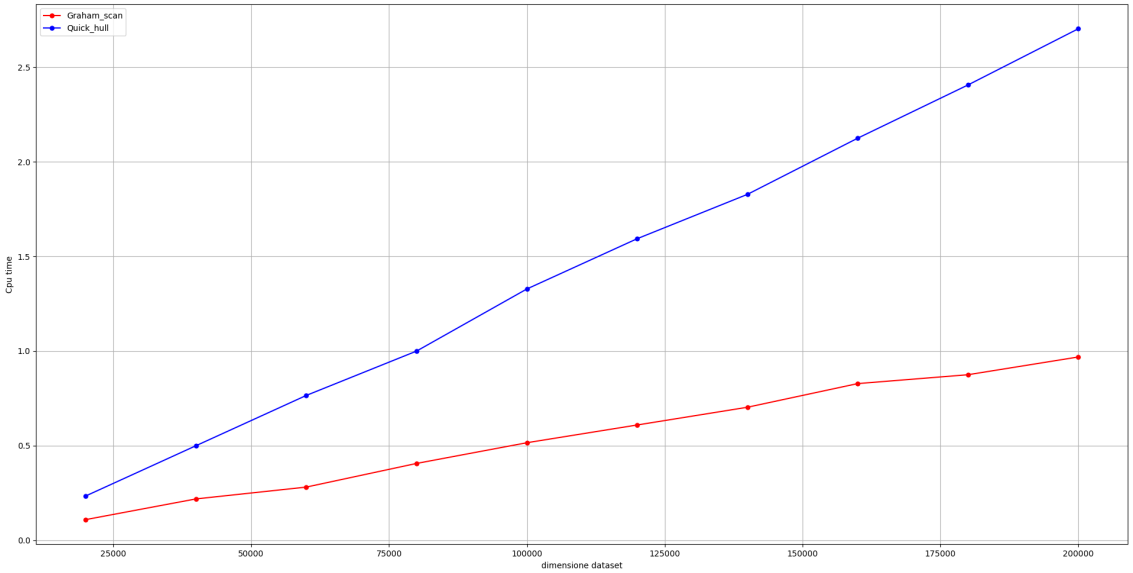

Figura 4.3: esecuzione con α=0.25.

In questo caso la differenza diventa abbastanza importante, specialmente con dataset di taglia elevata. Una cosa importante da notare è che in questo caso l'algoritmo di Graham sembrerebbe paradossalmente aver migliorato le sue performance in quanto non è mai andato sopra 1s di tempo di esecuzione, il motivo in realtà è molto semplice e verrà spiegato successivamente.

L'ultimo test viene effettuato con metà dei vertici facenti parte dell'inviluppo convesso, un caso estremo che non si verifica praticamente mai in caso di distribuzioni circolari con dataset di grande dimensione. La cosa più importante è che entrambi gli algoritmi mantengono per tutta l'esecuzione andamento *O(n logn)*, gli unici elementi che variano sono le costanti moltiplicative di tali algoritmi: quick_hull presenta un elevato peggioramento delle prestazioni all'aumentare del numero di punti facenti parte del convex hull, mentre graham_scan presenta un lieve miglioramento all'aumentare di α.

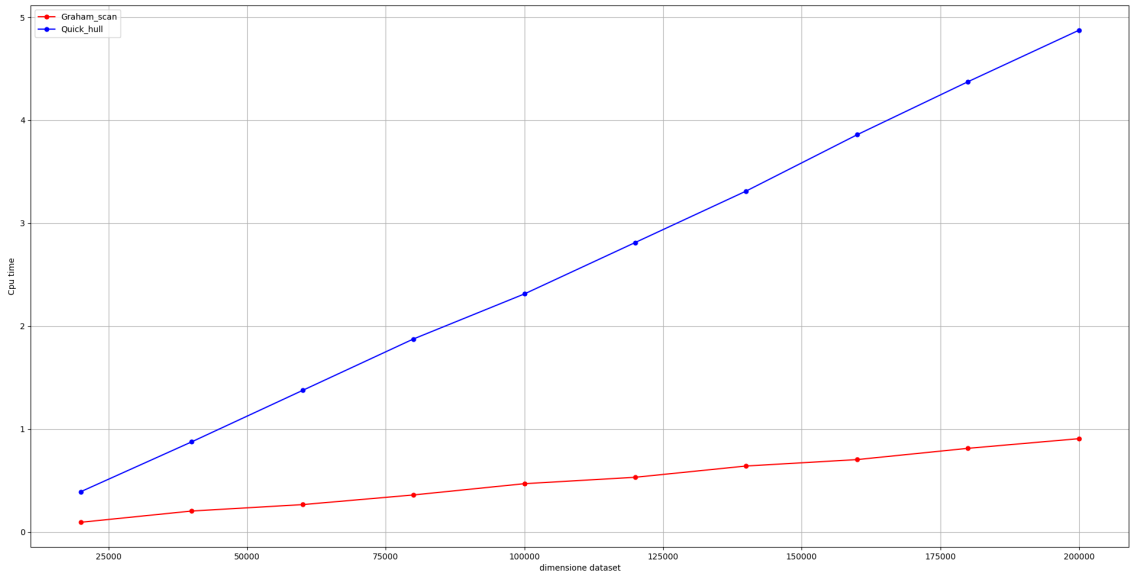

Figura 4.4: esecuzione con α=0.5.

In questo caso la differenza di performance è considerevole anche con dimensioni del dataset più piccole.

Analisi delle prestazioni di Jarvis_scan

L'algoritmo di Jarvis non è stato confrontato con gli altri due perché con un elevato numero di vertici dell'inviluppo convesso, ottiene prestazioni pessime. Tutto ciò non vuol dire assolutamente che Jarvis_scan non sia un algoritmo utile, anzi in alcune situazioni ottiene prestazioni nettamente migliori degli altri due algoritmi. Da quello che si deduce dall'analisi asintotica, Jarvis_scan presenta un tempo di esecuzione molto limitato nel caso in cui il numero di punti facenti parte del convex hull sia ridotto, nel caso in cui tale numero è costante allora l'algoritmo ottiene prestazioni lineari.

