UNIVERSITÀ DEGLI STUDI DI CAMERINO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Matematica

Dipartimento di Matematica e Fisica

Triangolazioni di Delaunay

Tesi Sperimentale di Laurea

in Analisi Numerica

Relatori:

Prof. Luciano Misici

Prof. Riccardo Piergallini

Laureanda:

Federica Benigni

Anno Accademico 1999-2000

A mia nonna Ada.

Indice

Elenco delle figure

Introduzione

Nel suo "*Principia Philosophiae*" (1644), Cartesio affermò che il sistema solare è fatto di vortici. In un'illustrazione (Figura 1) egli mostrò la decomposizione dello spazio in regioni convesse ognuna delle quali era composta di materia che si avvolgeva intorno ad una stella. Questo concetto comparve successiva-

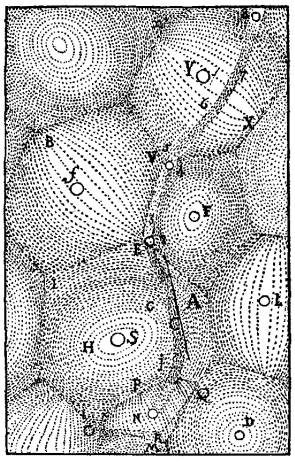

Figura 1: Decomposizione di Cartesio dello spazio in vortici.

mente indipendentemente da questa osservazione e si dimostrò subito utile in vari campi della scienza. Sono usati diversi nomi per questo stesso concetto a seconda del campo in cui sono utilizzati: trasformazione dell'asse mediale in

biologia e fisiologia, zone di Weigner-Seitz in chimica e fisica, domini d'azione in cristallografia e poligoni di Thiessen in meteorologia e geografia.
Dirichlet e Voronoy introdussero formalmente il concetto , intorno ai primi anni del Novecento, che prese il nome di *Diagramma di Voronoy*. Voronoy fu il primo a considerare il *duale* di questa struttura ma fu Delaunay a stabilirne le proprietà. Il duale del diagramma di Voronoy prese così il nome di *triangolazione di Delaunay*.
Con il tempo, gli algoritmi per la costruzione di quest'ultima struttura si dimostrarono più veloci e soddisfacenti, nonchè piú utili nelle applicazioni.
Infatti il metodo ad elementi finiti, particolarmente utile nella simulazione numerica e quindi nello studio di molti dei problemi reali, richiede la divisione dello spazio in elementi semplici come, ad esempio, triangoli in due dimensioni e tetraedri in tre dimensioni. In particolare, la triangolazione di Delaunay ha delle proprietà specifiche che la rendono preferibile ad altri tipi di triangolazioni.
Lo scopo di questa tesi è quello di fornire un quadro delle problematiche legate alla generazione di triangolazioni in due ed in tre dimensioni nonchè degli algoritmi maggiormente utilizzati. A questo scopo è stata svolta un'accurata ricerca bibliografica cercando di analizzare i risultati ottenuti dai piú recenti studi in questo campo.
E' stato, inoltre, implementato un algoritmo per l'inserimento di punti in un dominio poligonale e la generazione di una triangolazione di Delaunay del dominio stesso.
Vediamo in dettaglio il contenuto dei vari capitoli.

Il Capitolo 1 contiene le definizioni fondamentali che saranno utili nel seguito della trattazione.

Il Capitolo 2 illustra alcuni algoritmi per la costruzione della triangolazione di Delaunay di un insieme $\mathcal{S}$ di punti in $\mathbb{R}^2$ e in $\mathbb{R}^3$ con particolare attenzione al metodo incrementale che è stato utilizzato per la creazione del programma.

Il Capitolo 3 illustra alcuni algoritmi per la creazione di mesh dividendo la trattazione per le mesh strutturate e per quelle non strutturate evidenziando le differenze e i diversi campi di utilità ed applicazioni.

Il Capitolo 4, infine, spiega l'algoritmo usato nel nostro programma e mostra alcuni esempi di mesh generate con questo algoritmo.

Capitolo 1

Definizioni Generali

1.1 Introduzione

Lo scopo di questo capitolo è quello di fornire le differenti nozioni e definizioni di insiemi ed elementi che serviranno per gli algoritmi che introdurremo in seguito.
Qui tutte le definizioni vengono date per il caso bidimensionale e tridimensionale, ma possono essere estese senza difficoltà a dimensioni maggiori.

1.2 Il triangolo

Un triangolo è rappresentabile attraverso una terna ordinata di vertici, detti P_i, dati in senso orario o antiorario:

$$K = (P_1, P_2, P_3).$$

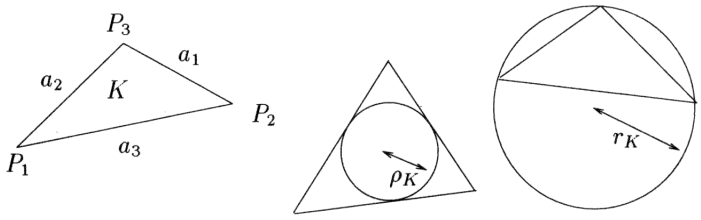

Figura 1.1: Il Triangolo

L'area del triangolo, allora, è un'area "con segno" ed è data da:

$$\mathcal{S}_K = \frac{1}{2} \begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix}$$

oppure da:

$$\mathcal{S}_K = \frac{1}{2} \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

dove x_i , y_i sono le coordinate dei vertici P_i $(i = 1, ..., 3)$ e $|.|$ indica il determinante della matrice. Mentre l'i-esimo lato è indicato da e_i $(i = 1, ..., 3)$ ed è il lato che congiunge (in questo ordine) il vertice P_{i+1} con il vertice P_{i+2} ,dove abbiamo assunto $P_j = P_{j-3}$ se $j > 3$.

Circumcerchio. Il cerchio circoscritto ad un triangolo, o circumcerchio, ha un ruolo fondamentale negli algoritmi di triangolazione. Questo cerchio è completamente descritto dal centro, *circumcentro*, e dal raggio, *circumraggio*. Ci sono due modi per calcolare il circumcerchio. O risolviamo un sistema lineare o usiamo l'equazione del cerchio. L'equazione del circumcerchio è data da:

$$\mathcal{C}_K(x, y) = 0$$

con

$$\mathcal{C}_K = \begin{vmatrix} l_1^2 - l^2 & l_2^2 - l_1^2 & l_3^2 - l_1^1 \\ x_1 - x & x_2 - x_1 & x_3 - x_1 \\ y_1 - y & y_2 - y_1 & y_3 - y_1 \end{vmatrix}$$

dove $l^2 = x^2 + y^2$ e $l_i^2 = x_i^2 + y_i^2, (i = 1, ..., 3)$; calcolando questo determinante possiamo facilmente calcolare sia le coordinate del circumcentro, x_C e y_C, che il circumraggio r_K.

Incerchio. Anche il cerchio inscritto al triangolo, l'incerchio, è di grande interesse per noi, così come il suo raggio, l' *inraggio*, ρ_K definito da:

$$\rho_K = \frac{\mathcal{S}_K}{p_K}$$

dove p_K è il semiperimetro del triangolo K.

1.3 Il tetraedro

Il tetraedro è un poligono con quattro facce triangolari. Esso è ben definito da una lista ordinata dei suoi quattro vertici P_i:

$$K = (P_1, P_2, P_3, P_4)$$

. Se assumiamo che le facce del tetraedro siano orientate, allora anche le normali a queste facce saranno orientate. Così il volume, V_K, dell'elemento K sarà dotato di "segno" e sarà dato da:

$$V_K = \frac{1}{6} \begin{vmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{vmatrix}$$

oppure

$$V_K = \frac{1}{6} \begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}$$

dove x_i, y_i, z_i sono le coordinate dei vertici P_i dell'elemento K. Tutto ciò ci

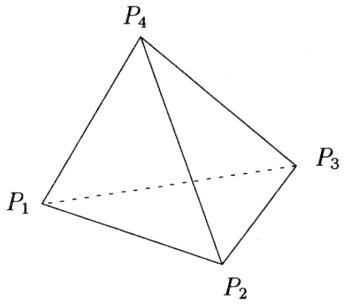

Figura 1.2: Il Tetraedro

permette di definire implicitamente le facce dell'elemento K. Una faccia di K è una terna ordinata (sono permesse tutte le possibili permutazioni), e si ha:

- faccia 1: $P_4\ P_3\ P_2$,
- faccia 2: $P_1\ P_3\ P_4$,
- faccia 3: $P_4\ P_2\ P_1$,
- faccia 4: $P_1\ P_2\ P_3$.

Analogamente gli spigoli di K sono considerati, non come elementi delle facce, ma come enti a sè stanti e sono implicitamente definiti come le coppie ordinate che seguono:

- spigolo 1: $P_1\ P_2$,
- spigolo 2: $P_1\ P_3$,
- spigolo 3: $P_2\ P_4$,
- spigolo 4: $P_2\ P_3$,
- spigolo 1: $P_2\ P_4$,
- spigolo 2: $P_3\ P_4$.

Quindi ciascuno spigolo è definito dal primo al secondo dei suoi vertici.

Circumsfera. Ad ogni tetraedro è associata la sua circumsfera, ossia la sfera circoscritta ai suoi vertici. Il raggio, o *circumraggio*, e il centro, o *circumcentro*, di questa sfera possono essere ottenuti analogamente al caso bidimensionale o usando l'equazione:

$$\mathcal{C}_K = \begin{vmatrix} l_1^2 - l^2 & l_2^2 - l_1^2 & l_3^2 - l_1^2 & l_4^2 - l_1^2 \\ x_1 - x & x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_1 - y & y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_1 - z & z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{vmatrix}$$

dove $l^2 = x^2 + y^2 + z^2$ e $l_i^2 = x_i^2 + y_i^2 + z_i^2$ $(i = 1, ..., 4)$, oppure risolvendo un sistema di equazioni lineari per trovare il circumcentro.

Come in due dimensioni esiste una formula che permette di calcolare il circumraggio e che permette di evitare il calcolo del circumcentro:

$$r_K = \frac{\sqrt{(a+b+c)(a+b-c)(b+c-a)(a-b+c)}}{24V_K}$$

dove a, b e c sono i prodotti delle lunghezze di due spigoli opposti.

Insfera. La sfera inscritta al tetraedro è detta anche insfera. Il raggio dell'insfera, o *inraggio*, è dato da:

$$\rho_K = \frac{3V_K}{S_1 + S_2 + S_3 + S_4}$$

dove S_i è la superficie della faccia i dell'elemento.

1.4 La qualità degli elementi

La *qualità* o il *rapporto atteso* di un dato elemento K è un valore che serve per misurare la forma dell'elemento stesso. L'interesse di questa quantità è dato dal fatto che per la maggior parte delle applicazioni la qualità degli elementi di una mesh è importante per la precisione dei risultati raggiunti. Ci sono molti modi per stabilire la qualità di un elemento e la scelta dell'uno o dell'altro dipende sicuramente dall'utilizzo della mesh che abbiamo in mente.

Per un triangolo K la prima misura di qualità può essere data da:

$$Q_K = \alpha \frac{h_{max}}{\rho_K} = \alpha \frac{h_{max} p_K}{S_K}$$

dove α è un fattore di normalizzazione in modo che per un triangolo equilatero Q_K sia 1, cioè $\alpha = \frac{\sqrt{3}}{6}$, h_{max} è il lato più lungo del triangolo, cioè il suo *diametro*, e ρ_k è il suo inraggio. Q_k varia tra 1, assunto quando il triangolo è equilatero, e ∞ per un elemento totalmente allungato. Per avere un range di valori tra 0 e 1 possiamo usare l'inverso di Q_K. Analogamente per il tetraedro:

$$Q_K = \alpha \frac{h_{max}}{\rho_K} = \alpha \frac{h_{max} S_K}{3V_K}$$

dove S_K è la somma delle S_i e $\alpha = \frac{\sqrt{6}}{12}$

Un'altra misura alternativa per il triangolo è:

$$Q_K = \beta \frac{{h_s}^2}{S_K}$$

dove β è un fattore di normalizzazione che assicura un valore unitario di Q_K per un triangolo equilatero, (cioè $\beta = \frac{\sqrt{3}}{12}$) e $h_s = \sqrt{\Sigma_{i=1}^{3} {L_i}^2}$ dove L_i è la lunghezza del lato i del triangolo. Analogamente per il tetraedro:

$$Q_K = \beta \frac{{h_s}^3}{V_K}$$

dove $h_s = \sqrt{\Sigma_{i=1}^{6} {L_i}^2}$ ed L_i è la lunghezza dell'i-esimo spigolo del tetraedro e $\beta = \frac{\sqrt{3}}{216}$.

Questo valore, pur essendo come il precedente il rapporto atteso per un dato elemento, è meno sensibile per la determinazione degli elementi mal formati specialmente in tre dimensioni.

Oltre a quelli appena indicati, ci sono altri modi per valutare la qualità di un simplesso. Prima di elencare alcune possibili misure di qualità (in tre dimensioni) sarà utile ricordare che i simboli usati hanno il valore di cui sopra e che r_K è il circumraggio della sfera, $h_{min} = \min_i L_i$ e che L_m è il valore medio degli L_i. Ricordiamo inoltre che:

Definizione 1. *L'angolo diedro tra due facce è il valore:*

$$\delta = \pi \pm \arccos \langle \overrightarrow{n_1}, \overrightarrow{n_2} \rangle$$

che dipende dalla configurazione delle due facce rispetto a $\overrightarrow{n_i}$ *, le normali uscenti dalle facce considerate.*

Definizione 2. *L'angolo solido ad un vertice, θ, è l'area superficiale della porzione di sfera unitaria centrata in quel vertice limitata dalle tre facce che partono da quel punto.*

Secondo questa notazione, le misure di qualità per un elemento possono essere date da:

- $\frac{r_K}{\rho_K}$, rapporto tra i raggi della sfera inscritta e circoscritta,
- $\frac{r_K}{h_{max}}$, rapporto tra inraggio e diametro,
- $\frac{h_{max}}{h_{min}}$, rapporto tra le lunghezze estreme,
- $\frac{{V_K}^4}{(\Sigma_{i=1}^4 {S_i}^2)^3}$, rapporto tra volume e area delle facce,
- $\frac{{L_m}^3}{V_K}$, rapporto tra lato medio e volume,
- δ_{max}, massimo angolo diedro tra due facce,
- θ_{min}, minimo angolo solido associato ad un vertice di K.

Ognuna di queste misure offre vantaggi e svantaggi e nessuna è, da sola, in grado di fornire la classificazione perfetta dell' elemento. Tuttavia è utile considerare che, dal punto di vista computazionale e implementativo, alcune possono risultare particolarmente dispendiose.
La definizione di rapporto atteso più usato per un simplesso è il rapporto tra il raggio della sfera inscritta e circoscritta. Avere un rapporto atteso limitato equivale a richiedere angoli (diedri o solidi) non troppo piccoli ([8]). In base all'ampiezza di questi angoli è possibile fare una classificazione dei simplessi in tre dimensioni che hanno un rapporto atteso non buono. I cinque tipi di elementi sono:

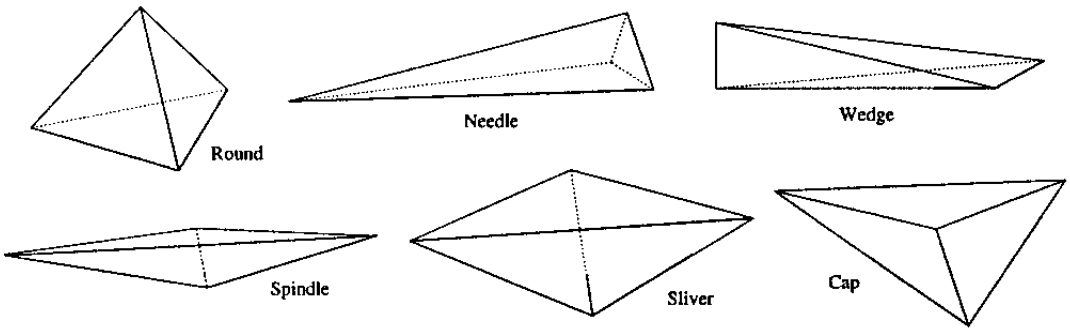

Figura 1.3: Tetraedro ottimo (in alto a sinistra) e tetraedri mal formati.

- **Needle.** Un tetraedro needle ha un angolo solido grande ma angoli diedri nè troppo grandi nè troppo piccoli.
- **Wedge.** Un tetraedro wedge ha angoli diedri piccoli. Un esempio è un tetraedro che è la chiusura convessa di due spigoli corti e perpendicolari fra loro.
- **Sliver.** Un tetraedro sliver ha angoli diedri sia grandi che piccoli ma angoli solidi non troppo grandi. Un esempio è un tetraedro formato da quattro punti quasi complanari e quasi alla stessa distanza sulla superficie della circumsfera. I tetraedri slivers sono distinguibili dagli altri tetraedri mal formati a causa del basso rapporto tra il raggio della circumsfera e lo spigolo più corto.
- **Spindle.** Un tetraedro spindle ha piccoli angoli solidi e grandi angoli diedri.
- **Cap.** Un tetraedro cap ha un angolo solido molto grande, quasi piatto. Il raggio della circumsfera è molto più grande dello spigolo più lungo.

Generalmente, invece, un tetraedo non mal formato è detto **round**.

1.5 I simplessi

Sia $\mathbb{R}$ uno spazio di dimensione d. Sia $\mathcal{S}$ un insieme di punti in $\mathbb{R}^d$. Se $\mathcal{S} = \{A_1, \cdots, A_n \cdots\}$, allora:

$$\sum_{i=0}^{n} \lambda_i A_i$$

rappresenta una combinazione lineare dei punti in $\mathcal{S}$. Se $\sum_{i=0}^{n} \lambda_i = 1$, questa combinazione lineare degli n elementi di $\mathcal{S}$ definisce un sottospazio di $\mathbb{R}^d$ detta **combinazione affine** degli $\mathcal{A}_i$. Se, per ogni i, $\lambda_i \geq 0$ allora quella combinazione è detta **combinazione convessa**.

La **chiusura convessa** di $\mathcal{S}$, detta $\mathcal{C}onv(S)$, è il sottoinsieme di $\mathbb{R}^d$ generato da tutte le combinazioni lineari convesse dei membri di $\mathcal{S}$. Questo insieme è il più piccolo insieme convesso che contiene l'insieme $\mathcal{S}$.

La chiusura convessa di un numero finito di punti in $\mathbb{R}^d$ è detta **politopo** ed è un insieme chiuso e limitato di $\mathbb{R}^d$. Un politopo in dimensione k (la cui chiusura affine è ancora in dimensione k) è un k-politopo. La chiusura convessa di $k+1$ punti $k \leq d$ è un particolare k-politopo detto **simplesso** o più generalmente k-simplesso.

Così, in due dimensioni, un 2-simplesso è un triangolo, mentre in tre dimensioni, un 3-simplesso è un tetreadro. D'ora in poi per riferirci ad un k-simplesso scriveremo più semplicemente *simplesso*.

Una k-faccia è un simplesso di dimensione $k < d$. Quindi una (-1)-faccia è

un insieme vuoto, una 0-faccia è un vertice, una 1-faccia è uno spigolo e così via. Molti dei risultati e delle formule da noi dati per i triangoli e i tetraedri, in realtà, valgono anche in dimensioni maggiori. In particolare i concetti di circumsfera, insfera, circumraggio e inraggio, cosiccome la definizione della qualità degli elementi.

1.6 La triangolazione

Definizione 3. *Un insieme di punti $\mathcal{S}$ in $\mathbb{R}^d$ ($d = 2$ o $d = 3$) è in* ***posizione generale****, in due dimensioni, se non esistono tre punti collineari o quattro punti cociclici, cioè che appartengono alla stessa circonferenza. In tre dimensioni, significa che non devono esistere quattro punti coplanari o cinque punti che appartengano alla stessa sfera.*

Sia ora $\mathcal{S}$ un insieme di punti in posizione generale e sia Ω il dominio in $\mathbb{R}^d$ definito da $Conv(\mathcal{S})$. Se K è un simplesso (triangolo o tetraedro secondo la dimensione d) allora:

Definizione 4. *$\mathcal{T}_r$ è un* ***ricoprimento simpliciale*** *di $\Omega = Conv(\mathcal{S})$ se valgono le seguenti condizioni:*

1. *L'insieme dei vertici degli elementi di $\mathcal{T}_r$ è esattamente $\mathcal{S}$.*
2. *$\Omega = \bigcup_{K \in T_r} K$.*
3. *Ogni elemento K in $\mathcal{T}_r$ è non vuoto.*
4. *L'intersezione dell'interno degli elementi, presi due a due, è un insieme vuoto.*

La terza condizione non è strettamente necessaria per definire un ricoprimento ma in questo contesto è utile e quindi verrà assunta.

Definizione 5. *$\mathcal{T}_r$ è una **triangolazione conforme** o semplicemente **triangolazione** di Ω se $\mathcal{T}_r$ è un ricoprimento simpliciale e se inoltre l'intersezione degli elementi di $\mathcal{T}_r$ presi a due a due è:*

- *un insieme vuoto,*
- *un vertice,*
- *uno spigolo,*
- *una faccia (se $d = 3$).*

Più generalmente, in dimensione d, l'intersezione può essere al più una k-faccia, per $k = -1, \cdots, d-1$.

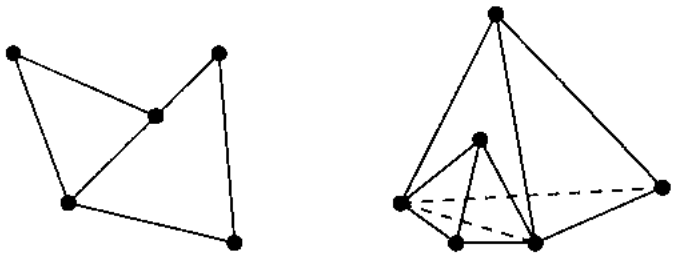

Figura 1.4: Elementi non permessi in una triangolazione

Esiste anche una relazione che riguarda il numero di k-facce in una triangolazione:

Formula di Eulero. In due dimensioni vale la seguente relazione generale:

$$ns - na + ne + c = 2,$$

dove ns è il numero dei vertici nella triangolazione, na il numero degli spigoli, ne il numero degli elementi e c il numero delle componenti connesse del bordo della triangolazione. Similmente in tre dimensioni, si ha:

$$ns - na + nf - ne = cste,$$

dove nf è il numero della facce nella triangolazione e $cste$ è una costante legata alla topologia del dominio, cioè il *genere* della superficie:

- $cste = 1$ per un dominio omeomorfo ad un boccia,
- $cste = 0$ per un dominio omeomorfo ad un toro,
- $cste = 2$ per un dominio omeomorfo ad una boccia avente una cavità sferica,
- ecc..

Esiste anche un'altra relazione che collega il numero degli elementi (ne), il numero degli spigoli interni (na_i) e il numero degli spigoli di bordo (na_f), ed è:

$$na_f - 2 \times na_i + 3 \times ne = 0$$

Inoltre, in tre dimensioni, la triangolazione di una superficie chiusa soddisfa la seguente relazione:

$$ns - na + nf = 2,$$

cioè, il numero dei vertici di bordo nella triangolazione (ns), il numero degli spigoli di bordo (na) e delle facce di bordo (nf) sono strettamente correlate.

Triangolazione di Delaunay. Tra i possibili tipi di triangolazioni, focalizzeremo l' attenzione sulla triangolazione di Delaunay. Sia $\mathcal{S}$ un insieme di punti.

Definizione 6. *$\mathcal{T}_r$ è una* ***triangolazione di Delaunay*** *di $\Omega = Conv(\mathcal{S})$ se il circumcerchio o la circumsfera (aperta)*[1] *associata ai suoi elementi è vuota.*

Questo criterio, detto anche *criterio della sfera vuota*, ci dice che la boccia aperta associata agli elementi non deve contenere nessun vertice (mentre la boccia chiusa contiene solo i vertici dell'elemento in considerazione). Questo criterio è ciò che caratterizza la triangolazione di Delaunay. Da questa proprietà ne derivano molte altre non meno importanti.
Infatti tra le differenti possibili triangolazione di S, in due dimensioni, la triangolazione di Delaunay:

- massimizza il minimo angolo formato dagli spigoli della triangolazione,
- minimizza il massimo circumraggio per ogni elemento.

Come conseguenza, una triangolazione i cui angoli non siano ottusi è una triangolazione di Delaunay.
In ogni dimensione, invece, valgono le seguenti proprietà:

- il massimo raggio della minima sfera associata agli elementi è minimo (la minima sfera di un dato elemento è la piú piccola sfera che contiene questo elemento),

[1] Considerare i dischi chiusi porta allo stesso risultato ma potrebbero formarsi elementi non simpliciali.

- in una triangolazione di Delaunay, l'unione delle circumsfere associate agli elementi che condividono un punto interno è contenuta nella stessa unione associata allo stesso punto in ogni altro tipo di triangolazione,
- la somma delle radici delle lunghezze degli spigoli pesati con la somma dei volumi degli elementi che condividono gli stessi spigoli è minima.

Inoltre se tutti i simplessi in una triangolazione contengono il loro circumcentro, allora quella triangolazione è una triangolazione di Delaunay.
Queste proprietà assicurano una certa regolarità alla triangolazione, anche se questo non esclude, soprattutto in dimensioni maggiori di due, la possibilità di ottenere degli elementi di bassa qualità.
Un problema molto importante è quello di assicurare l'esistenza, in una triangolazione, di un insieme di spigoli (in due dimensioni) o di facce (in tre dimensioni). Sia *Const* l'insieme di questi elementi.

Definizione 7. *$\mathcal{T}_r$ è una triangolazione* ***constrained*** *di Ω se gli elementi di Const sono anche elementi di $\mathcal{T}_r$.*

In particolare una triangolazione constrained può soddisfare il criterio di Delaunay localmente, eccetto che negli intorni degli spigoli e delle facce vincolate.

1.7 La mesh

Ora dobbiamo considerare un differente problema. Sia Ω un dominio chiuso e limitato di $\mathbb{R}^2$ o di $\mathbb{R}^3$, il problema è ora come costruire una triangolazione conforme di questo dominio. Tale triangolazione è chiamata **mesh** di Ω. Più precisamente:

Definizione 8. *T_h è una **mesh** di Ω se:*

1. $\Omega = \bigcup_{K \in T_h} K$.
2. *Ogni elemento K in $\mathcal{T}_h$ è non vuoto.*
3. *L'intersezione dell'interno di ogni coppia di elementi è un insieme vuoto,*
4. *L'intersezione di due elementi di $\mathcal{T}_h$ può essere:*
   - *un insieme vuoto,*
   - *un vertice,*
   - *uno spigolo,*
   - *una faccia (se $d = 3$).*

Chiaramente questa definizione è strettamente collegata alla definizione di triangolazione conforme data prima. La differenza fondamentale tra una triangolazione ed una mesh è che la triangolazione è il ricoprimento della chiusura convessa di un insieme di punti, mentre la mesh è il ricoprimento di un dato dominio, definito, in generale, dalla discretizzazione del suo bordo.
Quindi ora sorgono almeno due nuovi problemi, che sono correlati:

- il *rafforzamento*, in qualche modo, del bordo del dominio, nel senso che la triangolazione deve contenere il bordo stesso, quindi è una triangolazione constrained,
- la necessità di *costruire* un insieme di punti a partire dai vertici che definiscono il dominio, perchè in generale l'input è formato solo da questi punti.

Tuttavia, è chiaro che l'algoritmo utilizzato per costruire una mesh può essere derivato, con opportuni accorgimenti, dall' algoritmo utilizzato per costruire una triangolazione.

1.8 La qualità della mesh

Costruire una mesh non significa semplicemente trovare un qualsiasi ricoprimento del dominio che ci interessa, ma significa trovare una mesh di *buona qualità*. Chiaramente è difficile dare una definizione univoca di *qualità* di una mesh o di *mesh ottima*. L'ottimalità deve essere considerata in relazione al motivo per cui abbiamo dovuto costruire una mesh ed al suo successivo utilizzo. Ma in generale possiamo dare la seguente:

Definizione 9. *Sia Q_K la misura di qualità dell'elemento K nella mesh $\mathcal{T}$. La **qualità** di una mesh è la quantità:*

$$Q_M = \max_{K \in \mathcal{T}} Q_K.$$

Ci sono anche altri valori adatti a valutare la qualità di una mesh, per esempio la media delle qualità degli elementi, la distribuzione degli elementi basata sulla loro qualità, ecc.

In ogni caso, per come abbiamo definito la qualità di un elemento, è chiaro che una mesh è di buona qualità quanto più vicino ad 1 è il valore Q_M. Tuttavia è anche da considerare che, per gli scopi pratici, è importante tener conto del numero degli elementi o dei vertici di una mesh. Ad esempio, per minimizzare il costo computazionale nella simulazione numerica, è importante che il numero degli elementi di una mesh sia minimo. Si può quindi concludere che

una mesh è ottima se dà un adeguato compromesso tra qualità degli elementi e numero degli elementi.

Capitolo 2

La Triangolazione

2.1 Introduzione

Lo scopo di questo capitolo è quello di esaminare i vari tipi di triangolazioni e in special modo quella di Delaunay.
Gli utilizzi di questa particolare triangolazione sono molteplici: per la soluzione di problemi reali formulati in termini di equazioni alle derivate parziali, nella robotica, nel trattamento delle immagini, per quello che riguarda il CAD, nelle scienze geografiche, e così via.
Gli studi sulle triangolazioni risalgono già al tempo di Euclide (330-270 a.C.), ma in epoca più recente possiamo riferirci a Peter Gustav Dirichlet (1805-1859), Georgy Voronoy (1868-1908) e Boris Delone o Delaunay (1890-1980) come ai padri dello sviluppo moderno delle triangolazioni.
Qui considereremo lo spazio $\mathbb{R}^d$ con l'usuale metrica euclidea (con $d = 2$ o $d = 3$). Dato un insieme di punti $\mathcal{S}$, ci proponiamo di costruire la triangolazione di $Conv(\mathcal{S})$.

2.2 Diagramma di Voronoy e Triangolazione di Delaunay

Alla fine del 19° secolo Dirichlet dimostrò che è possibile, per un insieme di punti in due dimensioni, dividere il piano in celle basandosi su criteri di vicinanza. Più tardi Voronoy, studiando le forme quadratiche estese il criterio di Dirichlet allo spazio tridimensionale.
Sia $\mathcal{S}$ un insieme di n punti, o *siti*, P_i, $i = 1, ..., n$, in dimensione d.

Definizione 10. *Una **cella di Voronoy**, $V(A)$, di un sottoinsieme A di $\mathcal{S}$ è il luogo dei punti di $\mathbb{R}^d$ equidistanti da ogni sito in A e più vicino ad A che ad ogni altro sito che non è in A.*

Quindi per il singoletto $A = \{a\}$, $V(A)$ è l'insieme di punti più vicini ad a che a qualsiasi altro sito.
$V(A)$ può essere vuoto o perchè non ci sono punti equidistanti da ogni $a \in A$ o perchè ogni punto equidistante da ogni $a \in A$ è anche equidistante da un $a' \in \mathcal{S} - A$.

Definizione 11. *Un **diagramma di Voronoy**, V, è una collezione di celle di Voronoy, non vuote, $V(A)$, per ogni sottoinsieme A di $\mathcal{S}$.*

Le celle di Voronoy sono poligoni (poliedri in tre dimensioni) chiusi e convessi. Così le celle aperte e disgiunte ricoprono tutto lo spazio e costituiscono, nel piano (se $d = 2$), la *tassellazione di Dirichlet* o più in generale il diagramma di Voronoy in $\mathbb{R}^d$.
Basandoci su questa definizione, osserviamo che ogni cella di Voronoy è associata ad un punto e che in due dimensioni ogni cella ha il bordo costituito

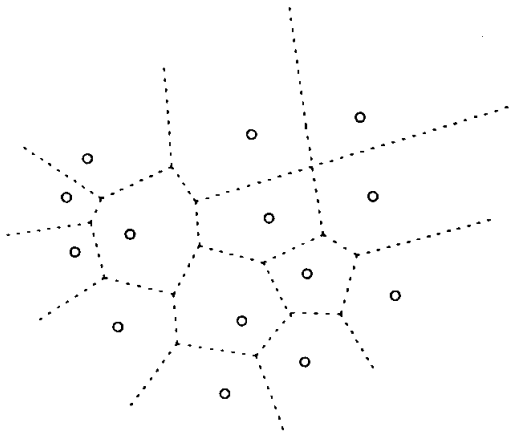

Figura 2.1: Diagramma di Voronoy in due dimensioni.

alternativamente da *lati di Voronoy* (celle 1-dimensionali) e *vertici di Voronoy* (celle 0-dimensionali); in tre dimensioni avremo anche *facce di Voronoy* (celle 2-dimensionali). Osserviamo anche che, se consideriamo il duale del diagramma di Voronoy, otteniamo una triangolazione di Delaunay. Questo fu il fondamentale risultato ottenuto da Delaunay e da lui pubblicato nel 1934 nel trattato *"Sur la sphère vide"*. Prima di analizzare questo importante risultato, diamo alcune definizioni.

Definizione 12. *Sia $A \subset \mathcal{S}$ e $V(A) \neq \emptyset$ una cella di Voronoy. Si definisce* ***cella di Delaunay****, $T(A)$, la chiusura convessa di A.*

Definizione 13. *Una* ***triangolazione di Delaunay*** *$\mathcal{T}$ è la collezione di tutte le celle di Delaunay $T(A)$, $\forall A \subset \mathcal{S}$ con $V(A) \neq \emptyset$.*

Teorema 1. *Sia $\mathcal{S}$ un insieme di* n *punti in $\mathbb{R}^d$ con diagramma di Voronoy V e triangolazione di Delaunay $\mathcal{T}$, allora V e $\mathcal{T}$ sono duali, cioè, per A, $A' \subseteq \mathcal{S}$, $V(A)$ è una faccia di $V(A')$ se e solo se $T(A')$ è un faccia di $T(A)$.*

dim. Innanzi tutto osserviamo che se $V(A)$ e $V(A')$ sono celle di Voronoy non vuote, allora:

$$V(A') \text{ è una faccia di } V(A) \Leftrightarrow A \subset A'$$

($\Leftarrow$) Sia $A \subset A'$. Per ogni $a \in A$ e per ogni $b \in A' - A$, sia $H_{a,b}$ l'iperpiano equidistante da a e b. Consideriamo come semispazio positivo, il semispazio $\Sigma_{r,s}$ delimitato da $H_{a,b}$ e contenente A. Sia C la chiusura di $V(A)$.
$V(A)$ è contenuto nell'intersezione di tutti i semispazi positivi, quindi C è contenuto nell'intersezione delle chiusure di tutti i semispazi positivi, cioè:

$$C \subset \bigcap_{a,b} (\Sigma_{a,b} \cup H_{a,b})$$

Sia $I = \cap H_{a,b}$, $\forall a \in A$ e $\forall b \in A' - A$. $I \neq \emptyset$, altrimenti non esisterebbero punti equidistanti tra i siti di A e i siti di A', ed essendo $A \subset A'$ risulterebbe $V(A') \doteq \emptyset$. Questo è assurdo perchè per ipotesi $V(A') \neq \emptyset$.
Sia H l'iperpiano passante per I che evita l' intersezione tra semispazi positivi e l'intersezione tra i semispazi negativi e tale che $C \cap I = C \cap H$. Allora H è un iperpiano di supporto a $V(A)$, cioè uno dei suoi semispazi chiusi contiene $V(A)$ e l'iperpiano interseca la chiusura di $V(A)$. Allora l'interno di $C \cap I$ è una faccia di $V(A)$.
Poichè $V(A')$ è un sottinsieme aperto di I e poichè in C ci sono anche punti equidistanti da tutti i siti di A', possiamo dire che $V(A')$ coincide con l'interno di $C \cap I$, quindi $V(A')$ è una faccia di $V(A)$.
($\Rightarrow$) Ogni punto di una faccia di $V(A)$ è nella chiusura di $V(A)$. Ogni $a \in V(A)$ è equidistante da ogni altro sito $b \notin A$. Se consideriamo $A' = A \cup b$, allora $a \in A'$ con $A' \supset A$.

Allora se $T(A)$ e $T(A')$ sono due celle di Delaunay, si ha:
$V(A')$ è una faccia di $V(A) \Leftrightarrow A \subset A' \Leftrightarrow A = A' \cap H$ con H iperpiano che supporta la chiusura convessa di $A' \Leftrightarrow T(A)$ è una faccia di $T(A')$.

□

Se i punti da triangolare sono in posizione generale vale il risultato predente e la triangolazione di Delaunay è unica. Se, invece, i punti dati includono $d+2$ punti cociclici (cosferici), il risultato precedente vale ancora e il duale del diagramma di Voronoy è un unico ricoprimento dell'insieme di punti dato che però include poligoni convessi diversi da simplessi. Poichè nelle applicazio-

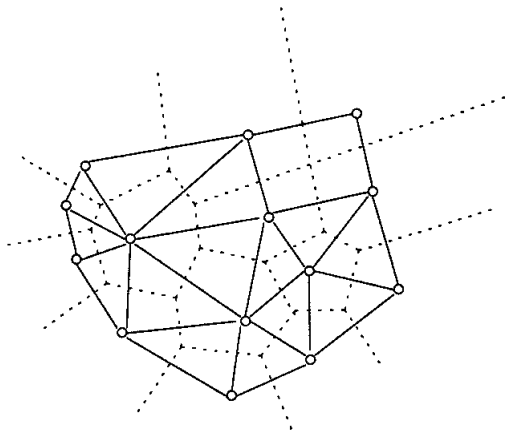

Figura 2.2: Ricoprimento di Delaunay di un insieme che contiene quattro punti cociclici.

ni, e soprattutto nel metodo ad elementi finiti, si usano triangoli e tetraedri, a partire dal ricoprimento di cui sopra possiamo ottenere una triangolazione dividendo i poligoni in triangoli e i poliedri in tetraedri.
E' chiaro che sono possibili parecchie divisioni e quindi parecchie triangolazio-

ni. Ad esempio, nella figura 2.2, esistono due possibili modi per dividere il quadrilatero in due triangoli ma una sola di queste rispetterà il criterio della sfera vuota.

Nello stesso trattato Delaunay dimostrò anche il seguente risultato che è alla base dei più importanti algoritmi per la costruzione di triangolazioni di Delaunay.

Lemma 1. *Sia $\mathcal{T}$ una arbitraria triangolazione della chiusura convessa di un insieme di punti $\mathcal{S}$. Se per ogni coppia di simplessi adiacenti in $\mathcal{T}$ vale il criterio della sfera vuota, allora questo criterio vale globalmente e $\mathcal{T}$ è una triangolazione di Delaunay.*

dim. Prima di tutto osserviamo che il criterio di Delaunay è *simmetrico*. Consideriamo due simplessi adiacenti che dividono una comune $(d-1)$-faccia e sia P (risp. Q) il vertice in questi simplessi opposti a questa faccia. Allora:

$$Q \notin B_P \Leftrightarrow P \notin B_Q$$

dove B_P (risp. B_Q) è la boccia associata al simplesso che ha P (risp. Q) come vertice (Figura 2.3).

Allora se $Q \notin B_P$, si ossserva che:

$$B_P \cap H_Q \subset B_Q \cap H_Q$$

o viceversa,

$$B_Q \cap H_P \subset B_P \cap H_P$$

dove, considerando la stessa configurazione per i due simplessi, H_P (risp. H_Q) è il semispazio contenente P (risp. Q) limitato dall'iperpiano che supporta la

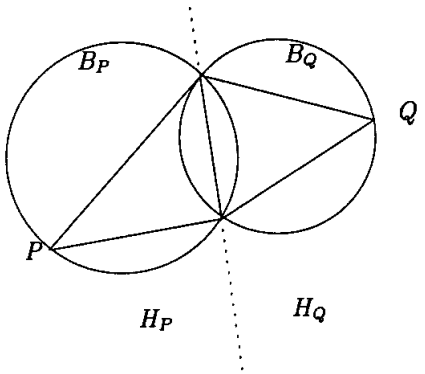

Figura 2.3: Bocce ed iperpiani.

$(d-1)$-faccia comune. E questo ci porta alla conclusione.

Ora assumiamo che il criterio della sfera vuota valga per ogni coppia di elementi adiacenti e che esista nella triangolazione un punto P_n contenuto nella boccia associata al simplesso che non è adiacente ai simplessi che hanno P_n come vertice. Sia K_0 questo simplesso (Figura 2.4). Consideriamo il punto G, che è il baricentro di K_0, e consideriamo il segmento $\overline{GP_n}$. Questa linea congiunge il simplesso K_0 al simplesso che indichiamo K_n (uno dei simplessi della boccia di P_n) ed interseca un insieme di simplessi, indichiamoli con $K_0, K_1, K_2, ..., K_n$. Se i punti sono in posizione generale questi n simplessi hanno intersezione contenuna in una $(d-1)$-faccia. Se i punti non sono in posizione generale cambiando la scelta del punto G possiamo ricondurci a questo caso.

Per ipotesi si ha:

$$P_n \in B_0,$$

dove B_i è la boccia associata al simplesso K_i e K_{n-1} è adiacente a K_n, dopo il

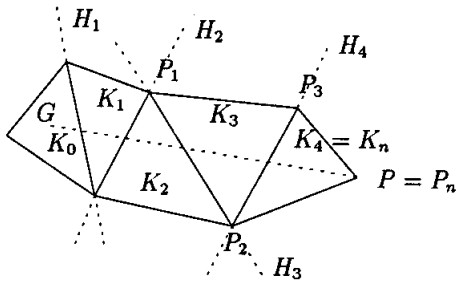

Figura 2.4: Il simplesso K_0.

criterio della sfera vuota per i simplessi adiacenti:

$$P_n \notin B_{n-1}.$$

I simplessi K_i sono scelti in base alla loro intersezione con il segmento $\overline{GP_n}$. Sia P_i il vertice di K_i che non appartiene a K_{i-1} e sia H_i il semispazio contenente K_i ma non K_{i-1}. Secondo la definizione, si ha che $P_n \in H_i$.
Abbiamo quindi provato che esiste un indice i tale che

$$P_n \in B_i$$

perciò

$$P_n \in B_{i+1}.$$

Infatti, poichè $P_n \in H_{i+1}$ e $B_i \cap H_{i+1} \subset B_{i+1} \cap H_{i+1}$, allora $P_n \in B_{i+1}$. Dato che $P_n \in B_0$, allora si ha che $P_n \in B_{n-1}$ contro ciò che avevamo asserito.

□

Definizione 14. *Siano P_i e P_j due siti in $\mathcal{S}$ e sia $\mathcal{D}$ la triangolazione di Delaunay di $\mathcal{S}$. Il lato che congiunge P_i e P_j è un* ***lato di Delaunay*** *se*

appartiene alla triangolazione $\mathcal{D}$, cioè se esiste un cerchio vuoto (sfera vuota) che passa per P_i e P_j. Dove con cerchio vuoto (sfera vuota) si intende che non contiene nessun vertice di $\mathcal{S}$ al suo interno.

Definizione 15. *Un elemento K di una triangolazione di Delaunay $\mathcal{D}$ è detto* ***elemento Delaunay****.*

Lemma 2. *Sia $\mathcal{T}$ una triangolazione. Tutti gli elementi di $\mathcal{T}$ sono Delaunay se e solo se tutti i lati di $\mathcal{T}$ sono Dealunay.*

dim. Consideriamo il caso bidimensionale.
($\Rightarrow$) Se tutti i triangoli di $\mathcal{T}$ sono Delaunay, allora il circumcerchio di ogni triangolo è vuoto. Poichè ogni lato di $\mathcal{T}$ appartiene ad un triangolo di $\mathcal{T}$, ogni lato è contenuto in un cerchio vuoto e così è Delaunay.
($\Leftarrow$) Supponiamo che tutti i lati di $\mathcal{T}$ siano Delaunay e che, per assurdo, esista un triangolo K che non sia Delaunay. Poichè $\mathcal{T}$ è una triangolazione, K non può contenere nessun sito eccetto i suoi tre vertici, così almeno un sito P_i è contenuto nel circumcerchio di K ma non nel triangolo stesso. Sia e_K il lato di K che divide P_i dall'interno di K e sia p_j il vertice di K opposto a e_K. Osserviamo che è possibile tracciare un cerchio che contiene e_K ma non contiene nè P_i nè P_j, il che significa che il lato e_K è Delaunay. E questo ci porta ad una contraddizione.

□

2.3 Algoritmi per la costruzione di Triangolazioni di Delaunay

Uno dei metodi per costruire una triangolazione di Delaunay e quello di sfruttare la sua dualità con il diagramma di Voronoy. Esistono anche altri metodi di cui più avanti parleremo, ma quello su cui focalizzeremo la nostra attenzione è il metodo incrementale noto anche come *algoritmo di Boyer-Watson.*

Il metodo incrementale

Sia $\mathcal{T}_i$ la triangolazione di Delaunay della chiusura convessa dei primi i punti in $\mathcal{S}$, consideriamo l'$(i+1)$-esimo punto di questo insieme ed indichiamolo con P. Lo scopo del metodo incrementale è quello di ottenere $\mathcal{T}_{i+1}$, la triangolazione di Delaunay che include il punto P come vertice, partendo da $\mathcal{T}_i$. A questo scopo introduciamo la procedura detta *Nucleo di Delaunay.*

Nucleo di Delaunay. E' una procedura locale come segue:

$$\mathcal{T}_{i+1} = \mathcal{T}_i - \mathcal{C}_P + \mathcal{B}_P$$

dove $\mathcal{C}_P$ è la cavità associata a P, cioè l'insieme degli elementi della triangolazione il cui circumcerchio (o circumsfera) contiene il punto P e $\mathcal{B}_P$ è l'insieme dei nuovi elementi della triangolazione che hanno P come vertice. Vediamo come costruire $\mathcal{B}_P$.

P rispetto a $\mathcal{T}_i$ può essere:

1. contenuto nella triangolazione $\mathcal{T}_i$,
2. esterno alla triangolazione $\mathcal{T}_i$.

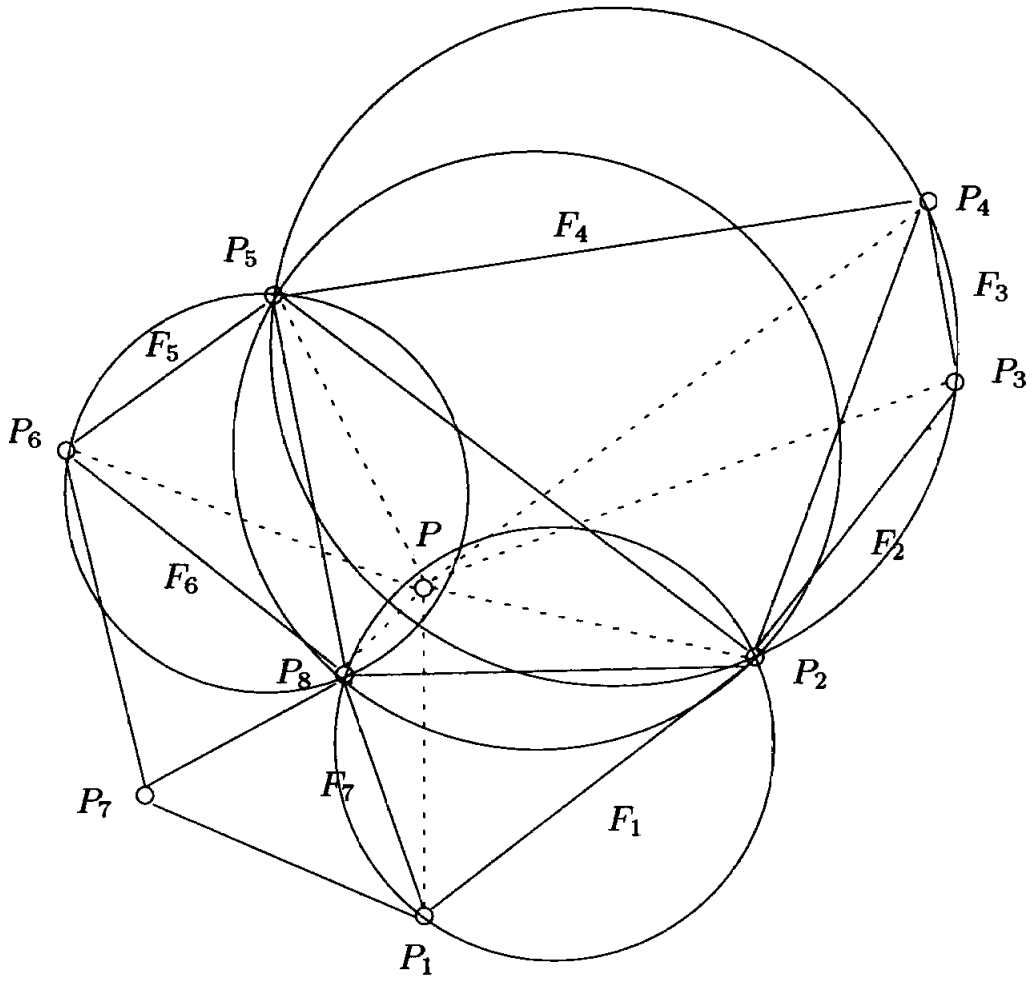

Figura 2.5: Inserimento di P: $P \in \mathcal{T}_i$.

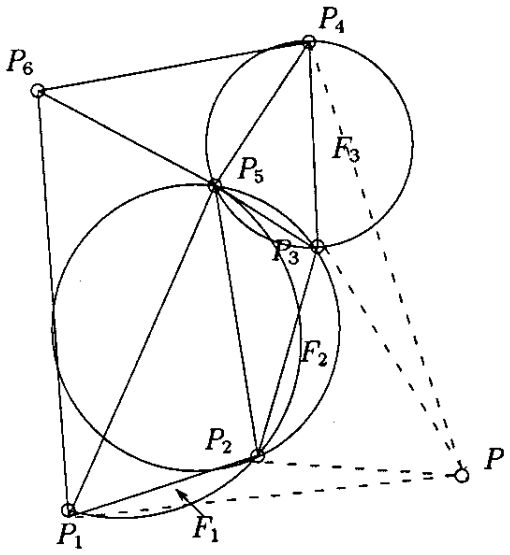

Figura 2.6: Inserimento di P: P fuori da $\mathcal{T}_i$ e da ogni circumdisco.

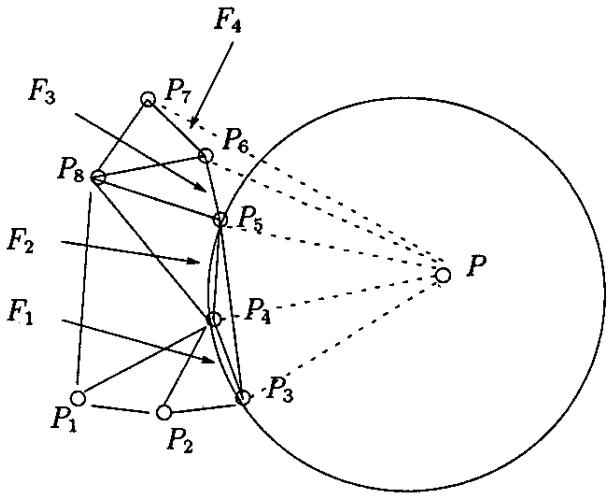

Figura 2.7: Inserimento di P: P fuori da $\mathcal{T}_i$ e dentro un circumdisco.

Nel primo caso (Figura 2.5) $\mathcal{B}_P$ è l'insieme di tutti i nuovi elementi ottenuti congiungendo P con tutti i lati (facce) di $\mathcal{T}_i$ visibili da P, dopo aver tolto $\mathcal{C}_P$. Ricordiamo che un lato (faccia) è detto *visibile* da un dato punto se l'elemento formato dal lato (faccia) e dal punto non interseca nessun elemento della triangolazione.

Se P è esterno a $\mathcal{T}_i$, dobbiamo ancora distinguere due casi:

(2a) P è esterno anche ad ogni circumcerchio (circumdisco) di $\mathcal{T}_i$.

(2b) P è contenuto in almeno un circumcerchio (circumdisco) di $\mathcal{T}_i$.

Nel caso (2a), $\mathcal{B}_P$ è dato semplicemente dagli elementi creati congiungendo P con i lati (facce) visibili da P (Figura 2.6).

Nel caso (2b) invece, $\mathcal{B}_P$ è dato dagli elementi creati congiungendo P con i lati (facce) visibili da P ma dopo aver tolto $\mathcal{C}_P$ (Figura 2.7).

Il risultato più importante per la costruzione del metodo è il seguente:

Teorema 2. *Sia $\mathcal{T}_i$ la triangolazione di Delaunay della chiusura convessa dei primi* i *punti dell'insieme $\mathcal{S}$, allora $\mathcal{T}_{i+1}$ è la triangolazione di Delaunay di questa chiusura che include il punto P, $(i+1)$-esimo punto dell'insieme, come vertice.*

dim. Prima di tutto assumiamo che i punti siano in posizione generale. La triangolazione del complementare di $\mathcal{C}_P$ rimane invariata, e anche la boccia circoscritta a questi simplessi è vuota e così questi simplessi sono Delaunay (segue dal criterio della sfera vuota). Vogliamo dimostrare che la sola possibile triangolazione di $\mathcal{C}_P$ quando consideriamo il punto P, è la boccia $\mathcal{B}_P$ definita congiungendo P con le facce esterne di $\mathcal{C}_P$.

Assumiamo per assurdo che un simplesso della nuova triangolazione non annoveri P come vertice, così necessariamente questo simplesso appartiene a $\mathcal{C}_P$, perchè i punti sono in posizione generale. In questo caso la triangolazione della cavità è unica essendo il duale del diagramma di Voronoy. Perciò questo simplesso viola il criterio di Delaunay. In altre parole la nuova triangolazione è composta da simplessi formati congiungendo P con le facce esterne di $\mathcal{C}_P$, che significa che $\mathcal{B}_P$ è la nuova triangolazione di $\mathcal{C}_P$.
Nel caso in cui i punti non siano in posizione generale, assumiamo che un simplesso, diciamo K, nella nuova triangolazione, non abbia P come vertice. Allora, esistono uno o più simplessi in $\mathcal{C}_P$ la cui circumsfera è identica a quella di K. Così, K viola il criterio di Delaunay e otteniamo il risultato precedente.

□

Osservazione: Qui abbiamo dimostrato solo il caso 1 (Figura 2.5) usando la dualità tra il diagramma di Voronoy e la triangolazione di Delaunay. In realtà potevamo anche procedere con una dimostrazione in due passi provando prima che $\mathcal{T}_{i+1}$ è una triangolazione valida e poi che è una triangolazione di Delaunay.

In pratica occorre applicare la procedura del nucleo di Delaunay ad ogni punto dell'insieme $\mathcal{S}$, partendo da un elemento (triangolo o tetraedro) formato scegliendo $d+1$ punti affinemente indipendenti in modo da definire $\mathcal{T}_{d+1}$, dove d è la dimensione dello spazio. Il problema del metodo è trovare la cavità di P. La procedura che segue ci aiuta in questo compito.
La misura di Delaunay. La misura di Delaunay è una caratterizzazione che ci permette di costruire la cavità associata ad un dato punto. Ricordiamo

che $\mathcal{C}_P$ è l'insieme degli elementi il cui circumcerchio (circumsfera) contiene il punto P. Se indichiamo con $d(P, O_K)$ la distanza euclidea tra il punto P e il punto O_K, che è il circumcentro del cerchio (sfera) associata all'elemento K con raggio r_K, allora un elemnto K in $\mathcal{T}_i$ è un membro di $\mathcal{C}_P$ se:

$$d(P, O_K) - r_K < 0,$$

oppure,

$$\frac{d(P, O_K)}{r_K} < 1$$

Definizione 16. *Il rapporto* $\frac{d(P,O_K)}{r_K}$*, indicato con* $\alpha(P, K)$*, è detto* ***misura di Delaunay*** *del punto* P *rispetto all'elemento* K*.*

Osserviamo che $\alpha(P, K)$ è misurato con la usuale metrica euclidea e quindi è indipendente dal punto P. Quindi la caratterizzazzione della cavità di P è data da:

$$K \in \mathcal{C}_P \Leftrightarrow \alpha(P, K) < 1.$$

Il metodo incrementale ridotto.