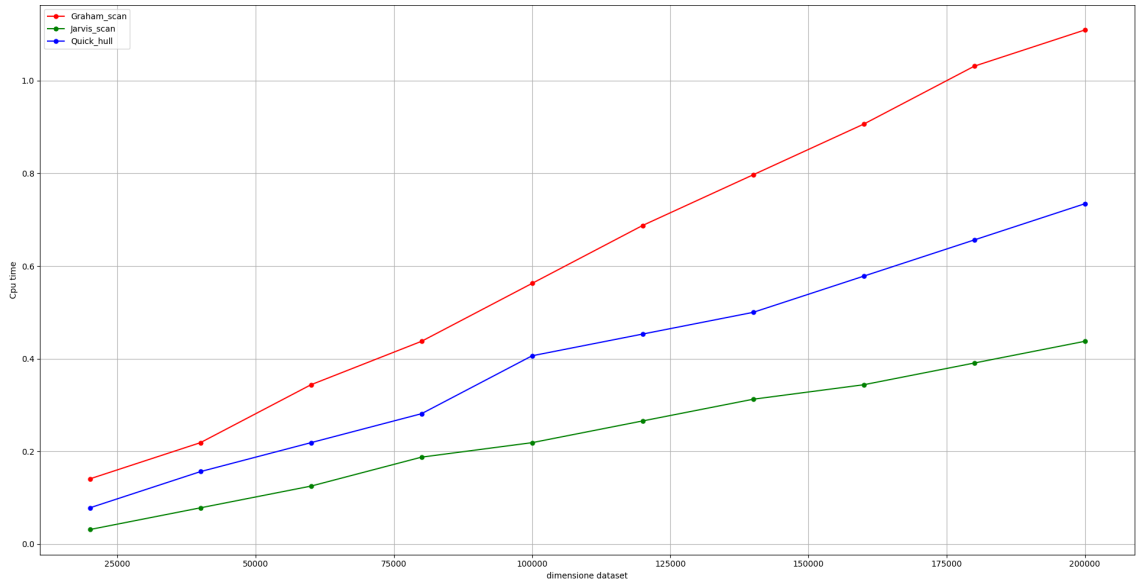

Figura 4.5: esecuzione degli algoritmi con |**CH**(*P*)|=3.

La figura 4.5 dimostra in maniera chiara come Jarvis_scan sia in assoluto il migliore dei tre algoritmi nel caso estremo in cui ci siano solo tre vertici a formare l'inviluppo convesso. Da notare che anche per quick_hull si osserva un miglioramento considerevole del tempo di esecuzione perché è stato in grado di scartare un numero elevato di vertici fin da subito.

L'algoritmo di Jarvis, come era prevedibile, ottiene risultati paragonabili agli altri due algoritmi quando $\alpha=O(logn)$ come illustrato in figura 4.6.

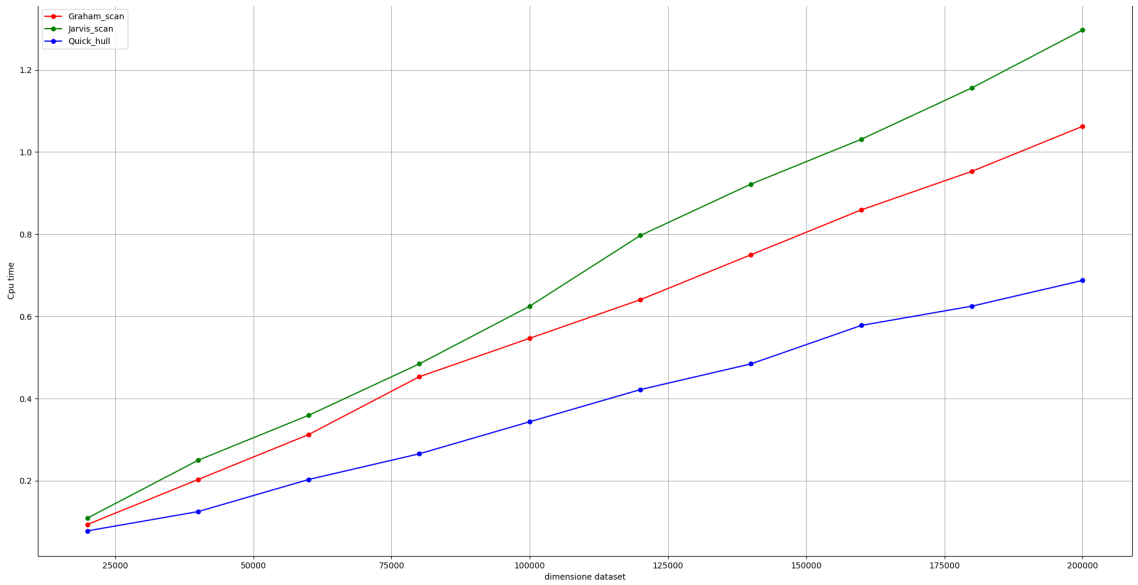

Figura 4.6: esecuzione degli algoritmi con $\alpha=O(logn)$.

Jarvis_scan non è assolutamente un algoritmo da utilizzare in caso di distribuzioni di punti casuali in quanto nella maggior parte dei casi ottiene prestazioni di gran lunga peggiori rispetto al grafico in figura 4.5; si pensi ad esempio ad una distribuzione circolare composta da n elementi, è provato che i punti facenti parte dell'inviluppo convesso saranno $h=O(n^{1/3})$, per cui l'andamento asintotico di Jarvis_scan sarà $O(nh) = O(nn^{1/3}) = O(n^{4/3})$. Nella pratica tale algoritmo è conveniente da utilizzare nel caso in cui sia già noto a priori che $|\mathbf{CH}(P)|$ è un valore basso.

Analisi delle prestazioni di Graham_scan

L'algoritmo di Graham è sicuramente quello che più dei tre che mantiene le sue prestazioni costanti al variare del numero di vertici del convex hull, il dettaglio su cui si sofferma questo sottoparagrafo è che all'aumentare di $|\mathbf{CH}(P)|$ le prestazioni dell'algoritmo migliorano, a differenza degli altri due.

Il motivo si può ritrovare nelle righe 5-6 dello pseudocodice, le quali controllano se un vertice è da scartare o meno. Risulta evidente che nel caso di pochi vertici che formano l'inviluppo convesso tali istruzioni verranno eseguite molteplici volte. Al contrario, se tutti o quasi i vertici fanno parte del convex hull ci sarà solo un controllo ad ogni iterazione del ciclo while.