La versione ridotta del metodo incrementale è basata su un'appropriata definizione della boccia contenente i punti dell'insieme $\mathcal{S}$, $Box(\mathcal{S})$, in modo tale da ritrovarci solo nel caso 1, cioè in modo che P sia sempre incluso nella precedente triangolazione. Questo stratagemma permette di semplificare, e di molto, l'algoritmo. Una volta che abbiamo terminato l'inserimento di tutti i punti di $\mathcal{S}$, quello che otteniamo è la triangolazione di Delaunay di $Box(\mathcal{S})$. Ora è

possibile ottenere la triangolazione della chiusura di $\mathcal{S}$, non necessariamente della chiusura convessa, rimuovendo qualche elemento dalla triangolazione.
I passi principali da fare per usare questo algoritmo sono:

1. analizzare $\mathcal{S}$ e selezionare gli estremi,
2. costruire $Box(\mathcal{S})$,
3. triangolare $Box(\mathcal{S})$ (sia $\mathcal{T}_0$ questa triangolazione e sia $i = 0$),
4. inserire ogni punto di $\mathcal{S}$ nella triangolazione, cioè:
   - cercare gli elementi in $\mathcal{T}_i$ che contengono il punto in considerazione (quella che è chiamata ricerca della *base* di P),
   - costruire la cavità di P usando $(d-1)$-facce adiacenti, partendo dalla base,
   - rimuovere la cavità, numerare gli elementi che hanno P come vertice, sostituirli alla cavità, scegliere $i = i + 1$ e prendere in considerazione un nuovo punto di $\mathcal{S}$.

Per triangolare un insieme di n punti con questo metodo si impiega un tempo pari ad $\mathcal{O}(n^2)$ nel peggiore dei casi, in particolare per punti cociclici in due dimensioni e cosferici in tre dimensioni.
Tuttavia, in pratica ha un comportamento lineare soprattutto nel caso in cui i punti non siano tantissimi, sia in due che in tre dimensioni.
Con questo metodo, in due dimensioni, ogni triangolazione di n punti contiene $\mathcal{O}(n)$ triangoli, in tre dimensioni contiene $\mathcal{O}(n^2)$ tetraedri.

Flip algorithm

L'algoritmo di cui ora parleremo si basa su trasformazioni locali o *flips* e fu per la prima volta introdotto da Lawson nel 1972 per costruire triangolazioni di Delaunay in due dimensioni. L'algoritmo è noto anche come *Lawson's flip method.*

Consideriamo un insieme finito di punti $\mathcal{S} \subset \mathbb{R}^d$ ($d = 2$ o $d = 3$). Il metodo comincia costruendo un'arbitraria triangolazione $\mathcal{T}$ di $\mathcal{S}$. Questa triangolazione viene poi modificata passo dopo passo con una sequenza di flips in modo che tutti i lati siano Delaunay, così da ottenere la triangolazione di Delaunay $\mathcal{D}$ di $\mathcal{S}$. Abbiamo già dimostrato (Lemma 1) che se i lati di una triangolazione sono tutti localmente Delaunay allora tutta la triangolazione è globalmente Delaunay. Una conseguenza di questo lemma è che se una triangolazione contiene un lato che non è Delaunay, allora contiene un lato che non è localmente Delaunay così da poter attuare il flip algorithm.

Flips in $\mathbb{R}^2$. Consideriamo un lato $e_{i,j} = (P_i, P_j)$ in $\mathcal{T}$. Se $e_{i,j}$ appartiene al bordo di $Conv(\mathcal{S})$ allora appartiene alla triangolazione di Delaunay di $\mathcal{S}$. Altrimenti, è un lato in comune a due triangoli, $f_{i,j,k}$ e $f_{i,j,u}$, in $\mathcal{T}$. Diciamo che $e_{i,j}$ è *localmente Delaunay* se appartiene alla triangolazione di Delaunay dei punti P_i, P_j, P_k e P_u. Se $e_{i,j}$ non è localmente Delaunay allora non può appartenere nemmeno alla triangolazione di Delaunay di $\mathcal{S}$. In questo caso si sostituisce $e_{i,j}$ con $e_{k,u} = (P_k, P_u)$. La sostituzione è lecita poichè, come vedremo fra poco, l'unione di $f_{i,j,k}$ e $f_{i,j,u}$ è convessa. Questa operazione è chiamata *edge flip* in $\mathbb{R}^2$ e il lato $e_{i,j}$ è detto *flippable*. Chiaramente $e_{k,u}$ è localmente Delaunay, vale infatti il seguente:

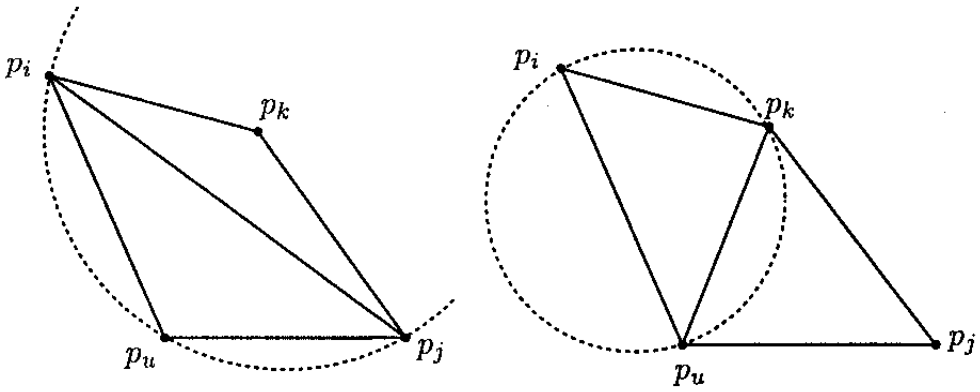

Figura 2.8: Le due triangolazioni dell'insieme $\{p_i, p_j, p_k, p_u\}$ in $\mathbb{R}^2$.

Lemma 3. *Sia $e_{i,j}$ un lato della triangolazione di $\mathcal{S}$. Allora, o $e_{i,j}$ è flippable o il lato creato dal flip è localmente Delaunay.*

dim. Siano P_u e P_k i vertici opposti ad $e_{i,j}$ che insieme a P_i e P_j definiscono un quadrilatero che ha come diagonale $e_{i,j}$.
Sia $\mathcal{C}$ il cerchio che passa per P_i, P_j e P_k. Allora, o P_u è fuori da $\mathcal{C}$ o è dentro $\mathcal{C}$. Se P_u è fuori da $\mathcal{C}$ allora $e_{i,j}$ è localmente Delaunay. Se P_u è dentro $\mathcal{C}$ allora P_u è contenuto nella parte di cerchio definita da $e_{i,j}$ ed opposta ad e_k. Allora il quadrilatero che contiene $e_{i,j}$ è vincolato ad essere strettamente convesso ed $e_{i,j}$ è flippable. Inoltre il cerchio che passa per e_k ed e_u e che è tangente a $\mathcal{C}$ in e_k non contiene gli estremi di $e_{i,j}$ così $e_{k,u}$ è localmente Delaunay.

□

Sarà utile introdurre la definizione di *mappa di sollevamento* che può trasformare il problema di creare la triangolazione di Delaunay di un insieme di n punti $\mathcal{S} \subset \mathbb{R}^2$ nel problema di costruire una chiusura convessa in $\mathbb{R}^3$.

Identifichiamo $\mathbb{R}^2$ con lo spazio xyz in $\mathbb{R}^3$ con $z = 0$. La mappa di sollevamento è una trasformazione geometrica che proietta il punto $P = (\pi_1, \pi_2) \in \mathbb{R}^2$ lungo l'asse z sul paraboloide di rivoluzione $U : z = (x^2 + y^2) \in \mathbb{R}^3$.
Sia $P_U = (\pi_1, \pi_2, {\pi_1}^2 + {\pi_2}^2)$ l'immagine di P e definiamo $S_U = \{P_U : P \in \mathcal{S}\}$.
La chiusura convessa di $\mathcal{S}_U$, $Conv(\mathcal{S}_U)$, è un poliedro convesso con n vertici in $\mathbb{R}^3$. Una faccia del poliedro è sul bordo "superiore" del poliedro stesso se,

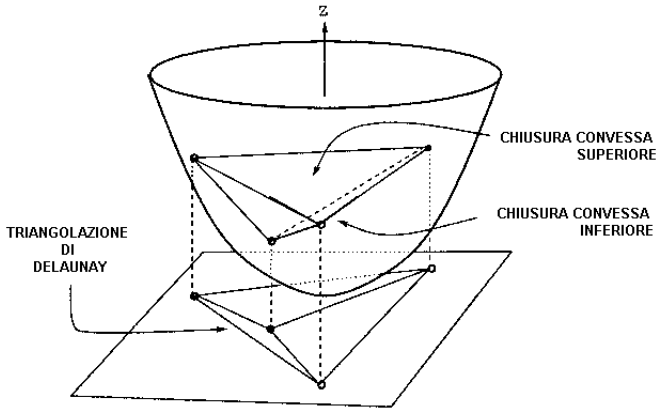

Figura 2.9: La mappa di sollevamento.

considerando il terzo asse del piano che contiene la faccia, essa giace dalla parte positiva di questo stesso asse. Altrimenti appartiene al bordo "inferiore" di questo poliedro.
Se proiettiamo tutte le facce sul bordo inferiore di $Conv(\mathcal{S}_U)$ parallelamente all'asse z in $\mathbb{R}^2$ allora otteniamo la triangolazione di Delaunay di $\mathcal{S}$.
Tenendo conto di questo, possiamo osservare che la triangolazione $\mathcal{T}$ è la triangolazione di Delaunay $\mathcal{D}$ se e solo se la sua proiezione $\mathcal{T}_U$ è esattamente il bordo inferiore di $Conv(\mathcal{S}_U)$, chiamiamola $\mathcal{D}_U$. Se un lato $e_{i,j}$ non è localmente Delaunay allora la sua proiezione non fa parte di $\mathcal{D}_U$. In altre parole un flip in

$\mathbb{R}^2$ corrisponde ad unire tetraedri in $\mathbb{R}^3$.

Questo procedimento termina quando $\mathcal{T}_U = \mathcal{D}_U$, cioè quando $\mathcal{T}$ diventa una triangolazione di Delaunay.

Si può dimostrare che, il numero di flip da attuare per rendere Delaunay una qualsiasi triangolazione di n siti in $\mathbb{R}^2$ è, nel peggior caso, un $\mathcal{O}(n^2)$. Ciò rende il Lawson's flip method un algoritmo quadratico.

Flips in $\mathbb{R}^3$. Per generalizzare il flip algorithm in tre dimensioni dobbiamo

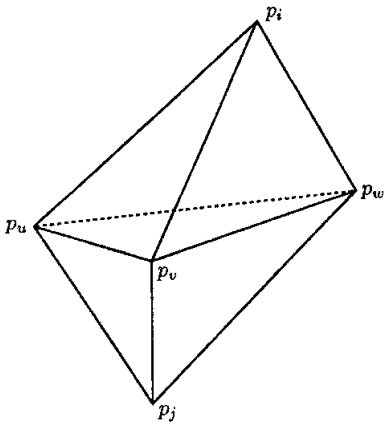

Figura 2.10: Prima triangolazione dell'insieme $\{p_i, p_j, p_k, p_u, p_v\}$ in $\mathbb{R}^3$. E' composta da due tetraedri: $t_{i,j,k,u}$ e $t_{i,j,k,v}$.

prima specificare che cosa intendiamo per flip in $\mathbb{R}^3$.

Consideriamo un insieme $\mathcal{S}'$ di cinque punti in $\mathbb{R}^3$ in posizione generale. Se esiste un punto $P \in \mathcal{S}'$ tale che $Conv(\mathcal{S}') = Conv(\mathcal{S}' - \{P\})$ allora $\mathcal{S}'$ ha una sola possibile triangolazione con quattro tetraedri tutti con vertice in P. Questa triangolazione è banalmente di Delaunay. Altrimenti tutti i punti di $\mathcal{S}'$ giacciono sul bordo di $Conv(\mathcal{S}')$ ed $\mathcal{S}'$ ha solo due possibili triangolazioni, una con due tetraedri e una con tre tetraedri: una di queste deve essere una

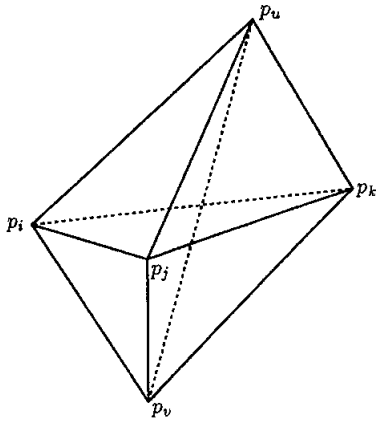

Figura 2.11: Seconda triangolazione dell'insieme $\{p_i, p_j, p_k, p_u, p_v\}$ in $\mathbb{R}^3$. E' composta da tre tetraedri: $t_{u,v,i,j}$, $t_{u,v,j,k}$ e $t_{u,v,k,i}$.

triangolazione di Delaunay.

Similmente al caso bidimensionale, occorre partire da una triangolazione iniziale di $\mathcal{S}$ e prendere in considerazione una faccia di un tetraedro (cioè un triangolo) che non sia localmente Delaunay.

Un triangolo $f_{i,j,k} = (P_i, P_j, P_k)$ è detto *localmente Delaunay* se appartiene al bordo della triangolazione o se è una faccia comune a due tetraedri, $t_{i,j,k,u} = (P_i, P_j, P_k, P_u)$ e $t_{i,j,k,v} = (P_i, P_j, P_k, P_v)$, e P_v non appartiene alla circumsfera determinata da P_i, P_j, P_k e P_u. Osserviamo che se P_v non appartiene alla circumsfera determinata da P_i, P_j, P_k e P_u allora P_u non appartiene alla circumsfera determinata da P_i, P_j, P_k e P_v.

Come illustrato nelle figure 2.10 e 2.11 distinguiamo due differenti flips per facce che non sono localmente Delaunay.

1. Sia $f_{i,j,k}$ un triangolo in $\mathcal{T}$ che non sia localmente Delaunay, e siano $t_{i,j,k,u}$

e $t_{i,j,k,v}$ i due tetraedri incidenti. Se l'unione dei due tetraedri è convessa, allora possono essere sostituiti con i tre tetraedri: $t_{u,v,i,j}$, $t_{u,v,j,k}$ e $t_{u,v,k,i}$ ed ognuno di loro è localmente Delaunay. Questo flip è detto *flip 2-3* o *triangle-to-edge flip*.

2. Sia $e_{u,v}$ un lato della triangolazione $\mathcal{T}$ che non è localmente Delaunay ed appartiene a tre tetraedri $t_{u,v,i,j}$, $t_{u,v,j,k}$ e $t_{u,v,k,i}$. Questi possono essere sostituiti con due tetraedri $t_{i,j,k,u}$, $t_{i,j,k,v}$ che sono localmente Delaunay. Si noti tra l'altro che il fatto che $e_{u,v}$ non sia Delaunay implica che $t_{i,j,k,u} \cup t_{i,j,k,v}$ sia convesso. Questo flip è chiamato *flip 3-2* o *edge-to-triangle flip*.

Sfortunatamente non è proprio così semplice cercare di generalizzare il flip algorithm in tre dimensioni. Infatti Joe (in [11]) ha dimostrato che il problema è che la triangolazione può contenere una faccia che non sia localmente Delaunay e a cui non sia possibile applicare il flip algorithm perchè l'esaedro che la contiene non è convesso, oppure può contenere un lato che non sia localmente Delaunay e che non può essere modificato perchè appartiene a più di tre tetraedri. In realtà Joe dimostra anche il seguente risultato:

Lemma 4. *Se un singolo punto P è aggiunto alla triangolazione di Delaunay dell'insieme di punti $\mathcal{S}$ in $\mathbb{R}^3$, allora esiste una sequenza di flips che danno come risultato la triangolazione di Delaunay di $\mathcal{S} \cup \{P\}$.*

Questo risultato è alla base dell'*incremental flip algorithm* in tre dimensioni. L'idea di base di questo algoritmo è la seguente. Supponiamo di avere un insieme $\mathcal{S}$ di n punti $\mathbb{R}^3$. Sia $4 < i \leq n$ e assumiamo che sia stata costruita la triangolazione di Delaunay dei primi $i-1$ punti di $\mathcal{S}$; chiamiamola $\mathcal{D}_{i-1}$. Si ag-

giunge l'i-esimo punto, $P_i \in \mathcal{S}$, alla triangolazione e tramite i flips ci si riporta ad avere tutti i lati e le facce localmente Delaunay e quindi la triangolazione che ne risulta è una triangolazione di Delaunay, $\mathcal{D}_i$. Si ripete il procedimento fino a che $i = n$.

Il problema dell'incremental flip algorithm è l'ordine in cui i vari siti debbano essere inseriti all'interno della triangolazione $\mathcal{D}_{i-1}$. Joe ha risolto questo problema inserendo i punti in ordine lessicografico, cioè muovendosi sempre nella stessa direzione, ad esempio l'asse x, così P_i sta fuori della chiusura di $\mathcal{D}_{i-1}$ ed è facile trovare le facce che sono visibili da P_i, cioè le facce con cui va unito il punto per ottenere i nuovi tetraedri. Joe ha dimostrato che, per un insieme di n punti in $\mathbb{R}^3$ il metodo impiega, nel caso peggiore, un tempo quadratico per costruire la triangolazione.

Edellsbrunner e Shah hanno dimostrato, invece, che l'algoritmo risulta molto più veloce se i punti vengono inseriti in modo randomico (*Randomized Incremental Flip Algorithm*).

Prima di tutto considerano un tetraedro "ipotetico" ed "infinito" che racchiude tutti i punti di $\mathcal{S}$. Poi tutti i punti di $\mathcal{S}$ vengono aggiunti dentro l'esistente triangolazione uno alla volta. L'unico problema è che per far questo è necessario memorizzare tutti i flips fatti in un grafo detto *history dag*, con un notevole impiego di spazio in memoria.

Divide and Conquer

Il Divide and Conquer è un metodo che si applica in numerosi problemi computazionali. L'idea di base è quella di dividere i siti in due parti, approssi-

mativamente della stessa grandezza, con una linea verticale (o un piano in tre dimensioni). Questo passo è noto come *divide step* del processo intero. Ricorsivamente, fino a che non si ottengono problemi elementari, si calcola il diagramma di Voronoy delle due parti e poi la triangolazione di Delaunay dei problemi parziali e si uniscono i risultati ottenuti. Questa fase del processo prende il nome di *merging step*.

Per costruire la triangolazione di $Conv(\mathcal{S})$, il metodo si basa nel cercare un separatore capace di dividere $\mathcal{S}$ in due sottoinsiemi non connessi, $\mathcal{S}_1$ ed $\mathcal{S}_2$. Occorre poi, costruire le triangolazioni di $Conv(\mathcal{S}_1)$ e di $Conv(\mathcal{S}_2)$ che vanno poi unite per ottenere la triangolazione finale. Il problema è ora come definire un modo per ottenere una triangolazione di Delaunay di $Conv(\mathcal{S})$ unendo due triangolazioni di Delaunay che possono, come queste, essere non connesse.

Un metodo possibile, in due dimensioni, consiste nel trovare due punti "estremi" per ogni sottinsieme rispetto alla loro linea separatrice. Da entrambe le parti occorre creare una lista di punti compresi tra questi estremi e poi ordinarla, seguendo un certo criterio. Poi, per ottenere una triangolazione $Conv(\mathcal{S})$, è solo necessario congiungere i punti di $Conv(\mathcal{S}_1)$ e $Conv(\mathcal{S}_2)$ con l'ordine precedentemente scelto.

Assumiamo, per esempio, che i punti del bordo di ogni sottinsieme siano ordinati in senso orario in una lista ciclica . Si ricerca, per ogni sottinsieme, il punto più vicino alla linea di separazione. Poi si costruisce il lato che ha questi punti come estremi (a_1 per il primo insieme ed a_2 per il secondo). Consideriamo a_1 e, per vicinanza rispetto al bordo di $Conv(\mathcal{S}_2)$, si scelgono, partendo da a_2, tutti i punti di questo bordo visibili da a_1, e si costruiscono i lati corri-

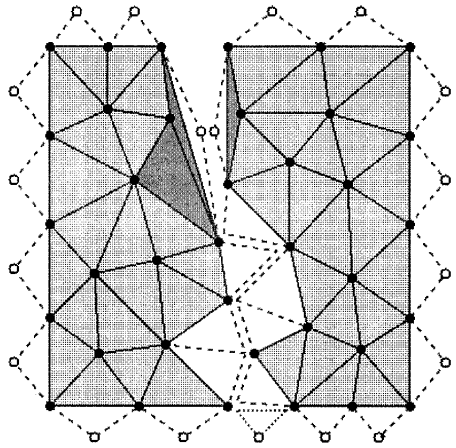

Figura 2.12: Mergin step. Le linee tratteggiate al centro sono i nuovi triangoli creati nel processo e quelli più scuri sono quelli che non sono Delaunay e verranno modificati con un flip.

spondenti. Si ripete il procedimento scandendo in ordine $\mathcal{S}_1$ partendo da a_1 ed $\mathcal{S}_2$ partendo da a_2. L'algoritmo termina quando non ci sono più punti visibili. Applicando dei flips alla triangolazione risultante, si ottiene la triangolazione di Delaunay finale (tuttavia osserviamo che un flip può portarci a dover cambiare totalmente le due triangolazioni iniziali).

In tre dimensioni il metodo è simile, l'unico problema è che costruire una triangolazione a partire dalle triangolazioni di $Conv(\mathcal{S}_1)$ e di $Conv(\mathcal{S}_2)$ diventa più complicato.

2.4 Triangolazioni constrained

Sia $\mathcal{S}$ l'insieme di punti da triangolare. Supponiamo di avere un insieme di vincoli, $Const$, cioè un insieme di lati in due dimensioni e di lati e facce in tre

dimensioni; cioè un insieme di k-facce ($1 \leq k \leq d-1$). Arricchiamo l'insieme $\mathcal{S}$ con gli estremi (estremi del lato o vertici della faccia) dei vincoli e assumiamo che i punti dell'insieme $\mathcal{S}$ così creato siano distinti. Il problema è ora di completare la triangolazione di $Conv(\mathcal{S})$ in modo che i vincoli di $Const$ siano presenti in qualche modo nella triangolazione.
Gli algoritmi di triangolazione fino ad ora discussi ci metteno in grado di costruire la triangolazione di $Conv(\mathcal{S})$. Una prima idea potrebbe essere quella di vincolare la costruzione stessa, cioè se un dato membro esiste ad un certo livello della triangolazione non dovrebbe essere più rimosso. In realtà questo non è sufficiente. Un altro metodo potrebbe essere quello di generare o rigenerare i vincoli. In realtà esistono due classi di metodi da investigare che dipendono dal modo in cui devono essere soddisfatti i vincoli. Il primo tipo attua trasformazioni locali per *rinforzare* dei dati vincoli, il secondo cerca di *modificarli* in modo da creare un insieme di vincoli ammissibili.

Alcune definizioni

Supponiamo che i membri di $Const$ non abbiano alcuna intersezione tra di loro.

Definizione 17. *Una k-faccia è **Delaunay ammissibile** se le $k+1$ celle di Voronoy associate al $k+1$-esimo vertice delle k-facce dividono una $(d-k)$-faccia.*

Possiamo quindi dire che si ha una k-faccia Delaunay ammissile ogni volta che è usato un algoritmo di triangolazione di Delaunay.

Definizione 18. *Una triangolazione $\mathcal{T}_r$* ***soddisfa esattamente*** *un insieme di vincoli $Const$ se ogni membro di $Const$ è un'entità di $\mathcal{T}_r$.*

Questo significa che se $Const$ contiene solo entità Delaunay ammissibili, allora la triangolazione di Delaunay, completata inserendo i vertici dei vincoli, conterrà esattamente $Const$.
In altre parole, la triangolazione constrained è ottenuta costruendo la triangolazione di Delaunay associata all'insieme dei vertici dei vincoli.

Definizione 19. *Una triangolazione $\mathcal{T}_r$* ***soddisfa debolmente*** *un dato insieme di vincoli $Const$ se ogni elemento di $Const$ è un elemento di $\mathcal{T}_r$ esattamente o come partizione.*

Questa definizione è meno vincolante e, da un punto di vista computazionale, più facile da soddisfare.
Generalmente si possono incontrare due tipi di problemi che dipendono da come $Const$ deve essere rappresentato in una triangolazione:

- secondo la Definizione 18, l'integrità dei vincoli deve essere preservata. In questo caso il metodo consiste, inizialmente, di modificazioni locali della triangolazione corrente in modo da rigenerare i vincoli.

- secondo la Definizione 19, l'insieme dei vincoli può essere modificato così da ottenere o un insieme Delaunay ammissibile o un opportuno insieme più facile da analizzare.

Definizione 20. *Un lato, o un triangolo, è* ***Delaunay constrained*** *se soddisfa le seguenti due condizioni.*

1. *i vertici sono visibili da ogni altro sito,*

2. *esiste una circonferenza che passa per i vertici del lato o del triangolo in questione che non contiene nessun sito di $\mathcal{S}$ visibile dall'interno del lato o del triangolo.*

Definizione 21. *$\mathcal{T}_c$ è detta **triangolazione di Delaunay constrained** di $\mathcal{S}$ se i circumcerchi aperti (circumsfere aperte) dei suoi elementi non contengono nessun punto visibile dai vertici di questi elementi.*

Osservazione: In entrambe le definizioni intendiamo che un punto P è *visibile* da un punto Q di $\mathcal{S}$ se il segmento $\overline{PQ}$ non interseca nessun punto di $\mathcal{S}$ e nessun segmento di *Const*.

Osserviamo inoltre che una triangolazione è una triangolazione di Delaunay constrained se tutti i lati o i triangoli sono Delaunay constrained.
Considereremo separatamente il caso bidimensionale e quello tridimensionale.

Il caso bidimensionale

In due dimensioni *Const* è formato da un insieme di lati. Nei metodi che andremo a considerare supporremo sempre di partire da una triangolazione del dominio che non includa gli elementi di *Const*.

Metodo della divisione dei vincoli

L'idea chiave è quella di ritriangolare ogni triangolo intersecato da un lato constrained in modo che i "sotto-lati" così creati (cioè i segmenti in cui rimane diviso il lato constrained) siano elementi della nuova triangolazione.

La prima cosa da fare è confrontare $Const$ con gli elementi della triangolazione e vedere quali lati non sono ancora inclusi nella triangolazione stessa. Poi i lati mancanti devono essere considerati uno alla volta.

Sia a il segmento, di estremi A e B, corrispondente al lato mancante. Ora

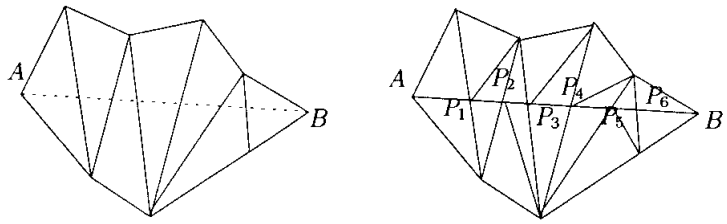

Figura 2.13: Pipe iniziale e partizione dei lati vincolati.

quello che occorre fare è:

- trovare il *pipe* associato all'elemento a, cioè l'insieme degli elementi della triangolazione che hanno almeno un lato che si interseca con a,
- trovare i punti di intersezione tra i lati interni del pipe ed a. Siano P_1, P_2,..., P_n questi punti,
- introdurre i lati $\overline{AP_1}$, $\overline{P_1P_2}$, ..., $\overline{P_nB}$ nella triangolazione,
- triangolare i quadrilateri che ora si sono venuti a formare con l'inserimento di questi nuovi lati.

Quindi quello che otteniamo è una triangolazione constrained di $\mathcal{S}$ nel senso, però della Definizione 19.

Metodo del rafforzamento dei vincoli

Prima di descrivere il metodo, facciamo alcune considerazioni sull'esistenza della soluzione.

Nel paragrafo 2.3.3 abbiamo osservato che, attraverso un numero finito di flips, è possibile ottenere da una triangolazione qualsiasi di $\mathcal{S}$ la triangolazione di Delaunay di $\mathcal{S}$. In realtà è possibile dimostrare anche l'inverso, in un certo senso, di questo risultato. Vale infatti il seguente:

Lemma 5. *Sia $\mathcal{T}_1$ un'arbitraria triangolazione dell'insieme di punti $\mathcal{S}$. Allora ogni altra triangolazione $\mathcal{T}_2$ di $\mathcal{S}$ può essere ottenuta tramite flips.*

dim: Occorre semplicemente utilizzare il risultato richiamato sopra. Allora da $\mathcal{T}_1$ è possibile ottenere la triangolazione di Delaunay $\mathcal{D}$ di $\mathcal{S}$ e poi, applicando i flips al contrario, sarà possibile ottenere $\mathcal{T}_2$.

□

Questi risultati mostrano, da una parte, che è possibile modificare, per mezzo di flips, una arbitraria triangolazione in modo da ottenerne la triangolazione di Delaunay, dall'altra parte, che è possibile modificare una arbitraria triangolazione in modo da ottenerne una completamente diversa.

Vale inoltre anche il seguente:

Teorema 3. *Esiste una triangolazione senza vertici interni che ricopre un qualsiasi dominio bidimensionale non intrecciato.*

dim: Cfr. [1].

Quindi, costruiamo questa triangolazione e vediamo come sia possibile ottene-

re, da questa, una triangolazione che includa gli elementi di *Const*, solo con degli scambi di lati.

Sia a un lato di *Const*, di estremi A e B, che manca dalla triangolazione. Costruiamo il pipe di a, $\mathcal{T}_a$, e sia $\mathcal{P}$ il poligono ottenuto dall'unione degli elementi di $\mathcal{T}_a$. Vogliamo triangolare il poligono $\mathcal{P}$ assicurando l'esistenza di a.

Per come è definito, a separa $\mathcal{P}$ in due sottopoligoni, $\mathcal{P}_1$ e $\mathcal{P}_2$.

Consideriamo, per esempio $\mathcal{P}_1$ e siano $A = P_0$, P_1, P_2,..., P_{n-1}, $P_n = B$ i

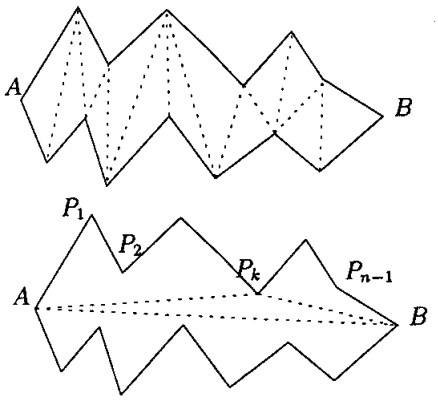

Figura 2.14: Poligono iniziale e prima ricorsione.

vertici di questo poligono. Cerchiamo tra questi P_j, $j = 1, ..., n-1$, il punto più vicino ad a. Chiamiamo questo punto P_k, nel caso che esso non sia unico possiamo scegliere secondo la numerazione del punto. Allora AP_k e BP_k dividono $\mathcal{P}_1$ in tre poligoni al massimo: uno è il triangolo ABP_k e gli altri due poligoni hanno comunque un numero di vertici minore del poligono iniziale.

Ora dobbiamo ripetere il procedimento per ogni poligono con numero di vertici maggiore di tre, ricordando che ora sono i lati AP_k e P_kB a svolgere il ruolo di a. E tutto questo va fatto iterativamente fino a che non abbiamo triangolato

$\mathcal{P}_1$. Stesso procedimento per $\mathcal{P}_2$. Quello che otteniamo è una triangolazione in cui compare $a = AB$. Si può dimostrare che questo algoritmo converge e che, applicato ad ogni lato di *Const*, implica l'esistenza della soluzione.

Metodo dell'edge swapping

Ora che abbiamo stabilito che la soluzione esiste, in accordo con i risultati richiamati sopra, sarà possibile ottenere un triangolazione constrained applicando degli opportuni flips.
Quindi quello che possiamo fare è:

- trovare il pipe associato al lato di *Const* mancante dalla triangolazione,
- *randomicamente* applicare dei flips ad ogni lato della triangolazione intersecato dal lato mancante.

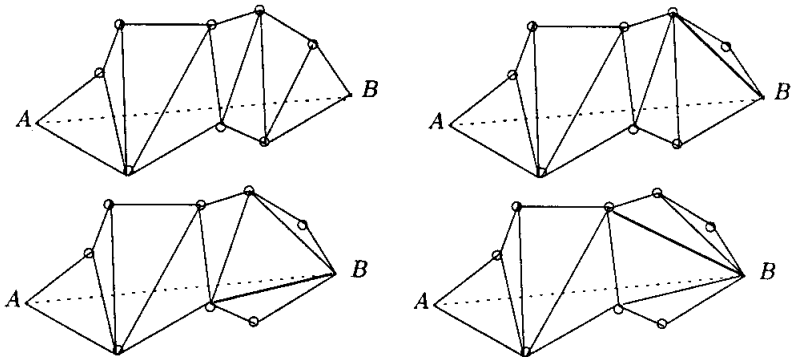

Figura 2.15: Pipe iniziale e primi tre scambi di lati.

Questo per ogni lato di *Const* mancante dalla triangolazione.
In generale, la triangolazione ottenuta in questo modo non sarà di Delaunay, a meno che non eseguiamo i flips in maniera tale da soddisfare il criterio di

Delaunay secondo la Definizione 21. Allora quella che otterremo sarà una triangolazione di Delaunay constrained.

Il caso tridimensionale

Mentre il caso bidimensionale è stato trattato e risolto con successo, non si può dire lo stesso del caso tridimensionale. Numerosi problemi sono ancora aperti ed, in particolar modo, l'esistenza della soluzione non è stata ancora chiaramente stabilita.
Certo questo non significa che non siano stati sviluppati alcuni metodi euristici per poter trovare funa soluzione adeguata ad alcuni tipi di problemi. Parecchi autori hanno dimostrato che è possibile ottenere una triangolazione di Delaunay constrained in moltissimi casi.

Metodo della divisione dei vincoli

In tre dimensioni il problema diventa dividere un segmento e dividere un triangolo.
Per quello che riguarda la divisione di un lato, quanto abbiamo detto nel caso bidimensionale può essere ricondotto con facilità al caso tridimensionale. Si cerca il pipe associato al segmento a, cioè al lato mancante. Le intersezioni di a con gli elementi della triangolazione si possono classificare nel seguente modo:

- a interseca una o due facce di un dato tetraedro,
- a interseca uno o due lati di un dato tetraedro,

- a interseca una faccia ed un lato di un tetraedro del pipe.

L'algoritmo consiste nel trovare il tipo di intersezione e nel definire i seguenti operatori:

- un operatore che divida un elemento la cui faccia è intersecata da a in tre "sotto-tetraedri",
- un operatore che tratti le intersezioni con i lati.

Applicando ricorsivamente questi operatori possiamo maneggiare tutte le possibili configurazioni.
Per quello che riguarda la divisione di una faccia, in linea di principio, il problema può essere risolto allo stesso modo. Il metodo consiste nell'inserire nella triangolazione le "sotto-facce" che otteniamo dall'intersezione delle facce vincolate con i tetraedri della corrente triangolazione.
Quindi, se quello che vogliamo è soddisfare debolmente i vincoli, allora è possibile trovare una soluzione del problema.

Metodo del rafforzamento dei vincoli

Se la nostra richiesta è quella di soddisfare i vincoli esattamente, purtroppo dobbiamo osservare che l'esistenza di una soluzione non è stata ancora provata. Cerchiamo di vedere le ragioni del limite del caso tridimensionale rispetto a quello bidimensionale.
Osserviamo per prima cosa che, mentre ogni poligono può essere triangolato, esistono dei poliedri che non possono essere triangolati senza aggiungere punti interni, detti *punti di Steiner*.

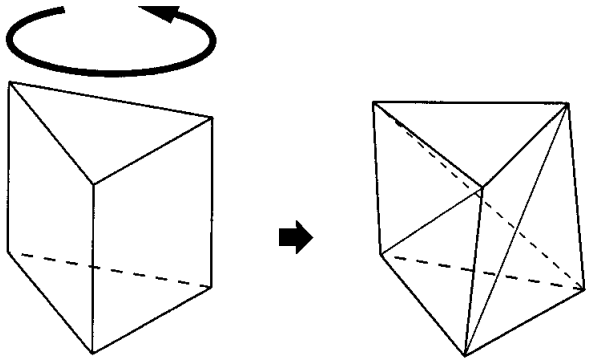

Figura 2.16: Poliedro di Schönardt.

Un esempio è il poliedro di Schönhardt in cui l'unica triangolazione contiene un tetraedro con volume esattamente zero.

Ora, il problema che sorge, è quello di scoprire i poligoni che possono generare delle simili anomalie e, in quel caso, riuscire a posizionare adeguatamente i punti di Steiner per ottenere una buona triangolazione. Quello che è ancora un problema aperto è quale sia il minimo numero di punti di Steiner richiesti per ottenere una buona triangolazione constrained.

Capitolo 3

La mesh

3.1 Introduzione

I sistemi fisici continui, come lo spostamento di aria lungo un aereo, la sollecitazione dell'acqua su una diga, il campo elettrico in un circuito integrato o la concentrazione dei reagenti in una reazione chimica, sono generalmente modellati usando equazioni alle derivate parziali. Per realizzare la simulazione al computer, i domini di questi sistemi continui devono essere discretizzati in modo da ottenere un numero finito di punti nello spazio (e nel tempo), in cui variabili, come velocità, densità e campo elettrico, possono essere calcolati.
Ci sono due tipi di mesh che si differenziano per le connessioni dei punti. Le *mesh strutturate* hanno connessioni regolari, ciò significa che ogni punto ha lo stesso numero di elementi vicini. Le *mesh non strutturate* hanno connessioni non regolari, cioè ogni punto può avere un numero diverso di elementi vicini. In alcuni casi parte della griglia può essere strutturata e parte non strutturata. Queste mesh vengono dette *ibride*.

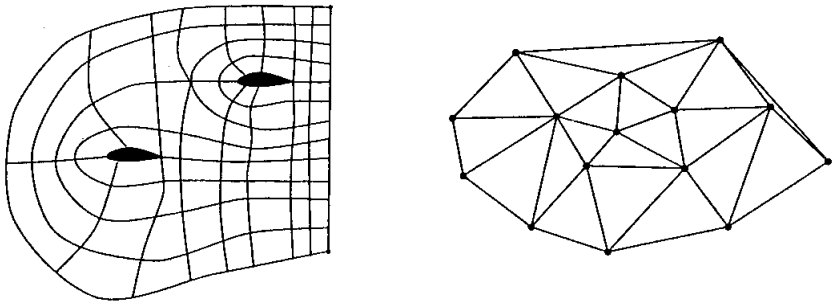

Figura 3.1: Mesh strutturata (sinistra) e mesh non strutturata (destra).

Lo scopo di questo capitolo è quello di fornire un esauriente quadro di entrambe le tecniche di generazione di mesh, strutturata e non strutturata, cercando di mostrare vantaggi e svantaggi.

3.2 Classe di metodi

Un'altra classificazione delle mesh può essere fatta in base al diverso approccio usato per la loro costruzione. Possiamo considerare cinque categorie:

1. *Metodi manuali o semi-automatici.* Sono metodi applicabili a domini geometricamente semplici. Alcuni rappresentanti di questa classe sono i metodi enumerativi, in cui gli elementi della mesh sono direttamente forniti dall'utente, e i metodi espliciti che traggono vantaggio dalla caratterizzazione geometrica del dominio.

2. *Metodi parametrici.* La mesh finale è il risultato di una trasformazione inversa della mappa di una griglia regolare di punti dallo spazio parametrico alla spazio fisico. Esistono due approcci principali che appartengono

a questa categoria che dipendendono da come è definita la mappa, cioè implicitamente o esplicitamente. E sono:

- i *metodi di interpolazione algebrica* in cui la mesh è ottenuta usando una interpolazione tra le curve di bordo (in due dimensioni) o le superfici di bordo (in tre dimensioni),
- i *metodi solution-based* in cui la mesh è generata basandosi sulla soluzione numerica di un sistema di equazioni alle derivate parziali (ellittiche, iperboliche o paraboliche) così da potersi affidare ad una soluzione analitica ben definita.

3. *Metodi di decomposizione del dominio.* La mesh finale è il risultato dell'unione delle mesh di vari domini di complessità minore in cui è stato diviso il dominio iniziale. Possiamo prendere in considerazione due approcci principali dovuti al tipo di mesh che abbiamo scelto per ricoprire il dominio, strutturata o non strutturata:

   - i *metodi di decomposizione a blocchi* in cui il dominio è decomposto in molti, ma semplici, sottodomini o *blocchi*, ognuno dei quali è ricoperto con una mesh strutturata,
   - i *metodi di decomposizione spaziale* in cui il dominio è approssimato con un'unione di celle disgiunte che ricoprono il dominio in considerazione, ogni cella è poi suddivisa negli elementi della mesh. Le tecniche di quadtree e di octree sono rappresentative di questa classe.

4. *Metodi di inserimento del punto e creazione dell'elemento.* Questi metodi partono da una discretizzazione del bordo del dominio e proseguono con

la creazione dei nodi interni del dominio e degli elementi nello stesso tempo. Il metodo *advancing front* e *Delaunay-based* sono i due diversi approcci che appartengono a questa categoria.

5. *Metodi costruttivi.* La mesh finale del dominio è il risultato dell'unione di diverse mesh usando trasformazioni topologiche o geometriche e ognuna di queste mesh è creata con uno dei metodi precedenti.

Il problema è, in generale, riuscire ad identificare il metodo adatto a creare una mesh adeguata allo scopo per cui la dobbiamo costruire. Due fattori concorrono alla scelta della mesh ideale: la geometria del problema e l'uso della mesh che dobbiamo fare.
In realtà questa classificazione non è vincolante ma, per una trattazione chiara, distingueremo solo tra mesh strutturate e non strutturate.

3.3 Generatori di Mesh Strutturate

In questo paragrafo descriveremo i metodi algebrici, i metodi basati sulla soluzione di equazioni alle derivate parziali (metodi PDE-based) e i metodi multiblock. Un metodo algebrico mira a costruire la mesh di un dominio a partire dalla mesh di un dominio più semplice come un quadrato, un quadrilatero o un triangolo, che sia legato, vedremo poi come, al dominio di partenza. Anche i metodi PDE-based mirano a costruire una mappa che porti il dominio fisico in un quadrato (cubo in tre dimensioni) per lavorare poi su quest'ultimo. Questi due metodi sono legati alla forma del dominio, cioè sono più facilmente attuabili quando il dominio ha una forma semplice. Un metodo multiblock, invece,

si basa sulla divisione del dominio in regioni semplici, le cui mesh vengono costruite con i metodi di cui sopra, e quindi ci permette di lavorare con domini particolarmente complessi. Unendo le mesh locali di tutte le regioni possiamo ottenere la mesh dell'intero dominio.

I metodi algebrici

Il primo passo per lo sviluppo di un metodo di questo tipo è quello di trovare una sequenza di mappe che portino lo spazio fisico, o dominio complesso, nello spazio logico (spazio parametrico) in modo da ottenere un dominio semplice sul quale lavorare. Poi, occorre creare una distribuzione di punti nello spazio logico, generare una mesh conforme al bordo del dominio e infine ritornare allo spazio fisico con le mappe inverse di quelle iniziali.
Le mappe e le distribuzioni dei punti di mesh possono essere scelte arbitrariamente, tuttavia, può essere utile forzare la discretizzazione del bordo dello spazio logico affinchè sia simile alla discretizzazione del bordo del dominio dato. Il controllo della distribuzione dei punti di mesh nello spazio logico ci permette di controllare la densità dei vertici di mesh nel dominio reale soprattutto per ottenere una mesh più fine nelle regioni di alta curvatura. Quello che dobbiamo fare è cercare di definire una mappa uno a uno dallo spazio logico allo spazio fisico con un metodo algebrico in cui la griglia è interpolata dal bordo dello spazio fisico.
Un metodo particolarmente usato per mappare il dominio è lo schema dell'interpolazione transfinita (TFI). Un'importante caratteristica è la sua capacità di controllare la distribuzione dei punti nella mesh ed in particolare l'inclina-

zione delle linee di mesh che intersecano le superfici di bordo. Analizzeremo ora in dettaglio lo schema TFI nel caso bidimensionale; la generalizzazione al caso tridimensionale è abbastanza semplice.
Consideriamo come dominio logico il quadrato unitario $[0,1]\times[0,1]$ con coordinate s e t, e come dominio fisico una qualche regione con coordinate x e y. Per generare la griglia nello spazio fisico dobbiamo creare una griglia nel quadrato unitario e poi trasportarla con una mappa nello spazio fisico. La mappa deve avere le seguenti caratteristiche:

- deve essere uno a uno, e
- i bordi dello spazio logico devono essere mandati dalla mappa nei bordi dello spazio fisico.

Per il resto la mappa può essere arbitraria, sebbene quello che poi ne risulta potrebbe essere una griglia non molto buona.
La TFI è un tipo di mappa in cui le coordinate fisiche, viste come funzioni delle coordinate logiche, sono interpolate a partire dai valori che hanno sul bordo del dominio logico.
L'interpolazione in due dimensioni è costruita come combinazione lineare di due interpolazioni unidimensionali, dette *proiezioni*, e dal loro prodotto. Definiamo per prima cosa le funzioni ϕ_0, ϕ_1 e θ_0, θ_1, dette funzioni *blending*, che useremo per l'interpolazione in entrambe le direzioni. ϕ_0 interpola i valori di bordo corrispondenti ad $s=0$ e ϕ_1 interpola i valori di bordo corrispondenti ad $s=1$ e similmente θ_0 e θ_1 nella direzione t. Le condizioni che $\phi_0(s)$ e $\phi_1(s)$

devono soddisfare sono:

$$\phi_0(0) = 1 \quad , \quad \phi_0(1) = 0$$
$$\phi_1(0) = 0 \quad , \quad \phi_1(1) = 1$$

e relazioni analoghe devono valere per θ_0 e θ_1 nella direzione t. La funzione blending più semplice è lineare, e ci restituisce una interpolazione lineare:

$$\begin{aligned} \phi_0(s) &= 1 - s \\ \phi_1(s) &= s \end{aligned}$$

Le proiezioni per le coordinate x sono fatte come segue:

$$\begin{aligned} P_s[x](s,t) &= \phi_0(s)x(0,t) + \phi_1(s)x(1,t) \\ P_t[x](s,t) &= \theta_0(t)x(s,0) + \theta_1(t)x(s,1) \end{aligned}$$

e il prodotto delle proiezioni è:

$$\begin{aligned} P_sP_t[x](s,t) &= \phi_0(s)\theta_0(t)x(0,0) + \phi_1(s)\theta_0(t)x(1,0) + \\ &+ \phi_0(s)\theta_1(t)x(0,1) + \phi_1(s)\theta_1(t)x(1,1) \end{aligned}$$

che è la funzione interpolante per i valori di x nei quattro vertici del dominio logico. Allora, l'interpolazione transfinita bidimensionale è data da:

$$(P_s \oplus P_t)[x] = P_s[x] + P_t[x] - P_sP_t[x]$$

che interpola tutto il bordo del dominio. Questa funzione interpolante è usata per mappare i punti di una griglia regolare dallo spazio logico allo spazio fisico così da generare la griglia in quest'ultimo spazio. E' possibile estendere lo schema TFI in modo da interpolare molte linee coordinate e non solo quelle di

bordo. Questo è utile per controllare la densità della griglia e per assicurare una mappa uno a uno, e quindi una griglia valida.
Lo schema TFI può portare a qualche problema. La mappa può "propagare" le singolarità, soprattutto dei vertici, all'interno del dominio e questo causa dei problemi soprattutto nella simulazione del movimento dei fluidi. Un altro problema è che questo metodo non è appropriato per geometrie complesse. Per domini convessi la mesh risultante è valida, ma per i domini non convessi, non sempre abbiamo questo risultato, in quanto l'immagine di un punto dentro il dominio logico potrebbe cadere fuori dal dominio reale.
Nonostante tutto, la generazione di griglie con lo schema TFI è molto veloce e si dimostra un metodo valido anche in tre dimensioni.

Metodi PDE-based

I metodi PDE-based, come generatori di mesh, rappresentano un'elegante alternativa ai metodi algebrici e possono essere usati quando il dominio, che qui chiameremo Ω, può essere identificato con un quadrato o un rettangolo, in due dimensioni, o con un cuboide, in tre dimensioni. Dato che generalmente i problemi per cui occorre una mesh sono formulati in termini di derivate parziali sembra logico usare un sistema di equazioni alle derivate parziali per trovare il sistema di coordinate che porti Ω nel dominio logico. Considereremo, ancora una volta, il caso bidimensionale.
Una mappa di una regione Ω del piano complesso è *conforme* se preserva gli angoli; in altre parole è preservato ogni angolo tra ogni coppia di curve che si intersecano in $z \in \Omega$. Il teorema della mappa di Riemann stabilisce che per

ogni dominio Ω omeomorfo ad un disco esiste una mappa uno a uno conforme, f, che manda l'interno di Ω nell'interno di un dominio omeomorfo ad un disco, per esempio lo stesso disco unitario o un quadrato unitario. Così una mappa

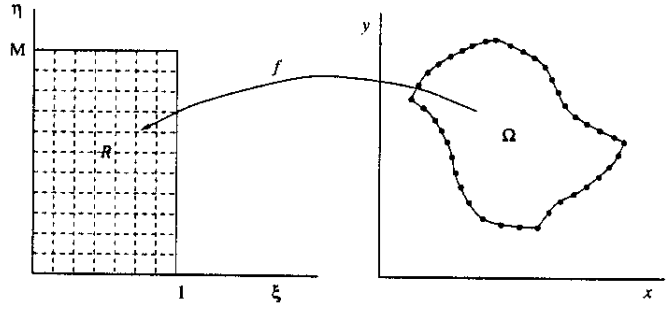

Figura 3.2: Mappa conforma f dal dominio Ω al rettangolo R.

conforme, da Ω ad un quadrato su cui è stata costruita una gliglia uniforme, induce una mesh strutturata su Ω tale che gli angoli di ogni elemento siano retti.

Il teorema della mappa di Riemann prova solo l'esistenza di questa mappa conforme ma non fornisce un algoritmo per calcolarla. Siano x e y il sistema di coordinate che descrive lo spazio reale e ξ e η il sistema riferito allo spazio logico. Scriviamo $z = x + iy$, dove i è l'unità immaginaria, e consideriamo la funzione complessa $f(z) = \xi(x, y) + i\eta(x, y)$. Se f è analitica, e questo è vero se f è conforme e $f'(z) \neq 0$, allora essa soddisfa le equazioni di Cauchy-Riemann: $\xi_x = \eta_y$ e $\xi_y = -\eta_x$. Così le funzioni ξ ed η sono armoniche e soddisfano le equazioni di Laplace: $\nabla^2\xi = 0$ e $\nabla^2\eta = 0$. Se f è conforme, anche la sua inversa lo sarà, perciò anche x e y come funzioni di ξ ed η sono armoniche e soddisfano $\nabla^2 x = 0$ e $\nabla^2 y = 0$.

Quindi scrivere (x, y) in funzione di (ξ, η) o viceversa è perfettamente equivalente e ci porta a due possibilità per risolvere il problema. Se (ξ, η) sono espresse in termini di (x, y) la mesh logica può essere trasformata in una mesh di Ω e il problema fisico può essere risolto in Ω nel modo classico. Se invece le coordinate (x, y) sono scritte in termini di (ξ, η), il problema fisico può essere scritto in termini di (ξ, η), risolto nel dominio logico e le soluzioni possono essere, poi, trasportate in Ω.

Metodi multiblock

Creare una mesh su un dominio complesso potrebbe essere complicato. Teoricamente, infatti, geometrie complesse possono essere mappate in una regione rettangolare ma questo porta ad una distorsione delle celle che non è accettabile. Un modo per affrontare il problema è quello di dividere l'intero dominio in parecchi sottodomini più semplici, *blocchi*, ognuno dei quali è ricoperto da una mesh strutturata fatta, o con un metodo algebrico, o con un metodo PDE. In pratica, in due dimensioni, il dominio è decomposto in triangoli o quadrilateri, se usiamo un metodo algebrico, o solo in quadrilateri, se usiamo un metodo PDE. In tre dimensioni il dominio viene diviso in regioni tetraedrali, pentaedrali o esaedrali, se usiamo un metodo algebrico, e solo regioni esaedrali, se usiamo un metodo PDE. Questi blocchi sono poi uniti insieme a creare un *multiblock*, e adattati in modo da avere un certo grado di continuità nei punti di interfaccia. Il grado di continuità sarà nullo se le coordinate dei punti della griglia sono diverse e sarà massimo se non c'è alcuna discontinuità nei punti

di griglia.

Così, il processo di generazione della griglia si divide in due parti: la decomposizione del dominio fisico in blocchi e la grigliatura di ogni blocco. Il processo di decomposizione, però, non è del tutto automatico e richiede un considerevole intervento dell'utente per la scelta dei blocchi se si vuole produrre una buona mesh.

In base alle richieste di continuità e di vincoli tra i blocchi, è possibile fare una classificazione dei metodi multiblock.

- **Overlapping.** Se non abbiamo particolari richieste per le interfacce dei blocchi, ogni blocco può essere grigliato separatamente. Quello che ne risulta è un sistema di "sotto-mesh" che si sovrappongono. Sebbene la mesh sia facile da generare, lo svantaggio principale di questa tecnica è che, se usiamo questa griglia per risolvere un metodo numerico, questo potrebbe non convergere.

- **Patched.** Se alla tecnica overlapping multiblock aggiungiamo come vincolo che le mesh di separazione siamo conformi al loro bordo comune, anche se le linee di mesh non sono continue, otteniamo il metodo patched multiblock. Con questo approccio l'interpolazione è più semplice e il raffinamento della mesh può essere fatto localmente senza doversi allargare ai blocchi limitrofi.

- **Composite.** Se richiediamo che le linee di mesh siano continue lungo il bordo otteniamo il metodo composite multiblock. Questa tecnica richiede

una numerazione globale dei vertici, ma il suo principale vantaggio è l'accuratezza che ne risulta.

Consideriamo ora l'aspetto geometrico. Innanzi tutto la geometria del dominio deve essere ben approssimabile dai blocchi e la forma dei blocchi deve essere la più vicina possibile ad una regione convessa in modo che l'applicazione locale del processo di mesh termini con successo.
Una scelta corretta del numero dei blocchi ci permette di ottenere una buona approssimazione della geometria; nelle regioni ad alta curvatura, ad esempio, occorrono un gran numero di blocchi. D'altra parte avere a che fare con un numero troppo grande di blocchi può creare dei problemi. Infatti diventano troppe le condizioni di raccordo sulle interfacce dei blocchi stessi, cioè le relazioni che servono ad assicurarci un buon grado di continuità.
Ora vediamo cosa dobbiamo fare dal punto di vista computazionale. Ricordiamo che ogni blocco è descritto dai suoi vertici, dai suoi lati e dalle sue facce, in tre dimensioni. L'idea è quella di esaminare tutti i vertici, tutti i lati e tutte le facce prima di fare la mesh di ogni blocco. Allora i passi da fare sono i seguenti:

1. *Definizione dei vertici.* In questo passo occorre solo fare una numerazione globale dei vertici.

2. *Definizione dei lati e mesh.* Ogni lato è definito dai suoi estremi, che sono vertici, e dal parametro di suddivisione n. Così ogni lato è diviso in $n+1$ segmenti, cioè lungo questo segmento sono creati n punti interni. Come prima dovremo definire una numerazione globale dei nuovi vertici creati

3. *Definizione delle facce e mesh.* Le facce sono definite dai lati che le delimitano. A seconda del tipo di faccia (triangolare o quadrilaterale) e a seconda del parametro di suddivisione. La faccia è grigliata usando, per esempio, un metodo algebrico. Come prima dovremo fare una numerazione globale dei vertici creati.

4. *Definizione dei blocchi e mesh.* Ogni blocco è definito dalle facce che lo delimitano. Il numero dei vertici interni può essere creato in modo sequenziale a partire dal primo vertice disponibile aggiungendo 1.