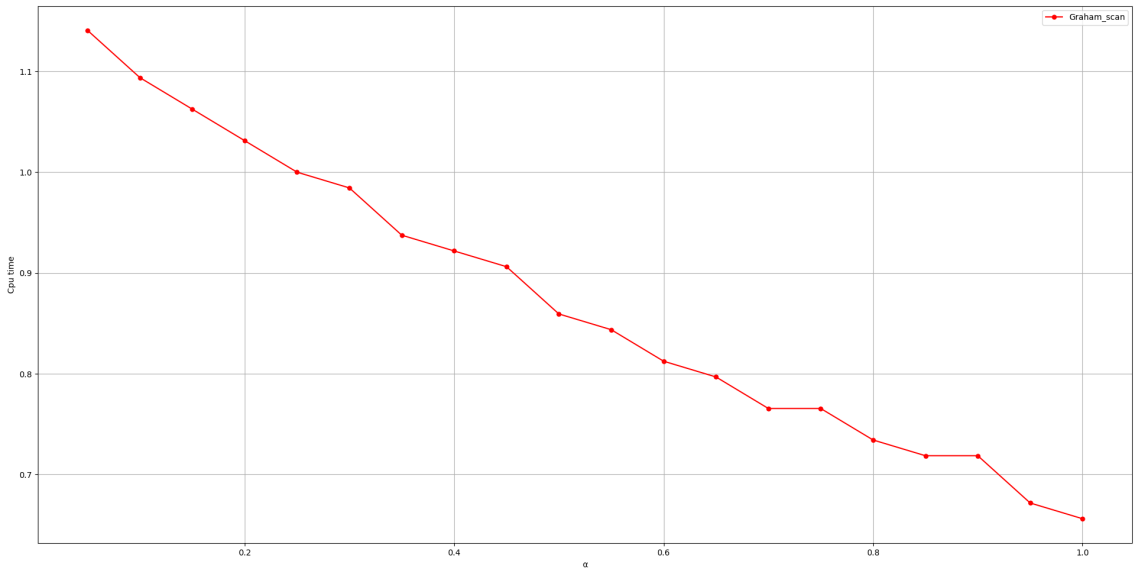

Figura 4.7: andamento di Graham_scan al variare di α.

In figura 4.7 è riportato il tempo di esecuzione di graham_scan (asse y) in funzione di un dataset di 200000 punti ad ogni esecuzione con α che parte da 0.05 e arriva a 1, da tale grafico risulta chiaro che c'è un netto miglioramento del tempo di esecuzione all'aumentare del fattore α.

Si può concludere allora che graham_scan è un algoritmo versatile e che può essere utilizzato in qualsiasi situazione senza doversi preoccupare della forma che può assumere l'inviluppo convesso. In più, se è noto che $|\mathbf{CH}(P)|$ sarà elevato rispetto al numero di punti di P allora tale algoritmo garantisce prestazioni eccellenti.

Analisi delle prestazioni di quick_hull

Si potrebbe dire che quick_hull sia una via di mezzo tra i due algoritmi precedenti, in quanto nonostante abbia prestazioni quadratiche nel suo worst-case, risulta molto improbabile che tale situazione si verifichi. Il vantaggio principale di tale algoritmo è che durante l'inizio della chiamata ricorsiva, in particolar modo le due chiamate a *find_hull* che vengono invocate nella funzione *quick_hull*, se i punti sono distribuiti in modo favorevole si può scartare un numero consistente di vertici dell'insieme di partenza a priori, perché rientreranno all'interno del triangolo composto dai vertici estremi e dal punto più lontano da tale segmento. Tale operazione garantisce un vantaggio delle performance non indifferente in alcuni casi.

Data una distribuzione circolare settando un valore di α=0 (ossia i vertici possono assumere qualsiasi posizione all'interno del cerchio), risulta che il numero di vertici facenti parte dell'inviluppo convesso è $O(n^{1/3})$. Allora, in questa situazione che si può considerare un caso ideale, occorre valutare quale tra graham_scan e quick_hull sia più veloce.

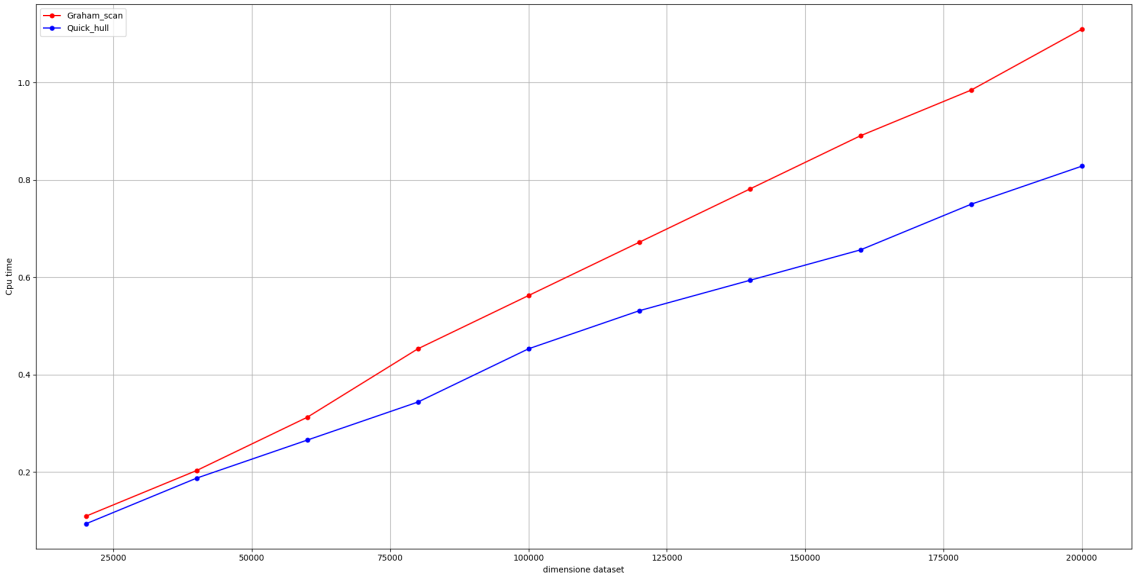

Figura 4.8: confronto tra graham_scan e quick_hull con α=0.

Tale grafico in effetti ricorda molto il confronto tra quick_sort e merge_sort, equiparando tali algoritmi rispettivamente a quick_hull e graham_scan, poiché hanno le corrispettive prestazioni al caso medio e peggiore. Così come quick_sort presenta prestazioni quadratiche in alcuni casi, che però nella pratica non si verificano quasi mai, vale lo stesso anche per quick_hull, quello che succede nella pratica in situazioni standard quick_sort è più veloce di merge_sort, come in questo caso quick_hull risulta essere più veloce di graham_scan.