5. *Costruzione della mesh globale.* Questo passo può essere fatto automaticamente a partire dalla numerazione globale fatta nei passi precedenti.

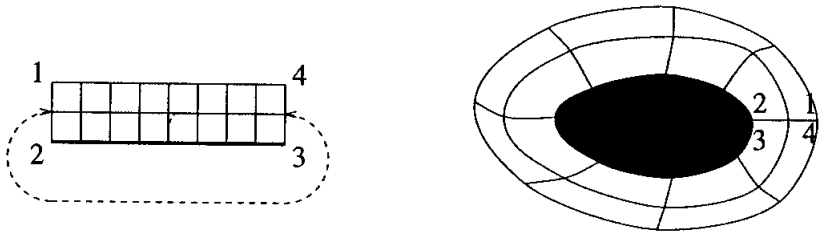

Figura 3.3: Griglia di tipo O

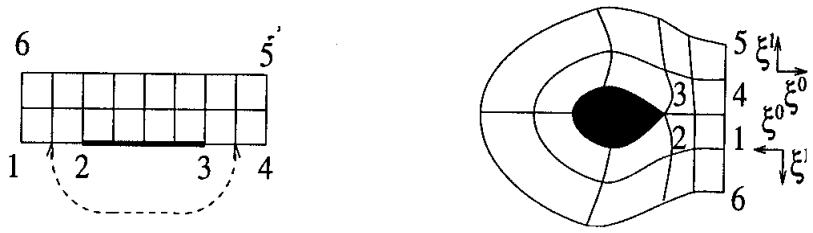

Figura 3.4: Griglia di tipo C

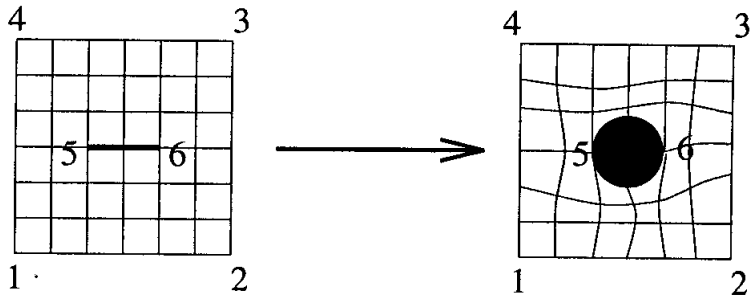

Figura 3.5: Griglia di tipo H

Fino ad ora abbiamo analizzato blocchi di forma rettangolare, in realtà esiste un'estensione del semplice blocco rettangolare che può essere usata per definire più adeguatamente alcuni tipi di domini che poi possono essere inseriti in uno schema multiblock come un singolo blocco. Un dominio può essere discretizzato secondo un'analisi di tipo O, C o H, come vediamo dalle figure 3.3, 3.4 e 3.5. Ci sono alcuni svantaggi nell'utilizzo di un metodo multiblock per la generazione di una mesh:

- La divisione in blocchi richiede un gran lavoro da parte dell'utente, perchè ogni dominio è diverso, occorre quindi una diversa configurazione dei blocchi stessi.

- Cambiare la geometria di un blocco richiede di dover cambiare la geometria di molti altri blocchi.

- Cambiare la distribuzione dei punti di griglia in un blocco porta a dei cambiamenti nella distribuzione dei punti nei blocchi vicini per mantenere il grado di continuità.

Dato il grande lavoro che richiede un metodo multiblock, per la maggior parte dei domini complessi è preferibile usare delle griglie non strutturate che vengono generate automaticamente e non richiedono affatto l'intervento dell'utente.

3.4 Generatori di Mesh non Strutturate

Il metodo Advancing Front

Gli studi sulla tecnica advancing front cominciarono circa trenta anni fa con l'analisi del caso bidimensionale. Continuarono poi allargandosi al caso tridimensionale, soprattutto negli ultimi sette anni. Il metodo, come lo descriveremo ora, è molto potente e permette la generazione di mesh non strutturate di alta qualità composte da simplessi (triangoli e tetraedri) per domini di forma arbitraria. La variante di questo metodo, proposta da Blacker e Meyers nel 1993, usa quadrileteri ed esaedri al posto di simplessi.
Il metodo advancing front è una procedura iterativa che parte da una discretizzazione iniziale del bordo, lati in due dimensioni e facce triangolari in tre. Questa discretizzazione iniziale è il *fronte*. Ad ogni passo si seleziona un'entità del fronte, viene creato un nuovo vertice, detto *ottimo*, e si inserisce il nuovo elemento così creato nella mesh. Il metodo può essere schematizzato come segue:

1. Definizioni preliminari:

   - input del bordo del dominio e mesh $\mathcal{T}$ composta dai soli elementi di bordo, detta *mesh di background* ,

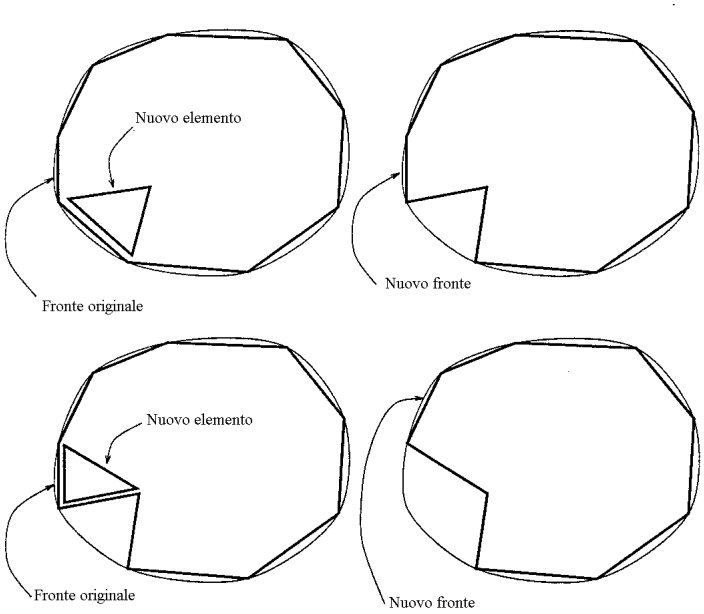

Figura 3.6: Il metodo Advancing Front

- specifica di una *funzione di distribuzione della grandezza degli elementi* che governa la generazione della mesh.

2. Inizializzazione del fronte $\mathcal{F}$ con $\mathcal{T}$

3. Analisi del fronte $\mathcal{F}$ (fino a che $\mathcal{F}$ non è vuoto):

   (a) selezionare un'entità del fronte f,

   (b) determinare il punto ottimo, P_{opt} per la nuova entità,

   (c) determinare se esiste già un punto P nella mesh che può essere usato al posto di P_{opt}. Se questo punto esiste $P = P_{opt}$,

   (d) formare un nuovo elemento K con f e P_{opt},

(e) controllare se K interseca qualche elemento della mesh. Se questo accade, trovare un nuovo punto P e tornare al punto (d).

4. Aggiornare il fronte e la mesh corrente:
   - rimuovere dal fronte $\mathcal{F}$, f ed ogni entità di $\mathcal{F}$ usata per formare K,
   - aggiungere le entità appena create di K nel fronte,
   - aggiornare la mesh corrente $\mathcal{T}$.
5. Se il fronte $\mathcal{F}$ non è vuoto ritornare al passo (3).
6. Ottimizzazione della mesh (opzionale).

Ora daremo alcuni dettagli sulle varie fasi del metodo in tre dimensioni.
Innanzi tutto, la mesh di background può essere ottenuta ad esempio con un metodo Delaunay-based o con un metodo octree. Nel primo caso non avremo vertici interni a parte la possibilità dell'esistenza di alcuni punti di Steiner in tre dimensioni, nell'altro caso invece ci saranno tutti i punti generati dalla griglia octree.
Uno dei maggiori problemi, che si possono incontrare quando si usa un metodo advancing front, è dato dal modo in cui convergono due fronti opposti. Infatti, se due fronti opposti hanno elementi di grandezza molto diversa, gli elementi che devono essere creati per farli unire saranno mal formati. E' quindi indispensabile introdurre una funzione che controlli la grandezza degli elementi, la così detta *funzione di controllo dello spazio*. Questa funzione associa ad ogni punto del dominio un valore h che è la lunghezza richiesta per ogni lato che parte da questo punto. Per costruire questa funzione dobbiamo usare le

informazioni che ci sono fornite dalla mesh di background. Il valore h_P, cioè h su ogni punto P del dominio, va calcolato per interpolazione lineare sugli h_{P_i}, $i = 1, ..., d$ (Fig. 3.7) dove d è la dimensione dello spazio e P_i sono i vertici dell'elemento della mesh che contiene P.

I valori di h sui punti di bordo possono essere trovati calcolando la media

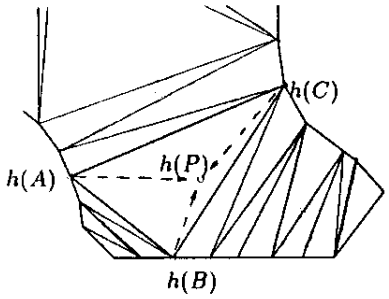

Figura 3.7: Costruzione della funzione di controllo dello spazio.

delle lunghezze dei segmenti che partono da ogni punto di bordo [10]. Un altro metodo è il seguente [17]. Sia P un vertice della mesh di background,

- se P è un vertice di bordo h_P può essere definito come segue:

$$h_P = \frac{l_{min}(P) + l_{max}(P)}{2}$$

  dove $l_{min}(P)$ (rispettivamente $l_{max}(P)$) rappresenta la lunghezza euclidea del lato di bordo più corto (risp. più lungo) connesso con P,

- se P è un punto di Steiner, allora

$$h_P = \frac{\sum_a \frac{1}{l(a)} h_a}{\sum_a \frac{1}{l(a)}}$$

  dove a rappresenta l'insieme dei lati che hanno P come come estremo e

h_a è il valore della funzione di controllo nel vertice opposto a P lungo il lato a.

Per analizzare i passi successivi è necessario introdurre alcuni concetti. Sia PQ un lato della mesh, h_P e h_Q i valori della funzione di controllo nei vertici P e Q e sia $h(t)$ la funzione monotona data dalla variazione di $h(t)$ lungo PQ in modo che $h(0) = h_P$ e $h(1) = h_Q$. La *lunghezza normalizzata del lato*, l_{PQ}, del lato PQ, rispetto ad $h(t)$, è definita come:

$$l_{PQ} = d_{PQ} \int_0^1 \frac{1}{h(t)} dt$$

dove d_{PQ} è l'usuale distanza euclidea tra P e Q.
Per esempio, la funzione più semplice che possiamo usare per $h(t)$ è la funzione ottenuta per interpolazione aritmetica: $h(t) = h_P + t(h_Q - h_P)$. Allora la lunghezza del lato normalizzata diventa:

$$l_{PQ} = d_{PQ} \frac{\log(\frac{h_Q}{h_P})}{h_Q - h_P}$$

.

Dopo aver completato la generazione della funzione di controllo dello spazio, possiamo partire con il processo di generazione della mesh vero e proprio. Il primo problema che incontriamo è quello di scegliere la faccia di base con cui iniziare il processo di generazione della mesh. Un metodo advancing front valido deve essere capace di generare una mesh, qualunque sia la scelta fatta a questo livello. Tuttavia, soprattutto in tre dimensioni, è importante tener conto di queste cose:

- è meglio minimizzare la grandezza del fronte in termini di numero di

punti o di facce in modo che le operazioni di ricerca e la validità e la qualità dei controlli siano ridotti,

- è preferibile rifinire il fronte in modo che il numero degli angoli diedri acuti sia ridotto.

E' importante, quindi, il genere di discretizzazione del bordo da cui partiamo. Infatti esiste una stretta relazione tra la qualità del triangolo di base, Q_f, di un tetraedro e la qualità del tetraedro stesso, Q_K. Seveno, in [16], ci fornisce una relazione empirica tra questi due valori ma che approssima molto bene la qualità di un tetraedro:

$$(Q_K)^{-1} = \frac{\sqrt{2}}{2}(Q_f)^{-1} + 1 + \frac{\sqrt{2}}{2}.$$

Quindi esiste una stretta relazione tra la qualità di una mesh di volume $Q_{\mathcal{T}}$ e la qualità della mesh di superficie.
Un altro problema è quello della determinazione del punto ottimo, e questa è la parte più importante del metodo. A partire dalla prima faccia scelta per formare il nuovo tetraedro, chiamiamola f, dobbiamo ricercare quel punto, all'interno del dominio, tale che:

- il nuovo tetraedro sia di buona qualità,
- la lunghezza dei lati così formati rispettino la funzione di controllo.

La lunghezza normalizzata del lato ottimo è 1, cioè il tetraedro K definito da f e da P è ottimo se e solo se:

$$\forall P_i \; vertice \text{ di f } \; l_{PP_i} = 1.$$

In pratica, la posizione di un punto ottimo è calcolata per iterazione.

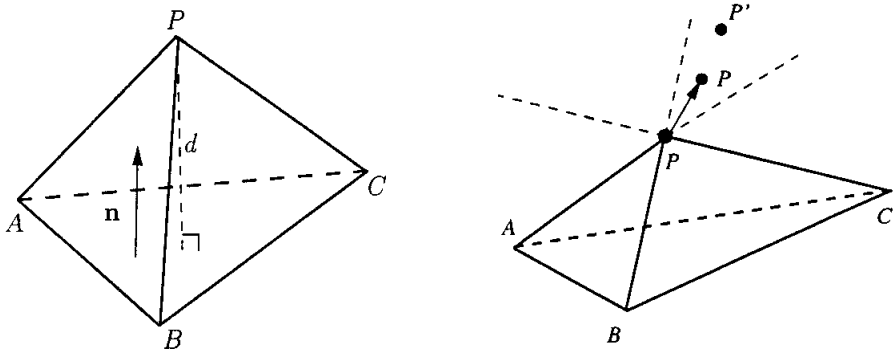

Figura 3.8: Posizionamento del punto ottimo (sinistra) e correzione del punto ottimo (destra).

1. Il vertice P è creato avanzando nella direzione normale ad f ad una distanza calcolata come la media aritmetica dei valori di h sui vertici della faccia per un fattore di normalizzazione in modo che una faccia equilatera debba portare ad un tetraedro regolare se h è costante.

2. **Repeat**:

   (a) Calcolare h_P dalla funzione di controllo dello spazio.

   (b) Calcolare le lunghezze normalizzate l_{AP}, l_{BP} e l_{CP}.

   (c) Scegliere i punti A', B', C' in modo che $l_{AA'} \approx 1$, $l_{AB'} \approx 1$, $l_{AC'} \approx 1$.

   (d) Calcolare G', il baricentro del triangolo $A'B'C'$.

   (e) P è spostato su G' tramite un coefficiente di rilassamento ω,
   $P = P + \omega\overrightarrow{PG'}$

   until le lunghezze sono vicine all'unità o le iterazioni hanno raggiunto un valore massimo.

Nella pratica $\omega = 0.1$ e servono al massimo cinque iterate per la convergenza verso il punto ottimo.

Ora dobbiamo scegliere ed ordinare i punti che sono i candidati per formare il nuovo elemento della mesh a partire dalla faccia di base f. E' importante sapere come questo elemento può essere connesso al fronte. Ci sono delle indicazioni teoriche e sperimentali da seguire per realizzare al meglio questo passo.

Siano n_{bf}, n_{if}, n_t il numero delle facce di bordo, delle facce interne e il numero dei tetraedri nella mesh finale. Allora si ha:

$$\frac{n_{if}}{n_t} = 2 - \frac{n_{bf}}{n_t}.$$

In media, per costruire un nuovo tetraedro vengono create meno di due facce. Dato che il tetraedro è costruito da una faccia già esistente, più della metà degli elementi della mesh può essere generato connettendo una faccia di base ad una faccia adiacente (che ha in comune un lato). La tabella 3.1 mostra la frequenza delle possibili connessioni tra il tetraedro che dobbiamo costruire e il fronte corrente (eccetto la faccia di base) durante il processo di generazione della mesh. I valori sono delle medie calcolate su molti esempi con un dominio molto grande. Il tipo di connessione è scelto in modo da ottenere il tetraedro migliore [16]. Con cp, ce e cf vengono indicati, rispettivamente, il numero dei punti, dei lati e delle facce create.

Osserviamo che la frequenza di una connessione con un punto adiacente è maggiore dell'80% (Casi 6, 7, 8 e 9) e che le connessioni con altri punti di bordo sono marginali (Casi 2, 3, 4 e 5). Quindi l'ordine con cui dobbiamo cercare i punti che sono candidati a formare il nuovo tetraedro è:

Caso	**Tipo di connessione**	%	cp	ce	cf
1	Nessuna connessione (Nuovo punto)	13.50	1	3	3
2	Connessione con 1 punto	0.08	0	3	3
3	Connessione con 1 lato	3.09	0	2	3
4	Connessione on due lati	0.10	0	1	3
5	Connessione con tre lati	0.00	0	0	3
6	Connessione con 1 faccia	59.20	0	1	2
7	Connessione con 1 faccia e 1 lato	0.18	0	0	2
8	Connessione con 2 facce	23.70	0	0	1
9	Connessione con tre facce	0.22	0	0	0

Tabella 3.1: Connessione di un tetraedro con il fronte.

1. il più adatto tra i punti adiacenti,
2. il punto ottimo definito come sopra,
3. il punto di bordo più adatto.

Ora ci sono dei controlli da fare per verificare la conformità e la validità di ogni elemento della mesh. Ricordiamo che un tetraedro è *valido* se:

1. ha volume strettamente positivo,
2. non interseca il fronte,
3. non contiene nessun altro punto della mesh.

Osserviamo che la condizione 2 è verificata non appena i lati creati non intersecano nessuna faccia del fronte e le facce create non intersecano nessun

elemento del fronte. Mentre la condizione 3 è necessaria per scoprire i seguenti casi:

- se l'elemento formato contiene tutto un elemento con una faccia in comune,

- se l'elemento così creato contiene un punto interno che non è connesso con nessun lato o faccia.

Ovviamente, per decidere se accettare o meno un elemento, è anche necessario fare un test sulla sua qualità (Cfr. Capitolo 1). Osserviamo che è sempre possibile costruire un tetraedro valido a partire da una faccia di base anche se a volte è necessario sacrificare la qualità della mesh. E' quindi spesso necessario un'ottimizzazione della mesh con i criteri che vedremo in seguito.
Anche se quello che abbiamo fino ad ora illustrato è un algoritmo molto valido, in tre dimensioni, un problema è come generare la mesh di superficie con cui inizializzare il fronte. Spesso allora viene usato un algoritmo di tipo Delaunay-based per generare la mesh di superficie.
Il pregio degli algoritmi di tipo advancing front è quello di creare degli ottimi tetraedri nelle vicinanze del bordo ma, se la funzione di controllo della spaziatura non è scelta opportunamente o non è buona la mesh di superficie iniziale, all'interno possono essere generati dei tetraedri malformati. Alcuni ovviano a questo problema usando un algoritmo di tipo advancing front sul bordo e di tipo Delaunay all'interno.
Algoritmi "misti" di tipo Delaunay-Advancing front sono stati ampiamente trattati, in particolare da Frey, Borouchaki e George ([17]) e da Fleischmann e Selberherr ([18]).

I metodi Delaunay-based

Come abbiamo già accennato nel Capitolo 2, costruire una mesh di un dominio con un metodo Delaunay è equivalente a costruire una triangolazione Delaunay constrained di un insieme di punti, opportunamente inseriti, all'interno del dominio, avendo come lati o facce vincolate gli elementi della discretizzazione del bordo del dominio stesso. Quindi, una volta che abbiamo costruito i punti all'interno del dominio, possiamo utilizzare i metodi che abbiamo visto nel Capitolo 2. In questo paragrafo tratteremo, quindi, solo i vari possibili algoritmi per l'inserimento dei punti.

Un modo per generare i punti prima del processo di mesh vero e proprio è quello di sovrapporre al dominio una griglia strutturata, con lati paralleli agli assi coordinati ad esempio, e di prendere come vertici di mesh sia i punti di intersezione delle rette della griglia tra di loro che i punti di intersezione delle rette con il dominio (Fig. 3.9). Se usiamo questo metodo, avremo bisogno

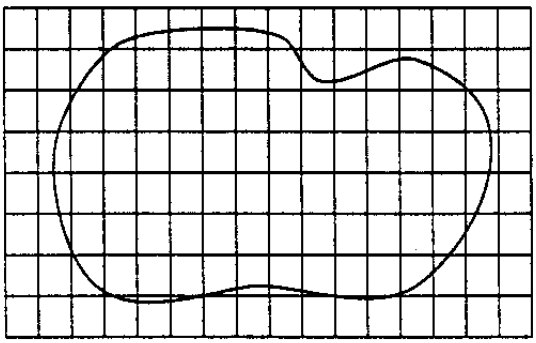

Figura 3.9: Griglia uniforme sovrapposta ad un dominio.

anche del processo di ottimizzazione soprattutto per migliorare la qualità della

mesh sul bordo.

Un metodo molto usato per inserire i punti è quello di generarli simultaneamente alla triangolazione, scegliendo il punto in modo da migliorare la qualità degli elementi. In due dimensioni l'algoritmo procede come segue:

1. Si forma una triangolazione iniziale di $\mathcal{B}(\Omega)$.
2. si inseriscono i punti di bordo nella triangolazione usando l'algoritmo di Boyer-Watson.
3. Si ordinano i triangoli secondo la loro qualità, dal peggiore al migliore.
4. Si prende il primo triangolo della lista e si inserisce un nuovo punto nel circumcentro di quel triangolo. In due dimensioni questo permette di ottenere angoli compresi tra 30° e 120°, cosa che non è garantita in tre dimensioni.
5. Ritriangolare usando l'algoritmo di Boyer-Watson.
6. Inserire i nuovi triangoli nella lista di qualità se non sono ancora sufficientemente buoni.

L'algoritmo termina o quando tutti i triangoli sono di buona qualità o quando abbiamo inserito un numero massimo di punti ([15]).

Un altro possibile metodo è quello proposto in [19]. Partiamo ancora una volta da una discretizzazione del dominio e dalla mesh creata con i soli punti di bordo del dominio o al massimo con pochi punti interni (i punti di Steiner in tre dimensioni). Questo metodo usa i lati della corrente mesh come supporto spaziale. Inizialmente associamo ad ogni vertice di bordo un passo h trovato

come media delle lunghezze (aree delle superfici) dei lati (delle facce) che hano in comune lo stesso vertice di bordo. L'idea di base è quella di considerare i lati della corrente mesh e di costruire un insieme di punti su di loro. Il processo è ripetuto fino a che viene richiesta la creazione di un nuovo punto sul lato, cioè fino a che il lato non è *saturo*.

I lati della mesh corrente vengono esaminati uno ad uno e vengono anche confrontate le loro lunghezze con i valori di h ai loro estremi. Lo scopo è quello di determinare se e quanti punti possono essere creati sul lato e dove devono essere collocati. Vediamo un tipo di distribuzione aritmetica per un lato AB di lunghezza d. Sia $h(0) = h_A$ il passo associato a $P_0 = A$ e $h_{n+1} = h_B$ il passo di $P_{n+1} = B$. Definiamo la sequenza α_i (e così il punto corrispondente P_i) come:

$$\begin{cases} \alpha_0 &= h(0) + r \\ \alpha_n &= h(n+1) - r \\ \alpha_i &= d(P_i, P_{i+1}) \end{cases}$$

dove $d(P_i, P_{i+1})$ è la distanza euclidea tra P_i e P_{i+1} mentre r è il rapporto di distribuzione, cioè il passo della discretizzazione. Questo ci serve per risolvere il sistema:

$$\begin{cases} \sum_{i=0}^{n} \alpha_i &= d \\ \alpha_{i+1} &= \alpha_i + r \end{cases}$$

quindi per trovare r ed n. Le soluzioni del sistema sono:

$$n = \frac{2d}{h(0) + h(n+1)} - 1 \quad \text{e} \quad r = \frac{h(n+1) - h(0)}{n+2}.$$

Poichè n deve essere un intero, la soluzione deve essere convertita in un numero intero. Ora che abbiamo calcolato r, ci è nota la posizione di ogni punto P_i, e

per ognuno di essi dobbiamo calcolare h. Il processo è ripetuto fino a che non si raggiunge la saturazione.
Questo metodo è efficiente sia in due che in tre dimensioni e produce, in media, una buona locazione dei punti anche se è necessaria, dopo la generazione della mesh, una fase di ottimizzazione.

3.5 Quadtree e Octree

I metodi di decomposizione spaziale sono stati inizialmente proposti nel 1975 da Knuth come un modo per rappresentare geometricamente dei domini complessi. I metodi octree (in tre dimensioni) e quadtree (in due dimensioni), che appartengono a questa classe, hanno avuto il loro sviluppo soprattutto nelle ultime due decadi e sono stati usati con successo per generare in modo automatico le mesh.
Se supponiamo di aver già discretizzato il bordo del dominio Ω, il metodo di generazione quadtree parte racchiudendo l'intero dominio dentro un quadrato, detto *root*, allineato con gli assi. Si divide questo quadrato in quattro quadrati congruenti e si continua dividendoli ricorsivamente. Ogni quadrato minimo, detto *leaf* conterrà almeno un punto del bordo di Ω. Queste divisioni possono anche essere regolate dall'utente inserendo una funzione spaziatura o una *condizione di bilanciamento*.
Ad esempio, l'algoritmo di Bern, Eppstein e Gilbert ([13]) prevede la seguente condizione di bilanciamento: nessun quadrato deve essere adiacente ad un altro quadrato che abbia dimensione minore di questo di una volta e mezzo. Questo causa molte divisioni per allargare il quadtree quindi aumenta il numero totale

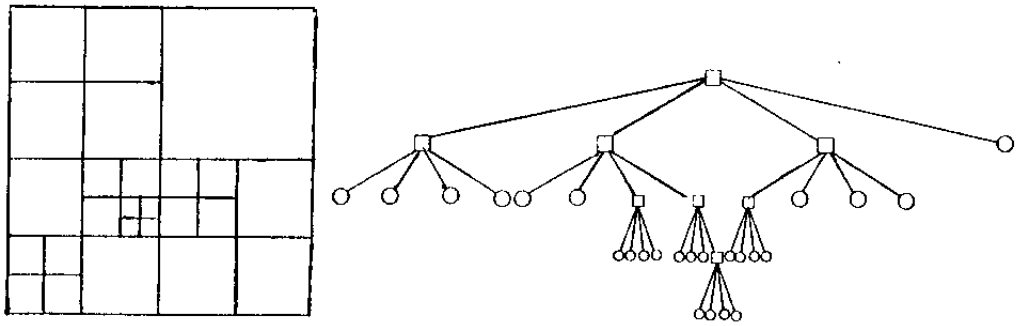

Figura 3.10: Decomposizione quadtree e corrispondente struttura dati (qui la profondità dell'albero è $p = 4$).

dei quadrati leaf di un fattore costante (al più 8). I quadrati devono essere poi *warped*, cioè "deformati", in qualche modo, per adattarsi al dominio Ω. Ci sono vari metodi per questo, ma qui di seguito riportiamo un metodo possibile, quello usato da Bern, Eppstein e Gilbert. Nel seguente pseudocodice, $|b|$ rappresenta la lunghezza del lato di b.

for ogni vertice v di Ω **do**

sia y il vertice del quadtree più vicino

spostare y su v

endfor

for ogni quadrato leaf b ancora attraversato da un lato e **do**

spostare i vertici di b che sono più vicini di $\frac{|b|}{4}$ da e dei loro punti più vicini su e.

endfor

scartare le facce dei quadtree warped che cadono fuori da Ω.

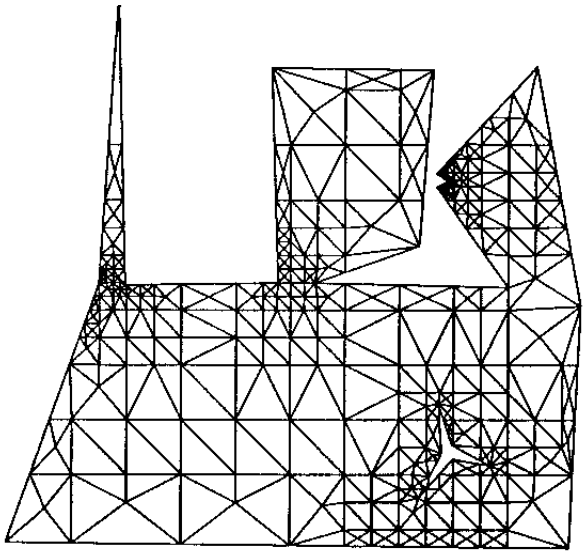

Figura 3.11: Mesh calcolata con un metodo quadtree.