Volendo entrare nel dettaglio dei risultati di questi test, si può giungere alle seguenti conclusioni:

- graham_scan: tra i tre è l'algoritmo che modifica meno le sue performance in base alla distribuzione dell'insieme di punti di partenza. È un algoritmo che si può utilizzare in qualsiasi contesto è manterrà sempre ottime performance, il caso in cui è più consigliato è quando il numero di punti facenti parte del convex hull è elevato rispetto all'insieme di partenza.
- Jarvis_scan: è sicuramente il migliore dei tre nel caso in cui il numero di vertici di $|\mathbf{CH}(P)|$ sia notevolmente ridotto, ma è assolutamente sconsigliato in distribuzioni di punti particolari nelle quali ci siano troppi punti appartenenti all'inviluppo convesso.
- quick_hull: in situazioni standard è il più efficiente dei tre algoritmi, bisogna solo prestare attenzione nel caso di distribuzioni meno usuali in cui i punti sono distribuiti in modo tale che un numero elevato di vertici faccia parte dell'inviluppo convesso, oppure nel caso in cui le chiamate a find_hull non siano bilanciate, in questi casi quick_hull ottiene prestazioni quadratiche.

4.2 Analisi quick_hull multidimensionale

In quest'ultimo paragrafo verrà presentato un confronto riguardante le prestazioni dell'algoritmo quick_hull nel caso multidimensionale all'aumentare delle dimensioni dello spazio dell'inviluppo convesso, i test verranno eseguiti a partire da uno spazio 2d fino ad arrivare ad uno spazio 4-dimensionale.

Per generare i dataset si utilizzerà la stessa logica usata nel caso specifico bidimensionale, con l'unica differenza che i punti verranno generati casualmente anche sulla superficie dell'ipersfera.

L'obiettivo a cui punta questo test è quello di verificare che la crescita temporale dell'algoritmo aumenta in maniera elevata aumentando le dimensioni, specialmente salendo sopra la terza,

coerentemente con quanto riportato nella sezione riguardante la complessità temporale di tale algoritmo. Risultano di minor importanza le prestazioni in termini assoluti: in particolar modo l'algoritmo implementato è consigliato solo per spazi fino alla terza dimensione.
Come si vedrà in seguito il motivo principale del peggioramento delle performance sarà da attribuire principalmente all'aumentare delle iterazioni del ciclo while (riga 6 dello pseudocodice), in quanto all'aumentare delle dimensioni aumentano anche il numero di facce dell'inviluppo convesso (a parità di punti del dataset iniziale).

Confronto asintotico della complessità

Questo sottoparagrafo tratterà un confronto riguardante la crescita della complessità dell'algoritmo quick_hull all'aumentare delle dimensioni. Il grafico rappresenterà in asse x e y rispettivamente la dimensione del dataset e il tempo di esecuzione, in questo caso i dataset avranno taglia nettamente inferiore a quelli in 2d, ed è stata scelta una percentuale di punti sulla superficie dell'ipersfera pari al 10%. Si noti che ciò riduce le probabilità che il numero di punti appartenenti a $|\mathbf{CH}(P)|$ sia esattamente il 10% per due motivi: innanzitutto perché in questi test i punti non sono generati uniformemente sulla superficie e poi perché il numero di punti generati è relativamente basso.

Dall'analisi svolta nel paragrafo inerente all'analisi asintotica di quick_hull multidimensionale si può dedurre che le prestazioni dell'algoritmo in questione andranno a peggiorare drasticamente salendo dalla quarta dimensione (compresa) in poi, ci sarà inoltre un peggioramento non asintotico "fisiologico", in quanto salendo di dimensione aumentano conseguentemente il numero di facce da processare.

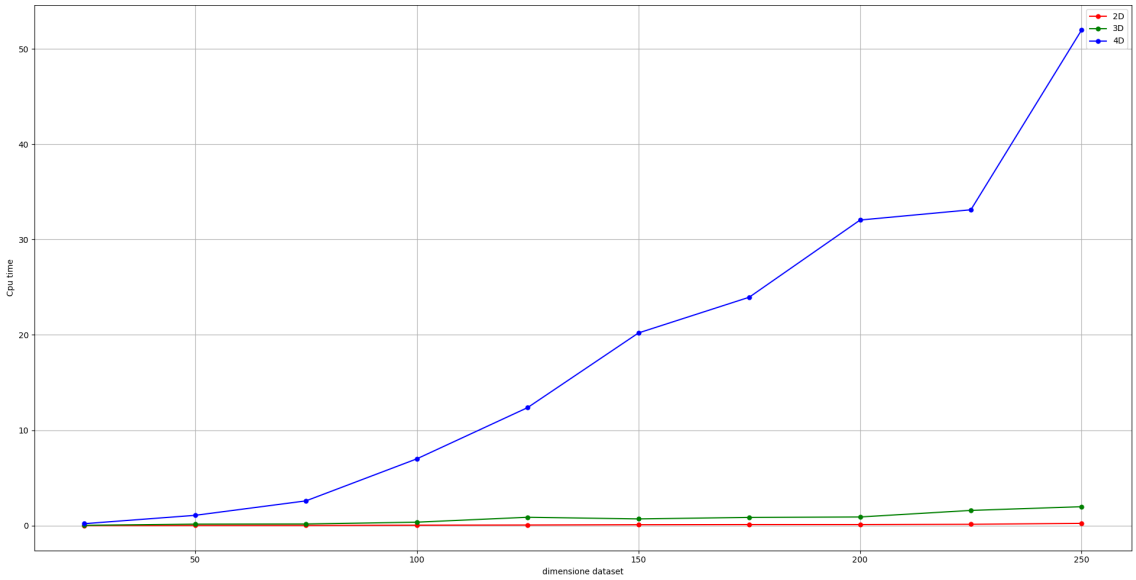

Figura 4.9: Analisi di quick_hull nel caso 2d, 3d, 4d.

Il grafico conferma tutte le premesse precedenti. Il motivo di un andamento meno regolare per quick_hull in 4d (specialmente verso la fine del grafico) è da ricercarsi nel fatto che i dataset generati non garantiscono al 100% un'esecuzione bilanciata e che il metodo utilizzato per trovare il simplesso iniziale non garantisce che vengano rimossi un numero elevato di vertici.
Si può notare inoltre che anche per la terza dimensione l'algoritmo sembra crescere con una velocità relativamente bassa.

L'ultimo test effettuato avrà lo scopo di spiegare l'aumento del tempo di esecuzione dell'algoritmo all'aumentare delle dimensioni, mettendo a confronto il numero di facce generate durante un'esecuzione di quick_hull in 2d, 3d e 4d. Infatti l'aumento di questo valore è la causa principale del peggioramento delle performance e tale valore cresce velocemente all'aumentare della dimensione del problema.

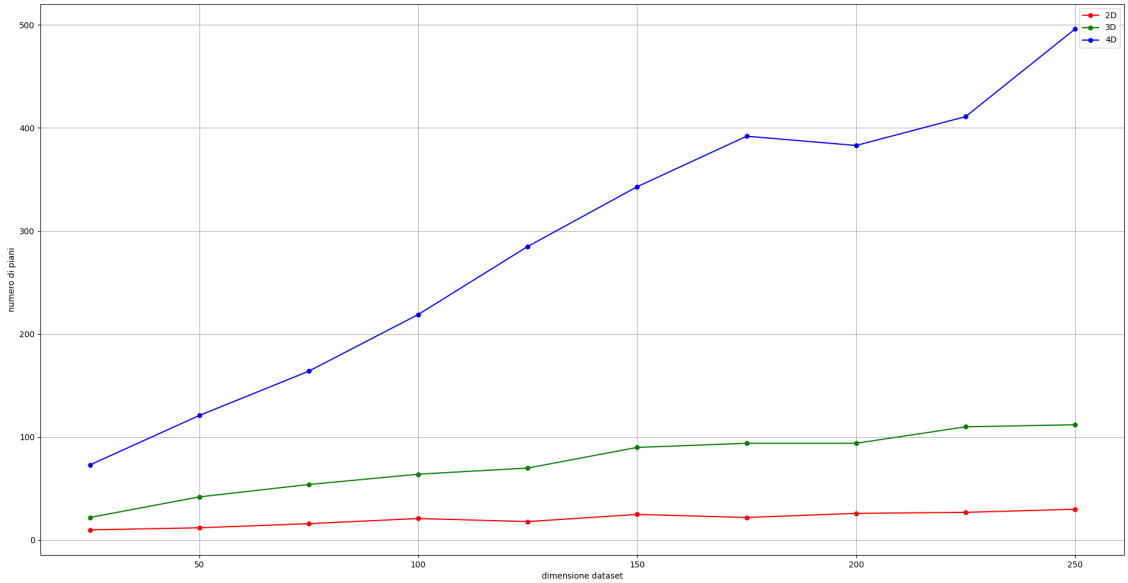

Figura 4.10: numero di facce processate durante un'esecuzione di quick_hull.

Risulta evidente che il numero di facce non cresce in maniera lineare all'aumentare delle dimensioni, o meglio ciò avviene fino alla terza dimensione, in particolare dalla quarta dimensione in poi se $|P|=n$, fin da subito si ha che il numero di facce processate è maggiore di n, mentre ciò non accade per dimensioni minori di quattro. Si ricorda che nell'analisi delle prestazioni del terzo capitolo si ha definito la quantità $f_h = O\left(h^{\lfloor d/2 \rfloor} / \lfloor d/2 \rfloor !\right)$ che indicava il numero di facce di un inviluppo convesso composto da h vertici. Si può notare che tale quantità è lineare fino alla terza dimensione, (come confermato dal grafico 4.10), mentre per la quarta (e anche la quinta) dimensione invece tale quantità aumenta con un andamenti quadratico, cubico per la sesta e settima e cosi via. Vedendo una crescita così elevata del numero di facce di **CH**(P), che si traduce in un numero sempre maggiore di iterazioni del ciclo while a riga 6 dello pseudocodice, è comprensibile che aumentando le dimensioni le prestazioni peggiorino drasticamente.

5 Conclusioni

Il lavoro effettuato durante lo svolgimento di questa tesi mi ha permesso di comprendere al meglio il problema dell'inviluppo convesso, facendomi capire che per problemi di questo genere, come spesso accade, esistono numerosi percorsi che portano alla stessa soluzione. Inoltre, il lavoro di programmazione svolto mi ha permesso di mettere in pratica concetti di geometria studiati solo a livello teorico oltre che applicare vari paradigmi di programmazione. In più, il campo applicativo dell'inviluppo convesso mi ha permesso di approfondire alcuni concetti studiati durante il corso di studi riguardanti altre materie come ad esempio la programmazione lineare.

La necessità di mettere a confronto gli algoritmi mi ha obbligato ad utilizzare al meglio il linguaggio Python, in modo tale da creare algoritmi efficienti e da poterli confrontare in maniera più oggettiva possibile, oltre che a provare varie possibilità per eseguire una determinata operazione e scegliere quella più efficiente.

Di algoritmi per l'inviluppo convesso ne esistono molti altri oltre a quelli trattati in questa tesi, specialmente in due dimensioni. Ad esempio è possibile combinare insieme gli algoritmi di Graham e di Jarvis per ottenere una procedura asintoticamente più efficiente degli altri due metodi denominata Chan's algorithm, che può essere estesa anche fino alla terza dimensione.

Infine, il confronto degli algoritmi creati ha permesso non solo di verificare in maniera empirica le loro prestazioni, ma soprattutto di capire in quale situazione conviene usare un determinato algoritmo rispetto ad un altro.

Fonti bibliografiche e sitografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition, MIT Press

https://it.wikipedia.org/wiki/Ricerca_operativa

Francesco Bottacin, *Algebra Lineare e geometria*, Esculapio, 2° edizione.

Jarvis, R., 1973. "*On the identification of the convex hull of a finite set of points in the plane*". *Information Processing Letters*, 2(1), pp.18-21. DOI: 10.1016/0020-0190(73)90020-3.

Greenfield,, Jonathan Scott*, "A Proof for a QuickHull Algorithm"* (1990)*. Electrical Engineering and Computer Science - Technical Reports. 65.* URL: https://surface.syr.edu/eecs_techreports/65.

Barber, C., Dobkin, D. and Huhdanpaa, H., 1996. "*The quickhull algorithm for convex hulls*". *ACM Transactions on Mathematical Software*, 22(4), pp.469-483. DOI: 10.1145/235815.235821.

http://www.mat.unimi.it/users/alzati/Geometria_Computazionale_98-99/apps/quickhull3d/teoria.html

Appendice A

Codice sorgente degli algoritmi per l'inviluppo convesso

Graham_hull.py:

```
import numpy as np
from common_functions import polar_angle, d_angle, compare


def graham_scan(point_list: list):
  """ funzione che trova l'involucro convesso di point_list, utilizzando
  l'algoritmo di Graham.

  :param point_list: lista contenente i punti in cui trovare l'involucro
  convesso.

  :return: lista contenente i punti di CH(point_list) in senso
   antiorario. """

  # punto con ordinata minore più a sinistra
  p0 = min(point_list, key=lambda p: (p[1], p[0]))

  # lista di punti ordinati rispetto all'angolo polare con p0
  ordered_points = sorted(point_list, key=lambda p: polar_angle(p0, p) if
  not compare(p, p0) else float('inf'))

  # inizializzo stack contenente p0 e i primi due punti di point_list
  ordinata
  s = [p0, ordered_points[0], ordered_points[1]]

  for p in ordered_points[2: - 1]:
  # il ciclo continua fino a che l'angolo formato da s[-2], s[-1] e p
  effettua una svolta non a sinistra
    while d_angle(s[-2], s[-1], p) >= 0:
      s.pop()

    s.append(p)

  return s
```