Alla fine le celle dei quadtree warped sono triangolati in modo che tutti gli angoli non siano troppo piccoli .

Possiamo schematizzare il metodo quadtree come segue:

1. Definizioni preliminari:
   - costruzione di una funzione di distribuzione,
   - discretizzazione del bordo,
   - creazione di $\mathcal{B}(\Omega)$,
2. inizializzazione dell'albero con $\mathcal{B}(\Omega)$,
3. costruzione dell'albero:
   (a) selezione delle entità del bordo in ordine ascendente (punti, lati e facce),

(b) identificazione delle celle nel corrente albero che contengono queste entità,

(c) analisi della cella, e se questa contiene già un'entità della stessa dimensione, si raffina la cella, altrimenti si torna al punto (b),

(d) inserimento dell'entità nella cella e si ritorna al punto (a),

4. bilanciamento dell'albero,

5. creazione dei punti interni e degli elementi di mesh,

6. eventuale ottimizzazione.

Osserviamo che un metodo quadtree ha delle direzioni privilegiate di mesh, che sono quelle orizzontali e verticali (Figura 3.11).
Un octree è la naturale generalizzazione di un quadtree. Un cubo iniziale che limita il dominio è diviso in otto cubi congruenti ognuno dei quali è diviso ricorsivamente fino a che ogni cubo interseca il dominio in almeno un punto. Come in due dimensioni, una condizione di bilanciamento assicura che nessun cubo sia contiguo ad un altro che sia molto più piccolo del precedente.
Concludendo, possiamo dire che un metodo basato sulla decomposizione ad albero del dominio è valida, efficiente e flessibile anche se particolarmente dipendiosa dal punto di vista computazionale.

3.6 Ottimizzazione

L'ottimizzazione di una mesh rispetto ad un dato criterio è un'operazione frequentemente usata con vari scopi ognuno corrispondente ai vari tipi di ap-

plicazioni. Prima di tutto, l'ottimizzazione in sè è utile perchè la soluzione di un metodo numerico calcolato sui punti di mesh dipende dalla qualità della mesh.
Ci sono vari modi per attuare un'ottimizzazione di una mesh. I più usati sono:

- riposizionamento dei vertici,
- edge collapsing per rimuovere un vertice,
- flip di lati,
- flip di facce,
- rilassamento dei vertici, che può essere fatto combinando nel modo opportuno i tre metodi precedenti,
- divisione di un lato, per aggiungere un vertice, e così via...

In realtà, i metodi di ottimizzazione di mesh possono essere classificati in due categorie: quelli che mantengono le connettività e che quindi ricollocano i punti, e quelli che preservano la posizione dei vertici ma non le loro connettività.

Ottimizzazione che mantiene le connettività

Questo processo di ricollocamento dei vertici è legato a quello che nel Capitolo 2 avevamo chiamato $\mathcal{B}_P$, cioè l'insieme degli elementi della mesh $\mathcal{M}$ che hanno P come vertice. $\mathcal{B}_P$ sarà un insieme chiuso se P è un punto interno della mesh o aperto se il punto è di bordo.

Il metodo più semplice di ricollocamento dei nodi può essere sritto come:

$$P' = \frac{1}{n}\sum_{j=1}^{n} P_j \tag{3.1}$$

dove P_j sono gli altri vertici di $\mathcal{B}_P$.

Una prima variazione consiste nell'aggiungere dei determinati "pesi" associati ad ogni punto P_j, ottenendo:

$$P' = \frac{\sum_{j=1}^{n} \alpha_j P_j}{\sum_{j=1}^{n} \alpha_j}. \tag{3.2}$$

Possiamo anche usare il rilassamento di questo metodo. A questo proposito dobbiamo introdurre un punto ausiliario P^*. Come nella relazione 3.1

$$P^* = \frac{1}{n}\sum_{j=1}^{n} P_j, \tag{3.3}$$

e lo schema rilassato sarà definito come:

$$P = (1-\omega)P + \omega P^*, \tag{3.4}$$

con ω vicino ad 1 in due dimensioni e molto piccolo, invece, in tre dimensioni. Basandoci sullo schema rilassato delle relazioni 3.3 ed 3.4, detto **smoothing laplaciano**, possiamo definire alcuni metodi di ricollocamento dei punti.

Smoothing pesato: In questo caso, ad ogni punto è associato un peso come in 3.2, che allora diventa:

$$P^* = \frac{\sum_{j=1}^{n} \alpha_j P_j}{\sum_{j=1}^{n} \alpha_j}. \tag{3.5}$$

Smoothing basato sulla qualità degli elementi: Se partiamo da una mesh fatta di simplessi, possiamo considerare il punto P e la boccia $\mathcal{B}_P$ associata a questo punto e supponiamo che sia composta da n elementi . Sia f_j un lato esterno (una faccia esterna in tre dimensioni) di $\mathcal{B}_P$. Allora gli elementi di $\mathcal{B}_P$

non sono altro che le combinazioni di (P, f_j) per $j = 1, ...n$. Supponiamo di associare ad ogni f_j un punto ideale I_j che assicuri un'ottima qualità all'elemento virtuale (I_j, f_j). Utilizzando questi punti possiamo definire il processo di smoothing come segue:

$$P^* = \frac{\sum_{j=1}^{n} \alpha_j I_j}{\sum_{j=1}^{n} \alpha_j}, \tag{3.6}$$

dove gli α_j possono essere definiti come segue:

- $\alpha_j = 1$, i pesi sono costanti e si ritorna al metodo classico,
- $\alpha_j = 0$, per ogni elemento di $\mathcal{B}_P$ eccetto che per il peggiore di essi in termini di qualità per il quale scegliamo $\alpha_j = 1$,
- $\alpha_j = Q_{K_j}$, i pesi sono legati alla qualità degli elementi,
- $\alpha_j = Q_{K_j}^2$, i pesi sono legati al quadrato della qualità degli elementi,
- o, più generalmente, $\alpha_j = g(Q_{K_j})$, cioè i pesi sono legati ad una funzione che dipende dalla qualità degli elementi di $\mathcal{B}_P$.

In entrambi i casi, applichiamo la relazione 3.4 per rilassare il metodo.

Ottimizzazione che mantiene la posizione dei vertici

In due dimensioni, il metodo più semplice è quello di fare un **flip di un lato** in due dimensioni o il **flip di una faccia** in tre dimensioni, come abbiamo visto nel capitolo 2. Come sappiamo, in due dimensioni il flip consiste solo nello scambio della diagonale del quadrilatero formato dall'unione dei due triangoli di cattiva qualità, se questo è convesso. In tre dimensioni, anche se più complicato, abbiamo spiegato come è possibile applicare lo scambio. Un

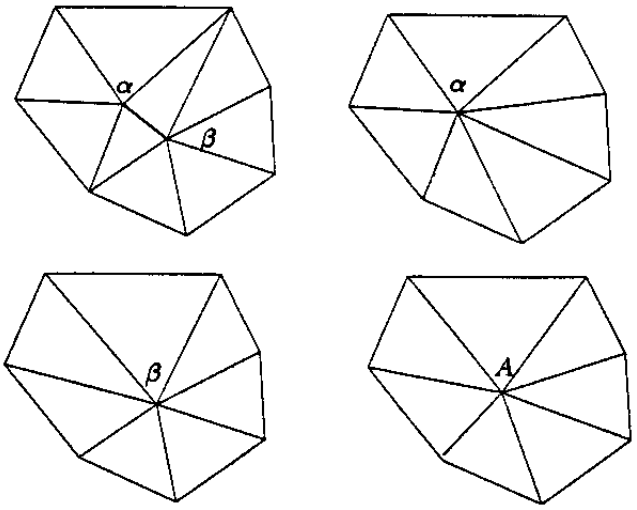

Figura 3.12: Edge collapsing.

altro metodo che possiamo usare è l'**edge collapsing**. Questo metodo consiste nel rimpiazzare un lato, $\alpha\beta$, (una faccia, f) comune a due triangoli (tetraedri) di cattiva qualità con un solo punto A (lato). In pratica, in due dimensioni, questo porta a posizionare α su β o viceversa, oppure a spostare entrambi i punti in una posizione intermedia tra α e β (Fig. 3.12).

Capitolo 4

Implementazione in ANSI C

4.1 Introduzione

In questo capitolo spiegheremo l'algoritmo usato per implementare un programma di generazione di mesh di poligoni.
Il lavoro è stato suddiviso in due problemi:

- Generazione di una discretizzazione del bordo del poligono e creazione di un insieme di punti all'interno del poligono stesso.

- Creazione della mesh del poligono.

Abbiamo, quindi, creato due programmi, implementati in ANSI C: `PUNTI.C` e `MESH.C`. Analizzeremo ora entrambi i programmi, cercando di focalizzare l'attenzione sulla struttura dati usata e sulle principali funzioni implementate.

4.2 L'inserimento di punti: `PUNTI.C`

Input e Output: Le informazioni di cui necessita in input questo programma sono quattro:

- numero di vertici del poligono,
- un intero, 0 o 1, per indicare se le coordinate dei vertici, che scriveremo di seguito, sono date in senso, rispettivamente, antiorario od orario,
- le coordinate (x, y) di ogni vertice,
- un parametro floating point `d` che mi indica il passo con cui fare la suddivisione del bordo del dominio e come distribuire i punti all'interno.

In output (su `PUNTI.DAT`) restituisce semplicemente le coordinate di tutti i punti che sono stati generati, sia quelli sul bordo che quelli interni.

Algoritmo di inserimento dei punti: Il metodo consiste nel generare, uno contenuto nell'altro, dei poligoni a partire da quello in input. I vertici dei nuovi poligoni sono ottenuti tramite intersezione di rette parallele ai lati del poligono originale poste a distanza `d` verso l'interno dei lati stessi, se i vertici sono dati in senso antiorario, verso l'esterno altrimenti. Questo fino a che i lati del nuovo poligono non si "scavalcano" o sovrappongono. Tutti i vertici sono numerati in senso orario o antiorario a seconda del verso in cui sono memorizzati i vertici del poligono originale. L'idea è poi quella di distribuire i punti di mesh sui lati di questi poligoni. Vediamo quali problemi sono stati incontrati nell'implementazione e con quali algoritmi e funzioni sono stati risolti.

Per prima cosa, osserviamo che potrebbe accadere che due lati non consecutivi

del nuovo poligono si intersechino. Cioè, si potrebbe generare un poligono in-

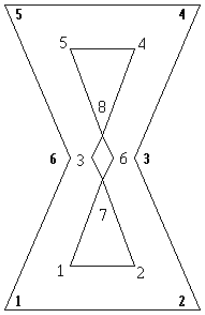

Figura 4.1: Poligono "figlio" intrecciato. Si osservi la numerazione dei vertici.

trecciato (Fig. 4.1) che potrebbe anche uscire dal poligono che l'ha generato. Quello che dobbiamo fare è dividerlo in modo da ottenere poligoni distinti e non intrecciati (Funzione `crea_polint`).

Per prima cosa, calcoliamo i punti di intersezione tra i lati, li numeriamo e li etichettiamo come "nuovi". Poi, percorrendo la figura in senso antiorario, creiamo (Funzione `crea_lista`) la lista di tutti i punti che abbiamo incontrato, comprendendo anche quelli nuovi e terminando non appena incontriamo il primo inserito. Per la figura 4.1, la lista creata sarà:

$$1\ 2\ 7_n\ 3\ 8_n\ 4\ 5\ 8_n\ 6\ 7_n\ 1$$

dove il pedice n sta ad indicare che i punti sono nuovi.

Ora scorriamo la lista fino a che non incontriamo un nuovo punto, ad esempio 7_n, memorizziamo i punti trovati e puntiamo a quel nuovo punto, da cui ripartiremo per formare il nuovo poligono. Poi, nella lista saltiamo alla posizione successiva in cui ritroviamo il nuovo punto, 7_n, continuiamo a scorrere la lista e a memorizzare i punti fino a che non incontriamo il primo numero memoriz-

zato. Questi punti mi danno il primo poligono non intrecciato. Nell'esempio di figura 4.1 il primo poligono sarà dato da: 1, 2, 7, 1, che è un triangolo.
Poi ripartiamo dal numero nuovo puntato e ripetiamo l'algoritmo. Nel nostro esempio il secondo poligono creato sarà: 7, 3, 8, 6, 7 e il terzo: 8, 4, 5, 8. Ora quello che dobbiamo fare è eliminare i poligoni che con questa numerazione sono ordinati in un senso diverso da quello del poligono originale (Funzione `crea_figlio`). In questo caso dovremo eliminare il quadrilatero 7, 3, 8, 6, 7. Quindi il nostro poligono originale ha due poligoni "figli" e ad ognuno di essi applicheremo l'algoritmo. Dopo aver costruito tutti i poligoni interni con

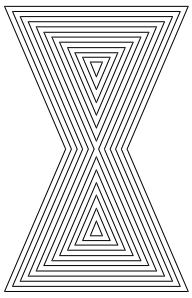

Figura 4.2: Creazione di tutti i poligoni interni. d=0.3

la funzione `crea_albero` (4.2) possiamo posizionare i punti sui lati di questi poligoni.
In pratica, su ogni lato di lunghezza `l` inseriamo la parte intera di `l/d` punti equidistanziati (Funzione `suddiniziale` e `suddsuccessive`). Poi aggiustiamo la distribuzione dei punti eliminando quelli che sono troppo vicini tra loro (Funzione `toglipuntivicini`).

Le funzioni usate: Le funzioni che sono state utilizzate in questo programma possono essere classificate in:

- Funzioni puramente algebriche.
- Funzioni di inserimento e cancellazione di elementi nelle liste.
- Funzioni che sviluppano il nucleo centrale dell'algoritmo.
- Funzioni di input e output.

A questo proposito si veda l'Appendice A.

La struttura dati: Data la natura stessa dell'algoritmo, abbiamo pensato di costruire una struttura dati ad albero per memorizzare i poligoni.

Le coordinate e i vertici sono memorizzati su liste semplicemente concatenate. Per quello che riguarda i vertici, nei campi principali ci sono un puntatore alle coordinate, un intero `nsud` e un float `l` che indicano il numero di suddividioni da attuare sul lato che ha come primo vertice il punto in questione e la lunghezza del lato stesso. La struttura `coda` contiene la label dei vertici del poligono intrecciato ed è una lista semplicemente concatenata che nei campi principali contiene le informazioni sulla lunghezza della lista stessa e un'etichetta intera, `vn` che indica quali elementi di questa coda sono "nuovi", cioè originati dalle intersezioni di lati non consecutivi del poligono.

Anche i poligoni sono memorizzati in una lista semplicemente concatenata. Nei campi principali ci sono un intero `oa` che indica l'ordine con cui sono dati i vertici dei poligoni, un intero `n` che rappresenta il numero di vertici del poligono e un puntatore alla lista dei "figli" del poligono stesso. D'altra parte, la lista dei figli, è una lista semplicemente concatenata contenente solo il puntatore al

padre.

Qui di seguito riportiamo le variabili globali usate nel programma.

- `rootc`: Root della lista delle coordinate.
- `rootcn`: Puntatore temporaneo alla lista delle coordinate. Questa variabile punta di volta in volta alla prima coordinata del poligono corrente che è stato inserita.
- `purootc`: Puntatore temporaneo alla lista delle coordinate. Questa variabile punta all'ultima coordinata inserita.
- `rootpunti`: Root della lista dei punti di cui farò la mesh.
- `purootpunti`: Puntatore all'ultimo elemento della lista dei punti di cui faremo la mesh.
- `rootp`: Root della lista dei poligoni.
- `tol`: E' il margine di errore ammesso in ogni operazione.

4.3 La creazione della mesh: `MESH.C`

Input e Output: Le informazioni di cui necessita in input questo programma sono:

- il numero di vertici del poligono,
- un intero, 0 o 1, per indicare se le coordinate dei vertici sono date in senso, rispettivamente, antiorario od orario,

- le coordinate (x, y) di ogni vertice,
- le coordinate di tutti gli altri punti di mesh.

In output, su `TRI.DAT`, restituisce i triangoli di cui è composta la mesh scrivendo le sei coordinate che corrispondono ai tre vertici di ogni triangolo sulla stessa riga. Su `LATI.DAT` restituisce i lati della triangolazione scrivendo sulla stessa riga le quattro coordinate che corrispondono ai due estremi. Su `MESH.MSH` scrive ogni punto di mesh, identificandolo con un'etichetta numerica e con le sue coordinate, e ogni triangolo, identificandolo con un'etichetta numerica e con le tre etichette dei vertici che lo compongono. Questo file è stato scritto per il software GID ([21]) che abbiamo usato per stimare la qualità delle mesh. Per la visualizzazione abbiamo usato un CAD tridimensionale ([22]).

La struttura dati: Tra le varie strutture che compongono una mesh esiste uno stretto legame, quasi gerarchico, che possiamo schematizzare come segue:

$$\textit{Punti} \longrightarrow \textit{Lati} \longrightarrow \textit{Triangoli}$$

Nel modo in cui abbiamo costruito la struttura dati abbiamo cercato di rispecchiare questo schema. Sebbene sia molto dispendioso in termini di memoria e di aggiornamento delle strutture dati stesse, offre molti vantaggi.

I vertici di mesh sono memorizzati in una lista semplicemente concatenata. Nei due campi principali sono allocate le coordinate dei punti e nel campo destro il puntatore all'elemento successivo.

I lati di ogni elemento sono memorizzati in una lista doppiamente concatenata. Nei due campi principali ci sono i puntatori alla lista delle coordinate che rappresentano gli estremi del lato stesso. Nel campo destro e sinistro ci sono

rispettivamente il puntatore all'elemento successivo e all'elemento precedente della lista.

Anche i triangoli sono memorizzati in una lista doppiamente concatenata. Nei campi principali ci sono tre puntatori alla lista di coordinate ($\mathtt{c}_1$, $\mathtt{c}_2$, $\mathtt{c}_3$) corrispondenti ai vertici del triangolo, tre puntatori alla lista dei lati ($\mathtt{l}_1$, $\mathtt{l}_2$, $\mathtt{l}_3$) e ai tre triangoli ($\mathtt{t}_1$, $\mathtt{t}_2$, $\mathtt{t}_3$) che condividono con l'elemento K i tre lati.

I puntatori alle coordinate sono memorizzati in modo che i vertici siano in senso antiorario. Lati e triangoli sono stati memorizzati in modo da ottenere le seguenti corrispondenze (Figura 4.3):

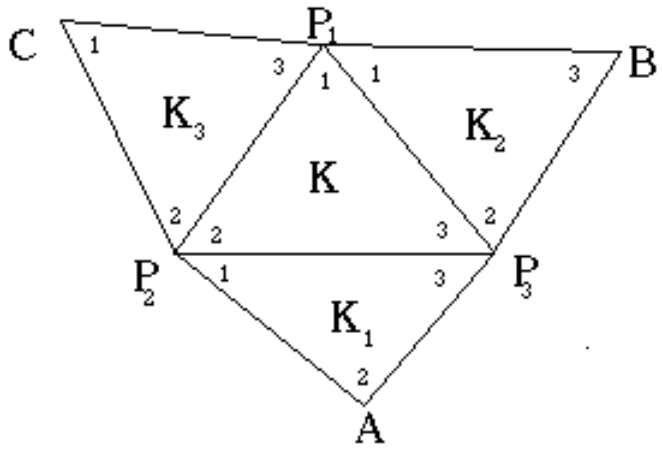

Figura 4.3: Definizione del triangolo K e dei suoi tre vicini K_i.

- P_i, $i = 1, ..., 3$ sono i vertici del triangolo,
- l_i, $i = 1, ..., 3$ sono i lati del triangolo K, dove l_i è il lato che si trova davanti a P_i,

- K_i, $i = 1, ..., 3$ sono i triangoli che con K condividono il lato l_i.

Qui di seguito indichiamo le variabili globali usate nel programma.

- `rootc`: Root della lista delle coordinate.
- `roottri`: Root della lista dei triangoli.
- `rootlati`: Root della lista dei lati della mesh.
- `triout`: Root della lista dei triangoli che cadono fuori dal poligono iniziale e che andranno eliminati per ottenere la mesh del poligono.
- `tol`: E' il margine di errore ammesso in ogni operazione.

Algoritmo di creazione della mesh: possiamo schematizzare l'algoritmo usato nel programma nel seguente modo:

1. Fase di acquisizione dei dati: dominio e punti interni.
2. Mesh:
   - Creazione del primo triangolo della mesh.
   - Inserimento dei punti in ordine lessicografico usando l'algoritmo di Boyer-Watson (Cf. Capitolo 2)
3. Ottimizzazione.

Vediamo ora in dettaglio quali sono le funzioni usate per effettuare l'algoritmo. Dopo aver acquisito il dominio e i punti di mesh con la funzione `leggidafile`, i punti sono riordinati in ordine lessicografico con la funzione `ordinapti`. Viene, poi, creato il primo triangolo della mesh unendo i primi due punti della lista

delle coordinate e il primo punto, a partire dal terzo, che formi un triangolo di area non nulla con i due punti precedenti (`primotri`).

I punti vengono ora inseriti nella triangolazione usando l'agoritmo di Boyer-Watson. Quello che dobbiamo controllare è dove si trova il punto P, che deve essere inserito, rispetto alla mesh creata. Al primo passo questa mesh sarà formata da un solo triangolo. La funzione `checkpoint` controlla se il punto si trova all'interno della triangolazione, all'esterno della triangolazione ma contenuto in qualche circumcerchio dei triangoli di mesh o se si trova all'esterno della triangolazione e di ogni circumcerchio. Nel primo caso restituisce un puntatore al triangolo a cui il punto appartiene, negli altri casi restituisce la lista dei lati di bordo nella mesh attuale. Da questa lista saranno successivamente eliminati (usando la funzione `visibile`) tutti quei lati che non sono visibili dal punto.

La funzione `crea_triangoli` si occupa dell'analisi di ogni punto e, dopo il controllo del punto stesso e dopo il test sulla visibilità, crea i nuovi triangoli unendo il punto con la lista dei lati visibili. Ogni triangolo è poi ordinato in modo che i suoi vertici siano in senso antiorario (funzione `orientatri`).

Come abbiamo spiegato nel Capitolo 2, quella che così abbiamo generato è la triangolazione del piú piccolo insieme convesso contenente i punti. Quello che rimane da fare per ottenere la mesh del poligono di partenza è eliminare i triangoli che cadono fuori dalla triangolazione. Il primo passo è quello di identificarli (funzione `inout`) controllando dove cade il baricentro di ogni triangolo. Questi triangoli vengono inclusi in una lista e poi eliminati prima di attuare l'ottimizzazione.

Il metodo con cui abbiamo eseguito l'ottimizzazione è quello descritto nel Capitolo 3. Si individuano quei triangoli che hanno angoli minimi piccoli, per esempio, minori di 20°, si individua il vertive P a cui appartiene quest'angolo. Questo punto viene poi spostato nel baricentro della figura ottenuta dall'unione di tutti gli elementi della mesh che hanno P come vertice (funzione `smooth`).

4.4 Esempi

Qui di seguito riporteremo alcuni esempi del funzionamento del programma su diversi domini e con diversa scelta del parametro `d`.

Le figure 4.4 e 4.5 mostrano lo stesso dominio con diversa scelta del parametro e quindi con diverso numero di punti di mesh. A fianco ci sono i grafici relativi alla qualità della mesh.

Le altre figure, invece, mostrano lo stesso dominio con lo stesso numero di punti di mesh ma prima (figura in alto) e dopo (figura in basso) l'ottimizzazione attuata con la funzione `smooth`. A fianco abbiamo sempre il grafico relativo alla qualità della mesh.

Osserviamo che, in ogni grafico, l'asse x rappresenta il valore dell'angolo minimo di ogni triangolo misurato in gradi e l'asse y rappresenta il numero degli elementi che hanno quell'angolo.

Inoltre con N abbiamo indicato il numero di punti di mesh e con N_T il numero di triangoli.

Figura in alto	**N**= 381	$\mathbf{N}_T$= 692	**d**= 0.06
Figura in basso	**N**= 821	$\mathbf{N}_T$= 1540	**d**= 0.04

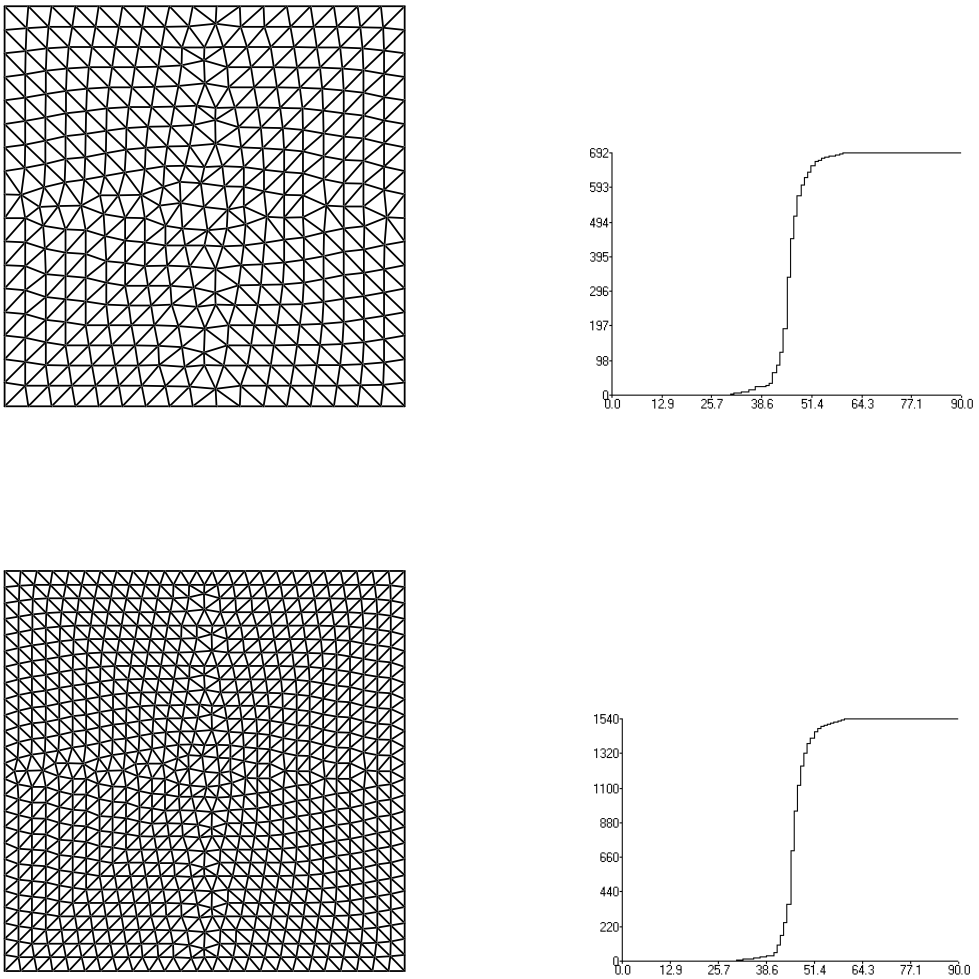

Figura 4.4: Dominio `quad.ini` con diversi valori del parametro.