Jarvis_hull.py:

```
import numpy as np
from common_functions import d_angle, compare, plot_hull, cross,
euc_distance, find_angle, find_idx


def jarvis_scan(point_list: list):
   """ funzione che trova l'involucro convesso tramite l'argoritmo di Jarvis.

   :param point_list: lista contenente i punti in cui trovare l'involucro
   convesso.

   :return: lista contenente i punti di CH(point_list) in senso antiorario."""

   # copio la lista di punti iniziali
   pts = point_list.copy()

   # punti con ordinata rispettivamente minore, maggiore più a sinistra,
   destra
   min_y, max_y = find_idx(point_list, 1)

   # punto in cui si stà lavorando al momento
   current_p = pts[min_y]
   ph = point_list[max_y]
   pl = current_p

  # lista contenente i punti CH(point_list) in senso antiorario partendo da
   pl
  hull = [current_p]

  del pts[min_y]

   # costruisco la catena destra
   while current_p[1] != ph[1]:
      idx = find_angle(current_p, pts, "min")
      current_p = pts[idx]
      hull.append(current_p)
      del pts[idx]

   pts.append(pl)

   # costruisco la catena sinistra
   while current_p[1] != pl[1]:
      idx = find_angle(current_p, pts, "max")
      current_p = pts[idx]
      hull.append(current_p)
      del pts[idx]

   # la lista contiene pl sia all'inizio che alla fine
   return hull[:-1]
```

quick_hull2d.py:

```
import numpy as np
from common_functions import  find_pos, find_index, plot_hull, find_furthest,
find_idx


def find_hull(s, pl, pr, hull):
   """ chiamata ricorsiva per trovare la parte superiore o inferiore
   dell'inviluppo convesso

   :param s: sottoinsieme della lista di punti originale contenente i punti
   sopra o sotto il segmento con estremi pl e pr
   :param pl: estremo sinistro
   :param pr: estremo destro
   :param hull: lista contenente i punti dell'involucro convesso che si andrà
   a riempire durante l'algoritmo

   :return: trova l'involucro convesso di s. """

   # caso base (s vuota)
   if not s:
      return

   # trova il punto più lontano dalla semiretta con estremi pl e pr
   pf, idx = find_furthest(pl, pr, s)

   # rimuovo il punto trovato
   del s[idx]

   # divido s in 2 sottoinsiemi: uno contiene i punti che stanno a sinistra
del  triangolo pl-pf-pr e l'altro i punti che stanno a destra
   pa1, pb1 = find_pos(pl, pf, s)
   pa2, pb2 = find_pos(pr, pf, s)

   # chiamate ricorsive
   if pl[0] < pr[0]:
      find_hull(pa1, pl, pf, hull)
      # inserico il punto in questo momento in modo tale da ottenere i punti
      in senso orario
      hull.append(pf)
      find_hull(pa2, pf, pr, hull)
   else:
      find_hull(pb1, pl, pf, hull)
      # inserico il punto in questo momento in modo tale da ottenere i punti
      in senso orario
      hull.append(pf)
      find_hull(pb2, pf, pr, hull)

     return
```

```
def quick_hull(point_list: list):
    """ funzione che trova l'involucro convesso di point_list usando l'algorito
    quickhull

    :param point_list: lista contenente i punti in cui trovare l'involucro
    convesso.

    :return: lista contenente i punti di CH(point_list) in senso antiorario."""

    ordered_points = point_list.copy()
    # punti con ascissa minore e maggiore
    min_x, max_x = find_idx(point_list, 0)

    pl, pr = ordered_points[min_x], ordered_points[max_x]

    # inizializzo l'inviluppo convesso con pl
    hull = [pl]

    del ordered_points[min_x]
    del ordered_points[max_x]

    # divido i punti in quelli che stanno sopra e sotto al segmento con estremi
    pl e pr
    s1, s2 = find_pos(pl, pr, ordered_points)

    # inizia la chiamata ricorsiva al metodo find_hull() per la parte sopra e
    sotto
    find_hull(s1, pl, pr, hull)
    # inserico il punto in questo momento in modo tale da ottenere i punti in
    senso orario
    hull.append(pr)
    find_hull(s2, pr, pl, hull)

    return hull
```

multidimensional_quick_hull.py:

```
import numpy as np

from common_functions import distance_pp, plot3d, plot_quickhul, is_inside,
not_in, set_normal, compute_faces, compute_horizon, compute_simplex, compare,
euc_distance, find_furthest, get_plane

class ridge:
  """ rappresenta uno dei contorni delle facce dell'inviluppo convesso"""
  def __init__(self, points):
    # punti del contorno
    self.points = points

  def __eq__(self, e):
    for p0 in self.points:
      for p1 in e.points:
        if compare(p0, p1):
          break
        else:
          return False
    return True

  def __hash__(self):
    l = len(self.points[0])
    return sum([hash((p[i] for i in range(l))) for p in self.points])


class facet:
  """ faccia dell'inviluppo convesso"""
  def __init__(self, points):
    # punti della faccia
    self.points = points
    # centroide della daccia
    self.center = np.sum(points, axis=0) / len(points[0])
    # normale della faccia
    self.normal = self.compute_normal()
    # punti visibili dalla faccia
    self.outside_list = []
    # contorni che compongono la faccia
    self.edges = self.compute_edges()

  def compute_edges(self):
    """ funzione che crea i contorni della faccia a partire da points. """
    # dimensione dei punti
    dim = len(self.points[0])

    return [ridge([self.points[j] for j in range((i - dim), i - 1)]) for i
    in range(dim)]
```

```
  def compute_normal(self):
    """ funzione che calcola la normale iniziale del piano senza tenere
     conto del suo orientamento. """
    # dimensione dei punti
    dim = len(self.points[0])

    # ottengo l'equazione dell'iperpiano del tipo: ax + by + cz = 1
    X = np.matrix(self.points)
    unitary = np.ones((1, dim), dtype=np.int32)

    eq = np.dot(np.linalg.inv(X), unitary.transpose())
    return eq / np.linalg.norm(eq)

  def set_outside(self, points):
    """ funzione che stabilisce i punti visibili dalla faccia

    :param points: insieme di punti da valutare

    :return: la outside-list della faccia rispetto a points. """
    self.outside_list = [p for p in points if
np.dot(self.normal.transpose(), p - self.points[0]) > 10 ** -10]

  def __eq__(self, f):
    for p0 in self.points:
      for p1 in f.points:
        if compare(p0, p1):
          break
        else:
          return False
    return True

  def __hash__(self):
    l = len(self.points[0])
    return sum([hash((p[i] for i in range(l))) for p in self.points])
```

```
def n_qhull(point_list: list):
    """ trova l'inviluppo convesso n-dimensionale di point_list.

    :param point_list: lista contenente i punti iniziali

    :return: una lista contenente le facce dell'involucro convesso. """
    dim = len(point_list[0])

    # creo un simplesso iniziale di dim + 1 punti
    s = compute_simplex(point_list, dim)

    # suddivido il simplesso in facce
    hull = {facet([s[j] for j in range((i - dim) - 1, i - 1)]) for i in
    range(dim + 1)}

    # setto le normali dei poligoni fuori dal simplesso
    set_normal(hull)

    # inizializzo il set di punti che si trovano fuori dal simplesso
    outside = [point for point in point_list if not is_inside(point, hull) and
    not_in(point, hull)]

    # calcolo le outside_list degli iperpiani di hull
    for p in hull: p.set_outside(outside)

    while outside:
        # iperpiano su cui lavorare durante questa iterazione
        current_plane, pf = get_plane(hull)

        # mantengo le facce non visibili da pf
        visible, H_old = compute_faces(hull, pf)

        # trovo l'insieme dell'orizzonte per pf
        horizon = compute_horizon(visible, H_old)

        # creo l'insieme delle facce da aggiungere a hull
        H_new = set()
        for fc in horizon:
            ho = fc.points.copy()
            ho.append(pf)
            H_new.add(facet(ho))

        # setto le normali dei poligoni fuori dal simplesso
        set_normal(H_new)

        # il nuovo involucro convesso
        hull = H_old | H_new

        # ricalcolo il nuovo outside set
        outside = [point for point in outside if not is_inside(point, hull) and
          not_in(point, hull)]

        # calcolo le outside_list dei piani di hull
        for p in hull:
            p.set_outside(outside)

    return [x.points for x in hull]
```