Figura in alto	**N**= 169	$\mathbf{N}_T$= 298	**d**= 0.15
Figura in basso	**N**= 547	$\mathbf{N}_T$= 1000	**d**= 0.08

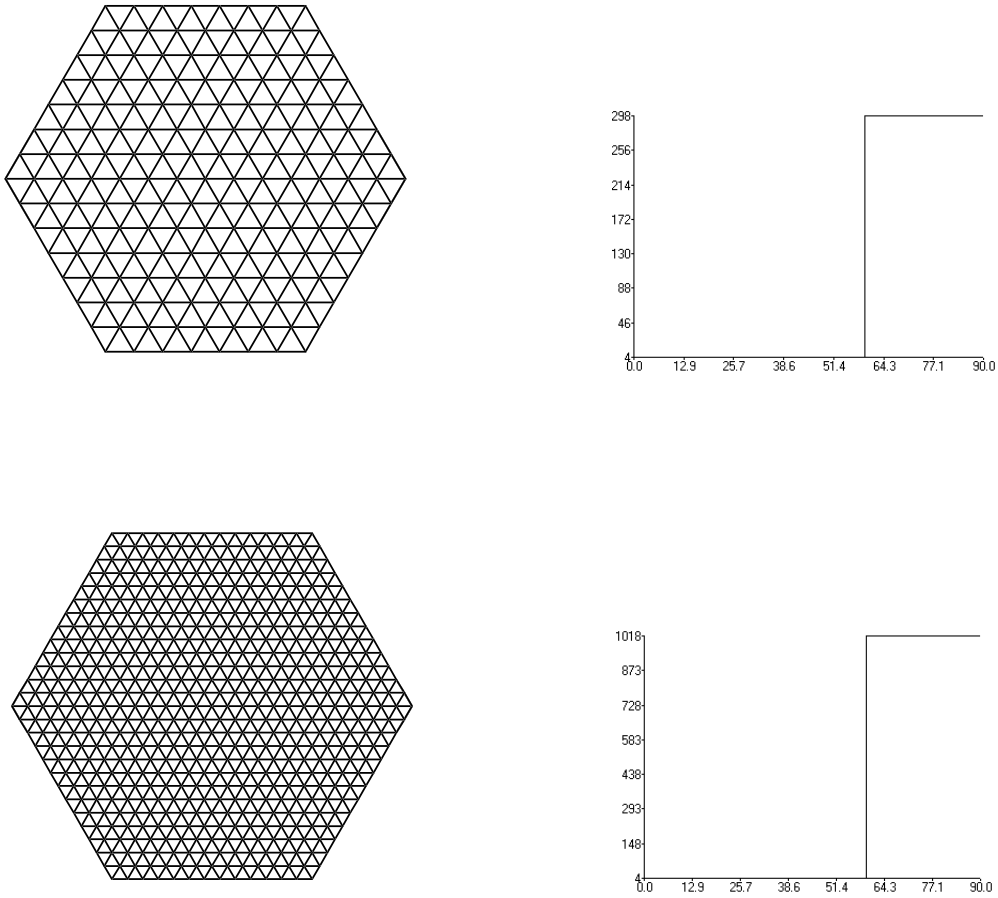

Figura 4.5: Dominio `esa.ini` con diversi valori del parametro.

$\mathbf{N}$= 591 $\mathbf{N}_T$= 1129 $\mathbf{d}$= 0.2

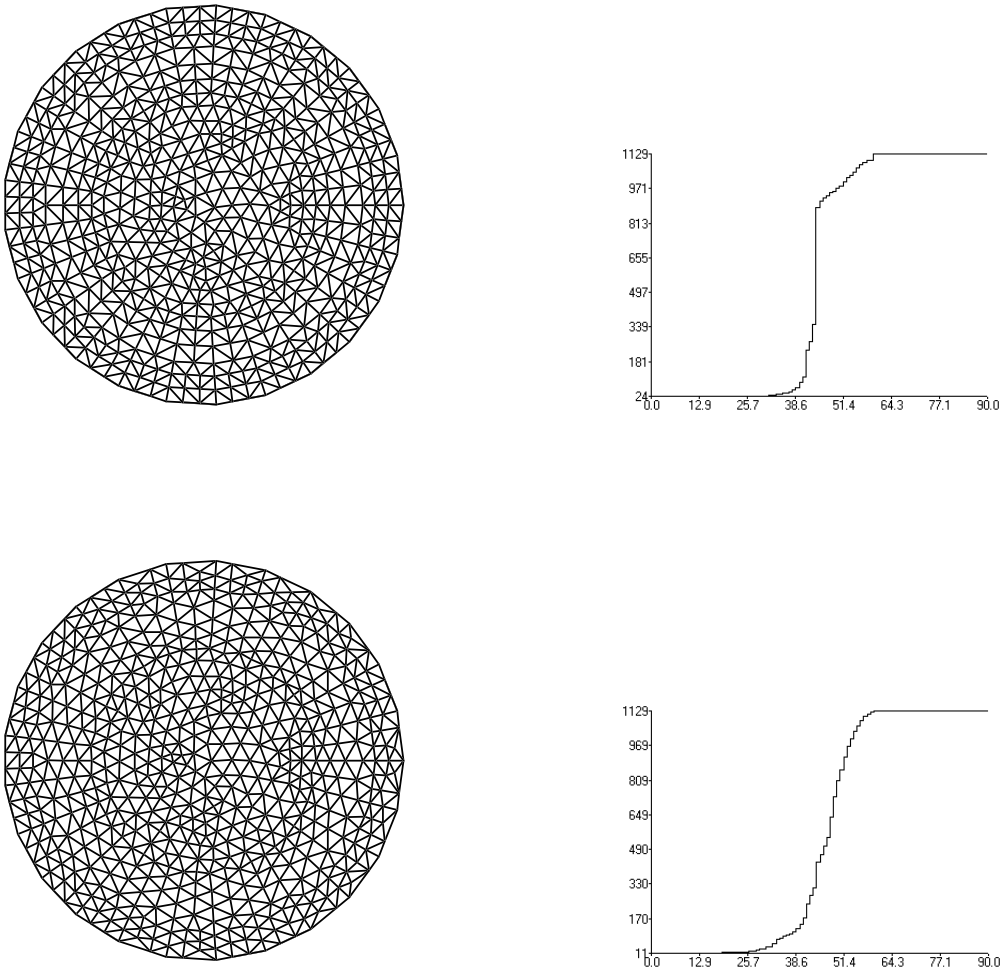

Figura 4.6: Dominio `circ.ini`.

$\mathbf{N}$= 358 $\mathbf{N}_T$= 611 $\mathbf{d}$= 0.2

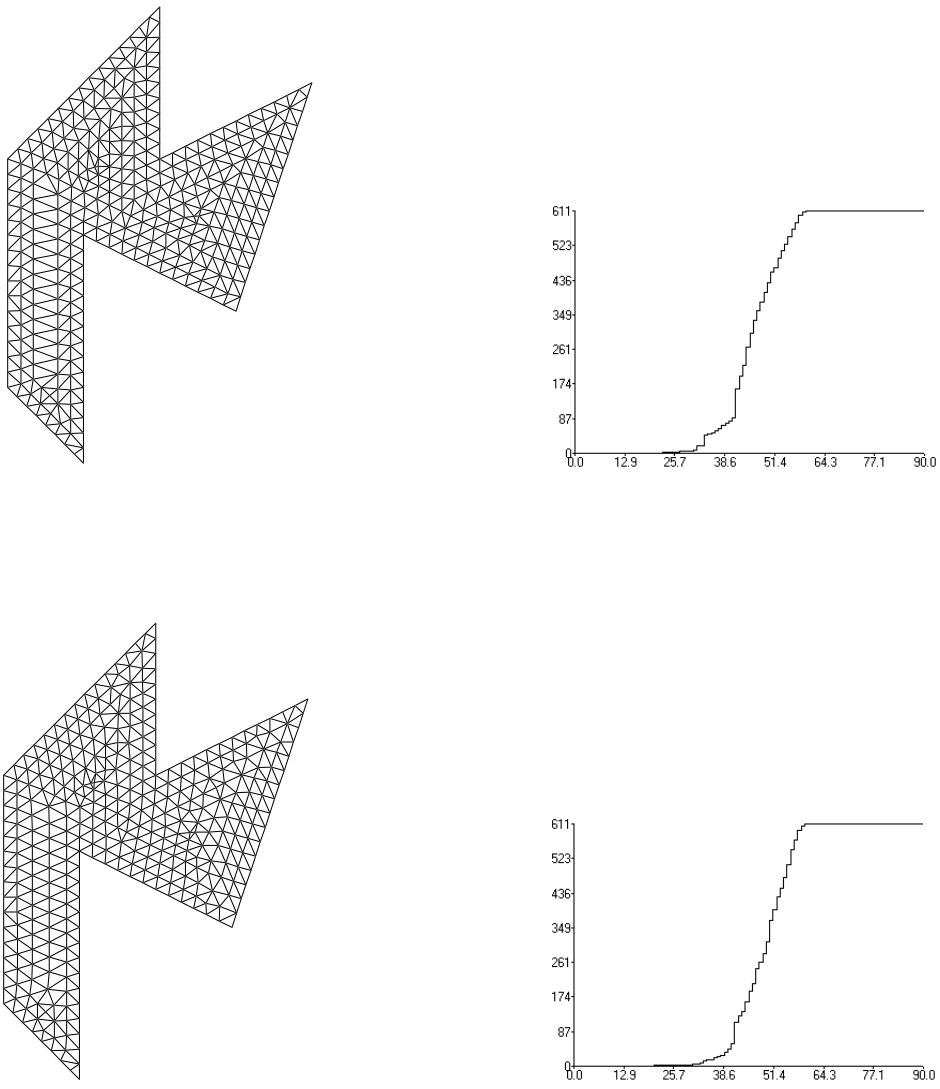

Figura 4.7: Dominio `poli1.ini`.

$\mathbf{N}$= 360 $\mathbf{N}_T$= 571 $\mathbf{d}$= 0.3

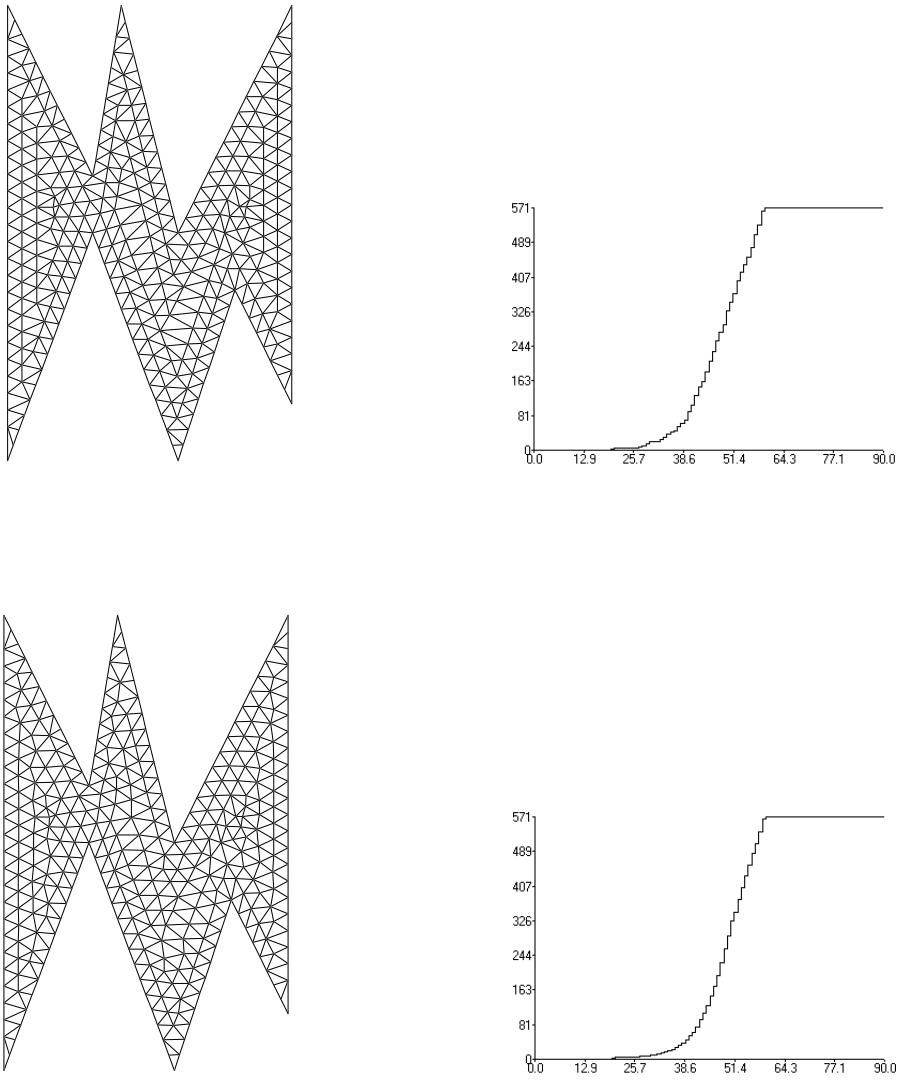

Figura 4.8: Dominio `poli2.ini`.

$\mathbf{N}= 232 \quad \mathbf{N}_T= 398 \quad \mathbf{d}= 0.3$

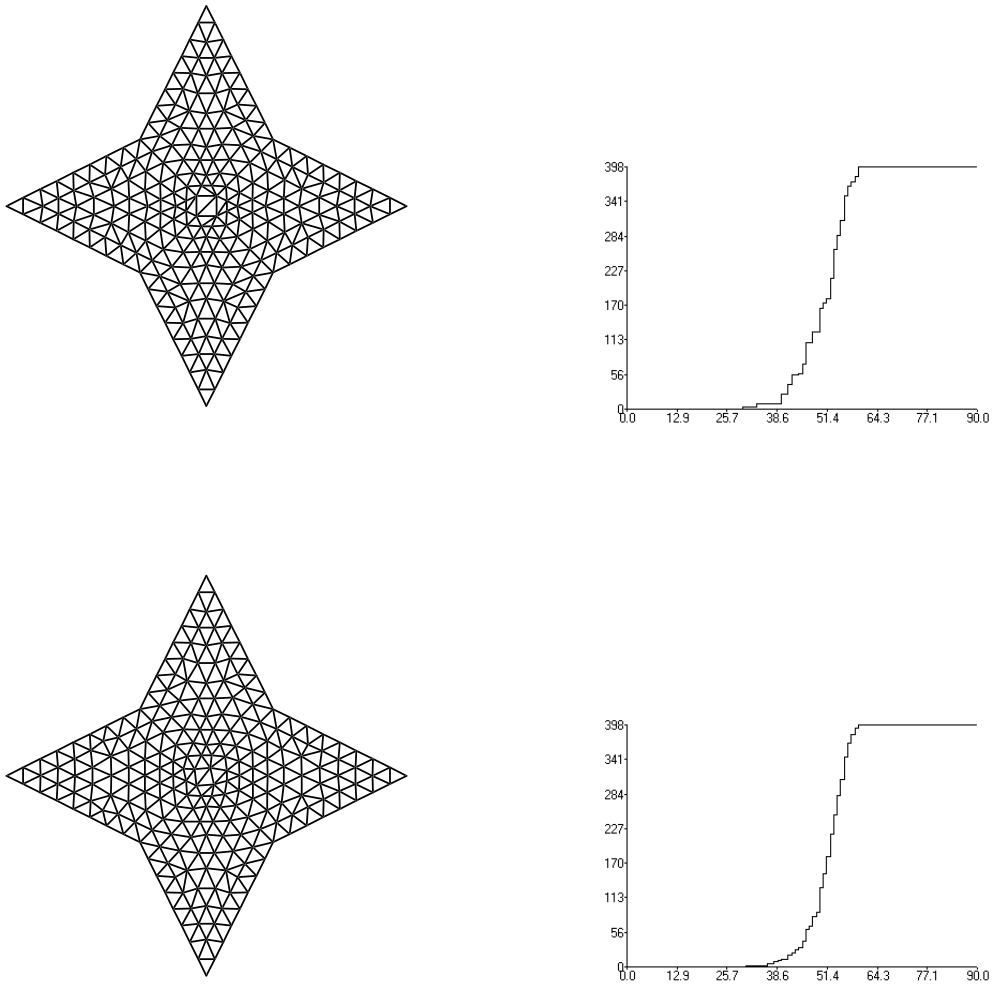

Figura 4.9: Dominio `stella.ini`.

$\mathbf{N}$= 454 $\mathbf{N}_T$= 810 $\mathbf{d}$= 0.2

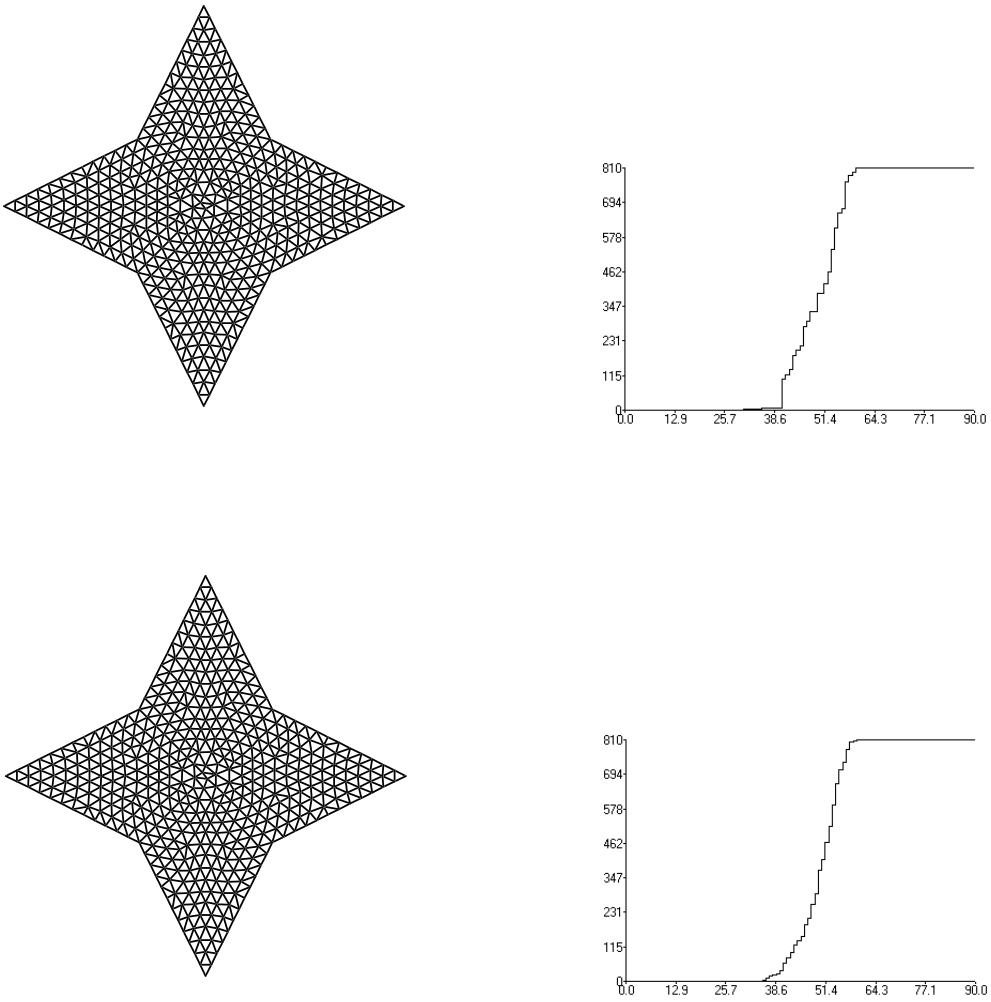

Figura 4.10: Dominio `stella.ini` con diverso valore del parametro.

Appendice A

Il programma PUNTI.C

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>

/*DEFINIZIONE DELLE STRUTTURE USATE*/
typedef struct listacoordinate * listac_ptr;
typedef struct listacoordinate
{
 double x,y;
 listac_ptr next;
} listac;

typedef struct listavertici * listav_ptr;
typedef struct listavertici
{
 int nsud;    /*# di suddivisioni sul la to che ha come
                primo vertice c*/
 double l;    /*lunghezza del lato sopra*/
 listac_ptr c ;
 listav_ptr next;
} listav ;

typedef struct listafigli * listaf_ptr;
typedef struct poligono
{
 int oa;
 int n;
 listaf_ptr lf;
 listav_ptr lv;
} listap;

typedef struct listafigli
{
 listap *pol;
 listaf_ptr next;
```

```
} listaf;

typedef struct coda * coda_ptr;
typedef struct coda
{
 int n;
 int vn;
 listac_ptr c;
 coda_ptr next;
} coda;

/*DEFINIZIONE DELLE FUNZIONI USATE*/

/*Funzioni di lettura e stampa*/
void stampa_albero(listap *);
listaf *leggidafile(listaf *);
void stampalistac(listac_ptr);
void stampalistav(listav_ptr ,int );

/*Funzioni di utilità per le liste*/
listac_ptr inscoo(listac_ptr ,double,double,listac_ptr *);
listav_ptr insv(listav_ptr ,listac_ptr );
coda_ptr inscoda(coda_ptr ,int,int,listac_ptr);
void cancella_listav(listav_ptr ,int );
void cancella_listac(listac_ptr);
void cancella_coda(coda_ptr );
coda_ptr trova_poscoda(int,coda_ptr ,double,double);
void aggiungi(listaf *);
listav_ptr cercapenultimo(listav_ptr ,int n);
coda_ptr cercapenultimoc(coda_ptr ,int n);
listav_ptr trovaprecedente(listav_ptr ,int);
int cambiavn(int,coda_ptr );

/*Funzioni algebriche*/
double dot(double a[],double b[]);
int convesso(listav_ptr,int );
void toglipuntivicini(listac_ptr,double);
listav_ptr aggiustapunti(listav_ptr ,int *,double);
double distanza(int,double,double,double);

/*Funzioni principali*/
listaf *crea_figlio(coda_ptr ,int,double);
coda_ptr crea_lista(listap *);
listap *creapolint(listap *,double,double,listac_ptr *);
listap *crea_albero(listap *,double);
void suddsuccessive(listav_ptr ,int,double);
void suddiniziale(double ,listaf *);

/*VARIABILI GLOBALI*/
FILE *f,*g,*f1;
listac_ptr rootc;       /*punt. alla lista delle coord.*/
```

```
listap *rootp;          /*punt. alla lista dei poligoni*/
listac_ptr rootcn;      /*punt. temporaneo alla lista delle coord.
                          dei poligoni interni*/
listac_ptr purootc;     /*punt. all'ultimo elemento della lista delle
                          coord. inserito*/
listac_ptr  purootpunti;/*punt. all'ultimo elemento della lista dei
                          punti gia' inseriti cui fare la mesh*/
listac_ptr rootpunti;   /*Punt. alla lista dei punti di mesh*/
double tol;
int stop;

/*PROGRAMMA PRINCIPALE*/
main()
{
 double d;
 int i,go,nvert;
 listap *ppol;
 listav_ptr plv;
 listaf_ptr pfigli;
 listac_ptr appo;
 f=fopen("albero.dat","w");
 g=fopen("poligono.in","r");
 f1=fopen("punti.dat","w");
 tol=1.e-10;
 do
  {
   rootc=NULL;
   rootpunti=NULL;
   purootpunti=NULL;
   rootp=(listap *)malloc(sizeof(listap));
   if(rootp==NULL)
    printf("Errore nell' alloc. di rootp\n");
   rootp->lv=NULL;
   rootp->lf=NULL;
   rootp->lf=leggidafile(rootp->lf);
   printf("Inserisci d\n");
   scanf("%lf",&d);
   if(d<tol) {printf("ERRORE d troppo piccola");return 0;}
   pfigli=rootp->lf;
   while(pfigli!=NULL)
   {
    listac_ptr plc;
    plc=purootpunti;
    suddiniziale(d,pfigli);
    if(plc==NULL) toglipuntivicini(rootpunti,d);
    else toglipuntivicini(plc->next,d);
    ppol=pfigli->pol;
    ppol=crea_albero(ppol,d);
    stampa_albero(ppol);
    pfigli=pfigli->next;
   }
```

```
   appo=rootpunti;
   while(appo!=NULL)
   {
    fprintf(f1,"%lf %lf\n",appo->x,appo->y);
    appo=appo->next;
   }
   printf("0- per continuare\n");
   scanf("%d",&go);
  }
  while(go==0);
  return 0;
 }

/*Per ogni poligono crea la lista numerata dei
vertici per poterlo sciogliere*/
coda_ptr crea_lista(listap *pol)
{
 coda_ptr pcoda,rootcoda;
 int cambia,i,n,j,inseg,npn,go;
 double x,y;
 listav_ptr plv,plvi,plvj,plvjp;
 listac_ptr appo,pcpn;
 pcoda=NULL;
 plv=pol->lv;
 n=pol->n;
 /*Inizializza la coda*/
 for(i=0;i<=n-1;i++)
 {
  pcoda=inscoda(pcoda,i,0,plv->c);
  if(i==0) rootcoda=pcoda;
  plv=plv->next;
  pcoda=pcoda->next;
  if(i==0) pcoda=rootcoda;
 }
 pcoda=inscoda(pcoda,0,0,(pol->lv)->c);/*Chiudo la coda*/
 plvi=pol->lv;
 npn=n;
 for(i=0;i<=n-1;i++)/*Scorro sui pti*/
 {
  plvj=plvi->next->next;
  for(j=i+2;j<=n-1;j++) /*Scorro sui segmenti*/
  {
   if(i!=0 || j!=n-1)
   {
    /*Calcoliamo le intersezioni e poi le inseriamo*/
    inseg=interseg(plvi->c,plvi->next->c,plvj->c,
          plvj->next->c,&x,&y);
    if(inseg==0)
    {
     pcpn=inscoo(rootcn,x,y,&appo);
     pcoda=trova_poscoda(j,rootcoda,x,y);
```

```
     pcoda=inscoda(pcoda,npn,1,appo);
     pcoda=trova_poscoda(i,rootcoda,x,y);
     pcoda=inscoda(pcoda,npn,1,appo);
     npn++;
    }
   }
   plvj=plvj->next;
  }
  plvj=pol->lv;
  for(j=0;j<=n-1;j++)
  {
   if((i!=0 && i!=j && j!=i-1) || (i==0 && j!=n-1 && j!=0))
   {
    /*Controlliamo se i vert. sono anche di intersezioni*/
    inseg=insegv(plvj->c,plvj->next->c,plvi->c->x,plvi->c->y);
    if(inseg==0)
    {
     pcoda=trova_poscoda(j,rootcoda,plvi->c->x,plvi->c->y);
     cambia=cambiavn(i,rootcoda);
     pcoda=inscoda(pcoda,i,0,plvi->c);
    }
   }
   plvj=plvj->next;
  }
  plvi=plvi->next;
 }
 plvi=pol->lv;
 for(i=0;i<=n-1;i++)
 {
  plvj=plvi->next;
  for(j=i+1;j<=n-1;j++)
  {
   if(fabs(plvi->c->x-plvj->c->x)<=tol
        && fabs(plvi->c->y-plvj->c->y)<=tol)
  {
  plvjp=trovaprecedente(plvj,n);
  if(insegv(plvjp->c,plvj->c,plvi->next->c->x,
     plvi->next->c->y)==1 || insegv(plvj->c,plvj->
     next->c,plvi->next->c->x,plvi->next->c->y)==2)
           scambia(i,j,rootcoda,0);
  else scambia(i,j,rootcoda,1);
 }
 plvj=plvj->next;
}
plvi=plvi->next;
}
return rootcoda;
}

/*Crea un figlio a partire da pcoda e ritorna il
punt. al primo figlio creato*/
```

```
listaf_ptr crea_figlio(coda_ptr pcoda,int oa,double d)
{
 listav_ptr plv,appov;
 listaf_ptr pfiglio;
 listac_ptr appo;
 listac_ptr plc;
 int i,nv,spia,ccw;
 double l,a[2],x,y;
 int j,npd,nsud;
 coda_ptr pc,pcodas;/*pcodas= prossimo figlio*/
 spia=1;/*Spia=0 -> Da pcoda non chiudo alcun poligono*/
 plv=NULL;
 pc=pcoda;
 pcodas=NULL;
 pfiglio=NULL;
 nv=0;
 for(;;)
 {
  if (pcodas==NULL && nv!=0 && pc->vn==1) pcodas=pc;
  if(pc->n==pcoda->n && nv!=0)
  {
   appov=cercapenultimo(plv,nv);
   appov->next=plv;
   break;
  }
  plv=insv(plv,pc->c);
  nv++;
  if(pc->vn ==1 && nv!=1)
  {
   i=pc->n;
   for(;;)
   {
    pc=pc->next;
    if (pc==NULL)
    {
     spia=0;
     break;
    }
    if (pc->n==i) break;
   }
   if (spia==0) break;
  }
  pc=pc->next;
  if (pc==NULL)
  {
   spia=0;
   break;
  }
 }
 if (spia==1)
 {
```