common_functions.py:

```
import numpy as np
from math import sqrt, pi, atan2


def find_pos(p1, p2, s):
   """ funzione che stabilisce i punti che si trovano sopra e sotto il
   segmento  definito da p1 e p2.

   :param p1: punto sinistrodel segmento
   :param p2: punto destro del segmento
   :param s: insieme di punti su cui cercare

   :return: i punti che si trovano sopra e sotto il segmento."""

   s1, s2 = [], []

   # calcolo i coefficienti m e q per ottenere l'equazione y = mx + q
   m = (p2[1] - p1[1]) / (p2[0] - p1[0])
   q = - m * p1[0] + p1[1]

   for p in s:
      if p[1] > m * p[0] + q:
         s1.append(p)
      elif p[1] < m * p[0] + q:
         s2.append(p)
   return s1, s2


def find_distance(p1, p2, p):
   """ funzione che trova la distanza tra il segmento p1-p2 e il punto p.

   :param p1: punto sinistro del segmento
   :param p2: punto destro del segmento
   :param p: punto su cui trovare la distanza

   :return: la distanza cercata. """

   # calcolo i coefficienti
   a = p1[1] - p2[1]
   b = p2[0] - p1[0]
   c = p1[0] * p2[1] - p2[0] * p1[1]

   return abs(a * p[0] + b * p[1] + c) / sqrt(a * a + b * b)
```

```
def find_furthest(pl, pr, s):
    """ funzione che trova il punto contenuto in s più lontano dal segmento pl-
    pr.

    :param pl: punto sinistrodel segmento
    :param pr: punto destro del segmento
    :param s: insieme di punti su cui cercare

    :return: il punto più lontano. """

    # valori iniziali
    max_dist = -1
    furthest_point = None
    dx, iterations = 0, 0
    # cerco il punto più distante
    for point in s:
        dist = find_distance(pl, pr, point)
        if dist > max_dist:
            max_dist = dist
            furthest_point = point
            idx = iterations
            iterations += 1

    return furthest_point, idx


def compare(p1, p2):
    """ funzione che controlla se due array sono uguali, sostituisce la
    funzione del modulo numpy, poiché quest'ultima effettua controlli che
    rallentano il tempo di esecuzione, in questo caso inutilmente.

    :param p1: primo punto
    :param p2: secondo punto

    :return: True se p1 e p2 sono uguali, False altrimenti. """

    for i in range(len(p1)):
        if p1[i] != p2[i]:
            return False
    return True


def polar_angle(origin, point):
    """ funzione che calcola l'angolo polare tra due vettori sfruttando la
    definizione di prodotto scalare.

    :param origin: punto che verrà reso l'origine.
    :param point: punto rispetto al quale calcolare l'angolo polare.

    :return: l'angolo polare di (origin - point) rispetto all'origine in senso
    antiorario. """

    center = point - origin
    return - center[0] / (sqrt(center[0]**2+center[1]**2))
```

```
def cross(p1, p2):
  """ funzione che calcola l'area con segno del parallelogramma formato da
  due vettori,sostituisce la funzione del modulo numpy, poiché quest'ultima
  effettua controlli che rallentano il tempo di esecuzione, in questo caso
  inutilmente.

  :param p1: primo vettore
  :param p2: secondo vettore

  :return: l'area con segno del parallelogramma formato da p1 e p2. """

  return p1[0] * p2[1] - p1[1] * p2[0]


def d_angle(p0, p1, p2):
  """ funzione che ritorna il vettoriale scalare tra (p2 - p0) e (p1 - p0)
  rispetto all'origine.

  :param p0: punto iniziale.
  :param p1: punto centrale.
  :param p2: punto finale.

  :return: un valore che può essere:
    - > 0: allora l'angolo effettua una svolta a sinistra
    - < 0: allora l'angolo effettua una svolta a destra
    - = 0: allora i punti sono collineari """

   return cross(p2 - p0, p1 - p0)


def get_plane(hull):
  """ funzione che ritorna il piano avente il punto con distanza maggiore in
  un inviluppo convesso temporaneo.

  :param hull: inviluppo convesso

  :return: il piano avente il punto con distanza maggiore in hull. """

  m_dist = 0
  for p in hull:
    for point in p.outside_list:
      dist = distance_pp(p.normal.transpose(), point)
      if dist > m_dist:
        m_dist = dist
        current_plane = p
        pf = point
  return current_plane, pf
```

```
def find_idx(point_list, coord):
  """ funzione che trova l'indice del punto minimo e massimo rispetto ad una
  data coordinata.

  :param point_list: lista su cui cercare i punti
  :param coord: coordinata di riferimento

  :return: l'indice del punto minimo e massimo rispetto a coord in
  point_list. """

  min_c, max_c = float('inf'), float('-inf')
  min_idx, max_idx = 0, 0
  it = 0
  for p in point_list:
    if p[coord] < min_c:
      min_c = p[coord]
      min_idx = it
    if p[coord] > max_c:
      max_c = p[coord]
      max_idx = it
      it += 1
  return min_idx, max_idx


def find_angle(point, point_list, flag):
  """ trova il punto avente minimo o il massimo angolo polare rispetto ad un
  punto di riferimento.

  :param point: punto di riferimento
  :param point_list: lista in cui cercare
  :param flag: se min cerca minimo altrimenti il massimo

  :return: il punto avente minimo o il massimo angolo polare rispetto a
  point. """

  if flag == "min":
    return find_min_angle(point, point_list)
  return find_max_angle(point, point_list)


def find_min_angle(point, point_list):
  """ funzione che trova il punto avente minimo angolo polare rispetto ad un
  punto di riferimento.

  :param point: punto di riferimento
  :param point_list: lista in cui cercare

  :return: il punto avente minimo angolo polare rispetto a point. """
  min_a = pi
  idx = 0
  for i in range(len(point_list)):
    if point_list[i][1] < point[1]: continue
    angle = atan2(point_list[i][1] - point[1], point_list[i][0] - point[0])
    if angle < min_a:
      min_a = angle
      idx = i
  return idx
```

```
def find_max_angle(point, point_list):
    """ funzione che trova il punto avente massimo angolo polare rispetto ad un
    punto di riferimento.

    :param point: punto di riferimento
    :param point_list: lista in cui cercare

    :return: il punto avente massimo angolo polare rispetto a point. """
    max_a = 0
    idx = 0
    for i in range(len(point_list)):
        if point_list[i][1] > point[1]: continue
        angle = atan2(point_list[i][1] - point[1], point_list[i][0] - point[0])
        if angle < max_a:
            max_a = angle
            idx = i
    return idx


def distance(p1, p2, p3):
    """ funzione che calcola la distanza tra p3 e il segmento avente p1 e p2
    come estremi.

    :param p1: estremo sinistro della semiretta
    :param p2: estremo destro della semiretta
    :param p3: punto su cui calcolare la distanza

    :return: la distanza tra p3 e il segmento avente p1 e p2 come estremi. """

    x = p1 - p2
    y = x * (np.dot(p3 - p2, x) / np.dot(x, x)) - p3
    return y[0] ** 2 + y[1] ** 2


def find_index(p, s):
    """ funzione che trova l'indice di p in s.

    :param p: punto in cui trovare l'indice
    :param s: lista in cui cercare

    :return: l'indice di p in s se p è presente, altrimenti -1 """

    for index, point in enumerate(s):
        if compare(point, p):
            return index
    return -1


def euc_distance(p1, p2):
    """ funzione che calcola la distanza euclidea fra 2 punti.

    :param p1: primo punto
    :param p2: secondo punto

    :return: il valore della distanza euclidea tra i due punti. """
    return sum((p1 - p2)**2)
```

```
def distance_pp(norm, p0):
    """ funzione che calcola la distanza tra un punto e un piano.

    :param norm: normale dell' iperpiano su cui calcolare la distanza
    :param p0: punto su cui calcolare la distanza

    :return: la distanza tra l'iperpiano con normale norm e il punto p0. """
    return abs(np.dot(norm, p0))


def is_inside(point, hull):
    """ funzione che stabilisce se un punto è all'interno di un inviluppo
    convesso.

    :param point: punto da stabilire se è interno o meno
    :param hull: inviluppo convesso di riferimento

    :return: True se point è interno a hull o sul suo perimetro, False
    altrimenti. """

    for pl in hull:
        p = np.dot(pl.normal.transpose(), pl.center - point)
        if p < 0:
            return False
    return True


def not_in(point, hull):
    """ funzione che stabilisce se un punto non costituisce l'involucro
    convesso.

    :param point: punto da stabilire se fa parte o meno dell'inviluppo convesso
    :param hull: inviluppo convesso di riferimento

    :return: True se il punto fa parte dell'inviluppo convesso, False
    atrimenti. """

    for pl in hull:
        for p in pl.points:
            if compare(point, p):
                return False
    return True


def set_normal(faces):
    """ funzione che inverte le normali dei piani di un poligono convesso le
    puntano verso l'interno.

    :param faces: faccedel poligono convesso. """
    # calcolo il centroide delle facce
    centroid = sum([fc.center for fc in faces]) / len(faces)

    for p in faces:
        d = np.dot(centroid - p.center, p.normal)
        if d > 0:
            p.normal = - p.normal
```

```
def compute_faces(hull, point):
  """ funzione che restituisce un set delle facce visibili e uno per le facce
  non visibili.

  :param hull: l'involucro convesso da esaminare
  :param point: punto rispetto al quale controlloare le facce non visibili

  :return: 2 set delle faccie di hull computate come scritto sopra. """

  visible, not_visible = set(), set()

  for face in hull:
    if np.dot(face.normal.transpose(), point - face.center) < 0:
      not_visible.add(face)
    else:
      visible.add(face)

  return (visible, not_visible)


def compute_horizon(visible, not_visible):
  """ funzione che definisce l'orizzonte tra visible e not_visible.

  :param visible: set delle facce visibili
  :param not_visible: set delle facce non visibili

  :return: un set che contiene l'intersezione degli edge tra i due insiemi di
  input. """

  v, n = set(), set()

  for face in visible:
    for e in face.edges:
      v.add(e)

  for face in not_visible:
    for e in face.edges:
      n.add(e)

  return v & n
```

```
def compute_simplex(point_list: list, dim):
  """ funzione che trova d + 1 punti non complanari.

  :param point_list: lista dei punti iniziali
  :param dim: dimensione dei punti

  :return: un set di punti non complanari tra di loro. """

  # prendo i punti con coordinata minima e massima per ciascun asse
  ref_points = [(min(point_list, key=lambda p: p[i]), max(point_list,
  key=lambda p: p[i])) for i in range(dim)]

  # trovo l' asse della distanza massima tra le coppie di punti appena
  cercate
  pl, pr = max(ref_points, key=lambda p: euc_distance(p[0], p[1]))

  pf = find_furthest(pl, pr, point_list)[0]

  simplex = [pl, pr, pf]

  # prendo altri punti
  while len(simplex) != (dim + 1):
    # calcolo il baricentro
    center = sum([p for p in simplex]) / len(simplex)
    pf = point_list[0]
    m_dist = euc_distance(center, pf)
    for i in range(1, len(point_list)):
      for p0 in simplex:
        if np.array_equal(point_list[i], p0):
          break
      else:
        dist = euc_distance(point_list[i], center)
        if dist > m_dist:
                  m_dist = dist
                  pf = point_list[i]
      simplex.append(pf)

  return simplex
```