```
  plv=aggiustapunti(plv,&nv,d);
  ccw=sensoantiorario(plv,nv);
  if (ccw==oa)
  {
   pfiglio=(listaf_ptr )malloc(sizeof(listaf));
   if(pfiglio==NULL)
        printf("Errore nell' alloc. di pfiglio\n");
   pfiglio->pol=(listap *)malloc(sizeof(listap));
   if(pfiglio->pol==NULL)
        printf("Errore nell' alloc. di pfiglio->pol\n");
   pfiglio->pol->lv=(listav_ptr)malloc(sizeof(listav));
   if(pfiglio->pol->lv==NULL)
        printf("Errore nell' alloc. di pfilglio->lv\n");
   pfiglio->pol->n=nv;
   pfiglio->pol->lv=plv;
   pfiglio->pol->oa=oa;
   pfiglio->pol->lf=NULL;
   if (pcodas != NULL) pfiglio->next=crea_figlio(pcodas,oa,d);
   else pfiglio->next=NULL;
  }
  else if (pcodas != NULL)
  {
   cancella_listav(plv,nv);
   pfiglio=crea_figlio(pcodas,oa,d);}
  }
  else if (pcodas != NULL)
  {
   pfiglio=crea_figlio(pcodas,oa,d);
  }
  return pfiglio;
 }

/*Scorre coda e scambia indice j lo scambia con i*/
scambia(int j,int i,coda_ptr rootcoda,int vn)
{
 listac_ptr  pcoo;
 do{
  if(rootcoda->n==i)
  {
   rootcoda->vn=vn;
   pcoo=rootcoda->c;
  }
  if(rootcoda->n==j)
  {
   rootcoda->c=pcoo;
   rootcoda->vn=vn;
   rootcoda->n=i;
  }
  rootcoda=rootcoda->next;
 }while(rootcoda!=NULL);
 return 0;
```

```
}

/*Da' il punt. a coda dopo j in cui devo inserire (x,y)*/
coda_ptr trova_poscoda(int j,coda_ptr rootcoda,double x,double y)
{
 coda_ptr pcoda;
 double x1,x2,y1,y2;
 pcoda=rootcoda;
 for(;;)
 {
  if(pcoda->n==j) break;
  pcoda=pcoda->next;
 }
 for(;;)
 {
  x1=pcoda->c->x; y1=pcoda->c->y;
  x2=pcoda->next->c->x; y2=pcoda->next->c->y;
  if(((x>=x1-1.e-10 && x<=x2+1.e-10)||
    (x>=x2-1.e-10 && x<=x1+1.e-10))&&
    ((y>=y1-1.e-10 && y<=y2+1.e-10)||
    (y>=y2-1.e-10 && y<=y1+1.e-10))) break;
  pcoda =pcoda->next;
 }
 return pcoda;
}

/*Prodotto scalare*/
double dot(double a[2],double b[2])
{
 return  (a[0]*b[0]+a[1]*b[1]);
}

/*Da' il punt. al primo el. di una lista di coord. dove inseriamo
(x,y),in appo ritorna il punt. alle coord. (x,y)*/
listac_ptr inscoo(listac_ptr q,double x,double y ,listac_ptr *appo)
{
 int i;
 if (q==NULL)
 {
  q=(listac_ptr)malloc(sizeof(listac)) ;
  if(q==NULL)
       printf("Errore nell' alloc. di q\n");
  *appo=q;
  q->x=x;
  q->y=y;
  q->next=NULL;
 }
 else  q->next=inscoo(q->next,x,y,appo);
 return q;
}
```

```
/*Da' il punt. al primo elemento di una lista di
vertici dopo averci inserito c*/
listav_ptr insv(listav_ptr q,listac_ptr c )
{
 int i;
 if (q==NULL)
 {
  q=(listav_ptr)malloc(sizeof(listav));
  if(q==NULL)
       printf("Errore nell' alloc. di q in insv\n");
  q->c=c;
  q->next=NULL;
  q->nsud=-1;
  q->l=0.0;
 }
 else q->next=insv(q->next,c);
 return q;
}

/*Se trova i nella coda se non e' il primo elem. ci mette 1
(nuovo) e da' 1, se invece i e' il primo elem. della coda
lascia tutto com'e' e ritorna 0*/
int cambiavn(int i,coda_ptr rootcoda)
{
 int cont;
 cont=1;
 do {
  if(rootcoda->n==i)
  if(cont==1)  return 0; /*Niente scambi*/
  else
  {
   rootcoda->vn=1;
   return 1; /*Uno scambio*/
  }
  rootcoda=rootcoda->next;
  cont++;
 }while(rootcoda!=NULL);
 return 0;
}

/*Da' il punt. pcoda dopo aver inserito il vertice ,n ,se e'
nuovo o vecchio (vn) e un punt. alle coord.*/
coda_ptr inscoda(coda_ptr pcoda,int n,int vn,listac_ptr pcoo)
{
 coda_ptr pcvec;
 if(pcoda!=NULL)
 {
  pcvec=pcoda->next;
  pcoda->next=(coda_ptr )malloc(sizeof(coda));
  if(pcoda->next==NULL)
       printf("Errore nell' alloc. in inscoda\n");
```

```
  pcoda->next->n=n;
  pcoda->next->next=pcvec;
  pcoda->next->vn=vn;
  pcoda->next->c=pcoo;
 }
 else
 {
  pcoda=(coda_ptr )malloc(sizeof(coda));
  if(pcoda==NULL)
       printf("Errore nell' alloc. in inscoda2\n");
  pcoda->n=n;
  pcoda->next=NULL;
  pcoda->vn=vn;
  pcoda->c=pcoo;
 }
 return pcoda;
}

/*Cerca il penultimo punt. nella lista di vertici
che parte da p e lunga n*/
listav_ptr cercapenultimo(listav_ptr p,int n)
{
 int i;
 listav_ptr pappo;
 pappo=p;
 for (i=1;i<=n-1;i++) pappo=pappo->next;
 return pappo;
}

/*Cerca il penultimo punt. nella lista di vertici
che parte da p e lunga n*/
coda_ptr cercapenultimoc(coda_ptr p,int n)
{
 int i;
 coda_ptr pappo;
 pappo=p;
 for (i=1;i<=n-1;i++) pappo=pappo->next;
 return pappo;
}

/*Cerca il punt. prec. a pl in una lista ciclica lunga n*/
listav_ptr trovaprecedente(listav_ptr pl,int n)
{
 int i;
 listav_ptr pappo;
 pappo=pl;
 for(i=1;i<=n-1;i++) pappo=pappo->next;
 return pappo;
}

/* Vede se il vertice (x,y)  sta sul segmento  da pc1 a pc2.
```

```
  OUTPUT DI INSEGV:
  0- il punto e' tra il primo e il secondo.
  1- il punto e' il primo.
  2- il punto e' il secondo.
  3- il punto e' sulla retta ma esterno al segmento.
  -1- il punto non e' sulla retta.
  -2- errore:i due punti coincidono.*/
int insegv(listac_ptr pc1,listac_ptr pc2,double x,double y)
{
 double t,x1,x2,y1,y2;
 x1=pc1->x;  y1=pc1->y;
 x2=pc2->x;  y2=pc2->y;
 if(fabs(x1-x2)>tol || fabs(y1-y2)>tol)
 {
  if(fabs((y2-y1)*(x-x1)-(y-y1)*(x2-x1))<=tol)
  {
   if(fabs(x2-x1)>tol)  t=(x-x1)/(x2-x1);
   else t=(y-y1)/(y2-y1);
   if(t>=-tol && t<=tol ) return 1;
   else if (t>=1.-tol && t<=1.+tol) return 2;
   else if(t>0. && t<1.) return 0;
   else return 3;
  }
  else return -1;
 }
 else return -2;
}

/*Cerca le intersezioni tra il
segmento da pc1 a pc2 e da pc3 a pc4.
OUTPUT DI ITERSEG:
0- il punto di intersez. e' tra i segmenti estremi esclusi
1- il punto e' fuori dai segmenti.
-1- non ci sono intersezioni.*/
int interseg(listac_ptr pc1,listac_ptr pc2,listac_ptr pc3,
             listac_ptr pc4, double *x,double *y)
{
 double det,t,u,x1,y1,x2,y2,x3,y3,x4,y4;
 x1=pc1->x;  y1=pc1->y;
 x2=pc2->x;  y2=pc2->y;
 x3=pc3->x;  y3=pc3->y;
 x4=pc4->x;  y4=pc4->y;
 det=-(x2-x1)*(y4-y3)+(x4-x3)*(y2-y1);
 if(fabs(det)>1.e-10)
 {
  u=((x2-x1)*(y3-y1)-(x3-x1)*(y2-y1))/det;
  t=(-(x3-x1)*(y4-y3)+(y3-y1)*(x4-x3))/det;
  if(u>1.e-10 && u<(1.-1.e-10) && t>1.e-10 &&
     t<(1.-1.e-10))
  {
   *x=t*x2+(1-t)*x1;
```

```
   *y=t*y2+(1-t)*y1;
   return 0;
  }
  else return 1;
 }
 else return -1;
}

/*Vede se un poligono e' orientato in senso orario (1)
controllando se esiste almeno un lato che non ha
nessun altro vertice alla sua sinistra.*/
int sensoantiorario(listav_ptr plv,int n)
{
 int spia;
 int j,k;
 double det;
 listav_ptr pvj,pvk;
 pvj=plv;
 for(j=1;j<=n;j++)
 {
  spia=1;
  pvk=pvj->next->next;
  for(k=1;k<=n-2;k++)
  {
   det=(pvj->next->c->x - pvj->c->x)*(pvk->c->y - pvj->c->y)-
       (pvk->c->x - pvj->c->x)*(pvj->next->c->y - pvj->c->y);
   if(det>0.+tol)
   {
    spia=0;/*ok*/
    break;
   }
   pvk=pvk->next;
  }
  if(spia==1)  break; /*senso orario*/
  pvj=pvj->next;
 }
 return spia;
}

/*Calcola l'intersezione tra le rette // ai segmenti passanti
a distanza dv e dw (positiva per la sinistra) rispettivamente.
OUTPUT: 1- PARALLELE
        0- RESTITUISCE L' INTERSEZIONE (x,y) */
int xrette(listac_ptr pcv1,listac_ptr pcv2,listac_ptr pcw1,
    listac_ptr pcw2,double dv,double dw,double *x,double *y)
{
 int io;
 double a11,a12,a21,a22,b1,b2,det;
 io=1;
 a11 = pcv2->y - pcv1->y;
 a12 = pcv1->x - pcv2->x;
```

```
 a21 = pcw2->y - pcw1->y;
 a22 = pcw1->x - pcw2->x;
 det = a11*a22 - a21*a12;
 if (fabs(det) <= tol) return io;
 b1 = pcv1->x*a11 + pcv1->y*a12;
 if (dv != 0.) b1 = b1 - dv*sqrt(a11*a11 + a12*a12);
 b2 = pcw1->x*a21 + pcw1->y*a22;
 if (dw != 0.) b2 = b2 - dw*sqrt(a21*a21 + a22*a22);
 *x = (b1*a22 - b2*a12)/det;
 *y = (b2*a11 - b1*a21)/det;
 io=0;
 return io;
}

/*Creazione di tutti i pol. interni a quello iniziale*/
listap *creapolint(listap *ppext,double dv,
                   double dw,listac_ptr *pcoo)
{
 listap *ppol;
 int spia,i ,j,n,nv;
 listav_ptr plvi,plv ,pvlist;
 coda_ptr pcoda,pucoda;
 listac_ptr appoc;
 double x,y,distv,distw;
 appoc=NULL;
 *pcoo=NULL;
 n=ppext->n;
 if(n<=2)  return NULL;
 ppol=(listap *)malloc(sizeof(listap));
 if(ppol==NULL)
       printf("Errore nell' alloc. in creapolint\n");
 ppol->n=n;
 ppol->lf=NULL;
 ppol->oa=ppext->oa;
 ppol->lv=NULL;
 plvi=ppext->lv;
 /*Calcolo le intersezioni e le inserisco*/
 for(i=1;i<=n;i++)
 {
  distv=distanza(plvi->nsud,plvi->l,dv,0.);
  distw=distanza(plvi->next->nsud,plvi->next->l,dw,0.);
  spia=xrette(plvi->c,plvi->next->c,plvi->next->c,
              plvi->next->next->c,distv,distw,&x,&y);
  if(spia==1)
  {
   printf("ERRORE sono //");
   free(ppol);
   return NULL;
  }
  else
  {
```

```
   *pcoo=inscoo(*pcoo,x,y,&appoc);
   ppol->lv=insv(ppol->lv,appoc);
  }
  plvi=plvi->next;
 }
 pvlist=cercapenultimo(ppol->lv,ppol->n);
 pvlist->next=ppol->lv;  /*La lista e' ciclica*/
 plvi=ppext->lv->next;
 pvlist=ppol->lv;
 /*Fase in cui scartiamo le figure mal ordinate*/
 nv=n;
 pucoda=NULL;
 for(i=1;i<=n;i++)
 {
  if(orienta(plvi->c,plvi->next->c,pvlist->c,
        pvlist->next->c)==1) pucoda=inscoda(pucoda,-1,1,NULL);
  else pucoda=inscoda(pucoda,-1,0,NULL);
  if(i==1) pcoda=pucoda;
  pucoda=pucoda->next;
  if(i==1) pucoda=pcoda;
  plvi=plvi->next;
  pvlist=pvlist->next;
 }
 plvi=ppext->lv->next;
 pvlist=ppol->lv;
 for(i=1;i<=n;i++)
 {
  if(pcoda->vn==0)
  {
   pvlist->c->x=(pvlist->c->x+pvlist->next->c->x)/2.0;
   pvlist->c->y=(pvlist->c->y+pvlist->next->c->y)/2.0;
   pvlist->next=pvlist->next->next;
   nv--;
  }
  else  pvlist=pvlist->next;
  pcoda=pcoda->next;
 }
 ppol->n=nv;
 ppol->lv=pvlist;
 if(nv>=3) return ppol;
 else {free(ppol);return NULL;}
}

/*Vede se i lati da pc1 a pc2 e da pcn1 a pcn2 ,che sono //,
sono orientati allo stesso modo(1, 0 altrimenti)*/
int orienta(listac_ptr pc1,listac_ptr pc2,listac_ptr pcn1,
            listac_ptr pcn2)
{
 if((pc2->x-pc1->x)*(pcn2->x-pcn1->x)>=0. &&
    (pc2->y-pc1->y)*(pcn2->y-pcn1->y)>=0.) return 1;
 else return 0;
```

```
}

/*Crea tutto l'albero a partire da rootpi*/
listap *crea_albero(listap *rootpi,double d)
{
 double dv,dw,x,y;
 coda_ptr pcoda; /*punt. al pol. intrecciato*/
 int i,go,io,nv,j,ccw;
 listap *ppol,*rootpint;
 listav_ptr pvlist;
 listac_ptr appo,plc;
 listaf_ptr pfappo,pfappoprec;
 coda_ptr pcdap;
 dv=d;dw=d;
 if(rootpi==NULL) return NULL;
 if(rootpi->n>=3)
 {
  rootcn=NULL;
  ppol=creapolint(rootpi,dv,dw,&rootcn);
  if(ppol==NULL)
  {
   if(convesso(rootpi->lv,rootpi->n)==1)
   {
    x=y=0.0;
    pvlist=rootpi->lv;
    for(j=1;j<=rootpi->n;j++)
    {
     x=x+pvlist->c->x;  y=y+pvlist->c->y;
     pvlist=pvlist->next;
    }
    x=x/rootpi->n; y=y/rootpi->n;
    purootpunti=inscoo(purootpunti,x,y,&appo);
    purootpunti=appo;
   }
   rootpi->lf=NULL;
   printf("Non ha piu' figli\n");
  }
  else
  {
   pcoda=crea_lista(ppol);
   pcdap=pcoda;
   rootpi->lf=crea_figlio(pcoda,ppol->oa,d);
   cancella_coda(pcoda);
   if (rootpi->lf!=NULL && rootcn!=NULL) aggiungi(rootpi->lf);
   pfappo=rootpi->lf;
   j=1;
   while(pfappo != NULL)
   {
    nv=(pfappo->pol)->n;
    pvlist= (pfappo->pol)->lv;
    j++;
```

```
    plc=purootpunti;
    suddsuccessive(pfappo->pol->lv,nv,d);
    toglipuntivicini(plc,d);
    pfappo=pfappo->next;
   }
   free(ppol);
  }
  if(rootpi->lf==NULL) return rootpi;
  else
  {
   pfappo=rootpi->lf;
   do
   {
    pfappo->pol=crea_albero(pfappo->pol,d);
    pfappo=pfappo->next;
   }
   while(pfappo!=NULL);
  }
 }
 else rootpi->lf=NULL;
 return rootpi;
}

/*Routine che stampa tutto l'albero creato*/
void stampa_albero(listap *ppol)
{
 listav_ptr plv;
 listaf_ptr plf;
 int i,n;
 plv=ppol->lv;
 n=ppol->n;
 fprintf(f,"%d\n",n);
 for(i=1;i<=n+1;i++)
 {
  fprintf(f,"%lf %lf\n",plv->c->x,plv->c->y);
  plv=plv->next;
 }
 plf=ppol->lf;
 if(plf!= NULL)
 do
 {
  stampa_albero(plf->pol);
  plf=plf->next;
 }while(plf!=NULL);
 return ;
}

/*Aggiorna rootc (vecchie) con rootcn(nuove)*/
void aggiungi(listaf_ptr plf)
{
 listac_ptr  pcoo;
```

```
 int spia,i,nv;
 listaf_ptr pfappo;
 listav_ptr plv;
 pcoo=rootcn;
 do
 {
  spia=0;
  pfappo=plf;
  do
  {
   plv=pfappo->pol->lv;
   nv=pfappo->pol->n;
   for (i=1;i<=nv;i++)
   {
    if (plv->c==pcoo)
    {
     spia=1;
     purootc->next=pcoo;
     purootc=pcoo;
     break;
    }
    plv=plv->next;
   }
   if (spia==1) break;
   pfappo=pfappo->next;
  }
  while (pfappo!=NULL);
  pcoo=pcoo->next;
 }
 while (pcoo!=NULL);
 purootc->next=NULL;
 return ;
}

/*Legge il pol. iniziale da un file esterno*/
listaf_ptr leggidafile(listaf_ptr plf)
{
 int i;
 double x,y;
 listac_ptr appo;
 listav_ptr pvlist;
 if (plf==NULL)
 {
  plf=(listaf_ptr )malloc(sizeof(listaf));
  if(plf==NULL)
       printf("Errore nell' alloc.leggidafile\n");
  plf->pol=(listap *)malloc(sizeof(listap));
  if(plf->pol==NULL)
       printf("Errore nell' alloc. di leggidafile2\n");
  plf->pol->lv=NULL;
  plf->pol->lf=NULL;
```

```
  plf->next=NULL;
  if (fscanf(g,"%d\n",&(plf->pol->n))!=EOF) /*num. vert.*/
  {
   fscanf(g,"%d\n",&(plf->pol->oa));  /*CCW o ACW*/
   for(i=0;i<=((plf->pol->n)-1);i++)
   {
    fscanf(g,"%lf %lf\n",&x,&y);  /*coord. vertici*/
    rootc=inscoo(rootc,x,y,&appo);
    plf->pol->lv=insv(plf->pol->lv,appo);
   }
   purootc=appo;
   pvlist=cercapenultimo(plf->pol->lv,plf->pol->n);
   pvlist->next=plf->pol->lv;  /*Lista ciclica*/
   plf->next=leggidafile(plf->next);
  }
  else {free(plf); plf=NULL;}
 }
 return plf;
}

/*Accorpa due pti se sono a distanza < di d/2 o se i lati
che partono da essi hanno cross vicino a zero*/
listav_ptr aggiustapunti(listav_ptr plv,int *nv,double d)
{
 listav_ptr plv1,plv2,plv3;
 int i,n;
 double a1[2],a2[2],a3[2],a4[2],a5[2];
 double coseno;
 n=*nv;
 plv1=plv;
 plv2=plv1->next;
 plv3=plv2->next;
 for(i=1;i<=n;i++)
 {
  for(;;)
  {
  a1[0]=plv1->c->x;a1[1]=plv1->c->y;
  a2[0]=plv2->c->x;a2[1]=plv2->c->y;
  a3[0]=plv3->c->x;a3[1]=plv3->c->y;
  a4[0]=a1[0]-a2[0];a4[1]=a1[1]-a2[1];
  a5[0]=a3[0]-a2[0];a5[1]=a3[1]-a2[1];
  if((dot(a4,a4)< d*d/4. ||
     fabs((a2[0]-a1[0])*(a3[1]-a1[1])-(a3[0]-a1[0])
     *(a2[1]-a1[1]))<tol) && (*nv)!=0 )
  {
   (*nv)--;
   plv1->next=plv2->next;
   plv2=plv1;
   plv1=trovaprecedente(plv1,*nv);
  }
  else if(dot(a5,a5)>tol && (*nv!=0))
```

```
  {
   coseno=dot(a4,a5)/sqrt(dot(a4,a4)*dot(a5,a5));
   if(coseno>cos(0.11) || coseno<cos(acos(-1.)-0.11))
   {
    (*nv)--;
    plv1->next=plv2->next;
    plv2=plv1;
    plv1=trovaprecedente(plv1,*nv);
   }
   else break;
  }
  else  break;
 }
 plv1=plv1->next;
 plv2=plv2->next;
 plv3=plv3->next;
}
if((*nv)==0) (*nv)++;
return plv1;
}

/*Da' la suddivisione iniz. del pol. di partenza*/
void suddiniziale(double d,listaf_ptr pfiglio)
{
 listap *ppol;
 listav_ptr plv;
 int i,j,nsud,nvert;
 double a[2];
 double l,x,y;
 listac_ptr appo;
 ppol=pfiglio->pol;
 nvert=ppol->n;
 plv=ppol->lv;
 for(i=1;i<=nvert;i++)
 {
  a[0]=plv->c->x - plv->next->c->x;
  a[1]=plv->c->y - plv->next->c->y;
  l=sqrt(dot(a,a));
  plv->l=l;
  nsud=l/d +1;
  plv->nsud=nsud;
  /*Ora suddividiamo il lato*/
  for(j=0;j<=nsud-1;j++)
  if(rootpunti!=NULL)
  {
   x=plv->c->x+(plv->next->c->x-plv->c->x)*(double)j/
                                          (double)nsud;
   y=plv->c->y+(plv->next->c->y-plv->c->y)*(double)j/
                                          (double)nsud;
   purootpunti=inscoo(purootpunti,x,y,&appo);
   purootpunti=appo;
```

```
  }
  else
  {
   rootpunti=inscoo(rootpunti,plv->c->x,plv->c->y,&appo);
   purootpunti=rootpunti;
  }
  plv=plv->next;
 }
 return ;
}

/*Restituisce la distanza del lato del poligoni
interno che andiamo a costruire.
n   suddivisione del lato di partenza
l   lunghezza del lato di partenza
eps distanza di riferimento
h   parametro tra 0 e 1 a scelta*/
double distanza(int n,double l,double eps,double h)
{
 if (h<0. || h>1.0) return -1.;  /*Errore nell' input*/
 return h*eps*sqrt(3.0)/2.0+(1-h)*l*sqrt(3.0)/(2.0*n);
}

/*Trova il numero di punti desiderati*/
int trovanpd()
{
 return 1;
}

/*Restituisce il numero di suddivisioni di un lato
n   numero di suddivisioni desiderate
l   lunghezza del lato da dividere
eps distanza di riferimento*/
int nsuddivisioni(int n,double l,double eps,double h)
{
 if (h<0. || h>1.0) return -1.;  /*Errore nell' input*/
 return h*((int)(l/eps)+1)+(1-h)*n;
}

/*Stampa le coord. dei punti di una lista da plc
puntatore a listac*/
void stampalistac(listac_ptr plc)
{
 while (plc != NULL)
 {
  printf("x=%lf y=%lf",plc->x,plc->y);
  plc=plc->next;
 }
 return ;
}
```

```
/*Stampa tutti gli elementi di una lista di vertici*/
void stampalistav(listav_ptr plv,int n)
{
 int i;
 for(i=1;i<=n;i++)
 {
  printf("x=%lf y=%lf  ",plv->c->x,plv->c->y);
  plv=plv->next;
 }
 return ;
}

/*Funzione che cancella un elemento da listav*/
void cancella_listav(listav_ptr plv,int n)
{
 if(n==0 || plv==NULL) return ;
 else
 {
  listav_ptr p;
  p=plv->next;
  free(plv);
  cancella_listav(p,n-1);
 }
}

/*Funzione che cancella un elemento da listac*/
void cancella_listac(listac_ptr plc)
{
 if(plc==NULL) return ;
 else
 {
  listac_ptr p;
  p=plc->next;
  free(plc);
  cancella_listac(p);
 }
}

/*Funzione che cancella tutta una coda*/
void cancella_coda(coda_ptr pcoda)
{
 if(pcoda==NULL) return ;
 else
 {
  coda_ptr p;
  p=pcoda->next;
  free(pcoda);
  cancella_coda(p);
 }
}
```

```
/*Crea le sudd. dei lati dei poligoni interni*/
void suddsuccessive(listav_ptr plv,int nv,double d)
{
 listav_ptr appov;
 listac_ptr appo;
 int npd,nsud,i,j;
 double a[2],l,x,y;
 appov=plv;
 for(i=1;i<=nv;i++)
 {
  npd=trovanpd();
  a[0]=appov->c->x-appov->next->c->x;
  a[1]=appov->c->y-appov->next->c->y;
  l=sqrt(dot(a,a));
  appov->l=l;
  nsud=nsuddivisioni(npd,l,d,1.0);
  appov->nsud=nsud;
  for(j=0;j<=nsud-1;j++)
  {
   x=appov->c->x+(appov->next->c->x-appov->c->x)*(double)j/
                                                 (double)nsud;
   y=appov->c->y+(appov->next->c->y-appov->c->y)*(double)j/
                                                 (double)nsud;
   purootpunti=inscoo(purootpunti,x,y,&appo);
   purootpunti=appo;
  }
  appov=appov->next;
 }
}

/*Elimina uno dei due punti troppo vicini*/
void toglipuntivicini(listac_ptr plc,double d)
{
 listac_ptr plc1,plc2;
 while(plc!=NULL)
 {
  plc1=plc->next;
  plc2=plc;
  while(plc1!=NULL)
  {
   double dist;
   dist=(plc1->x-plc->x)*(plc1->x-plc->x)+
        (plc1->y-plc->y)*(plc1->y-plc->y);
   if(dist<=d*d/4.0)
   {
    if(plc1==purootpunti)
    {
     purootpunti=plc2;
    }
    plc2->next=plc1->next;
    free(plc1);
```

```
    plc1=plc2->next;
   }
   else
   {
    plc1=plc1->next;
    plc2=plc2->next;
   }
  }
  plc=plc->next;
 }
}

/*Determina se un poligono e ' convesso(1) o no(0)*/
int convesso(listav_ptr plv,int n)
{
 int spia,j;
 double det;
 listav_ptr pvj,pvk;
 spia=1;
 pvj=plv;
 for(j=1;j<=n;j++)
 {
  pvk=pvj->next->next;
  det=(pvj->next->c->x-pvj->c->x)*(pvk->c->y-pvj->c->y)-
      (pvk->c->x-pvj->c->x)*(pvj->next->c->y-pvj->c->y);
  if(det<0.0-tol)
  {
   spia=0;
   break;
  }
  pvj=pvj->next;
 }
 return spia;
}
```

Appendice B

Il programma MESH.C

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>

/*DEFINIZIONE DELLE STRUTTURE USATE*/
typedef struct listacoordinate * listac_ptr;
typedef struct listacoordinate
{
 int n; /*Label dei punti*/
 double x,y;
 listac_ptr next;
} listac;

typedef struct listalati * listal_ptr;
typedef struct listalati
{
 listal_ptr prev;
 listac_ptr c1,c2;
 listal_ptr next;
} listal;

typedef struct triangolo * listat_ptr;
typedef struct triangolo
{
 int n;              /*Label del triangolo*/
 listat_ptr prev;    /*Punt. al triangolo precedente*/
 listac_ptr c1,c2,c3;/*Punt. alle coord dei
                       tre vertici in senso antiorario*/
 listal_ptr l1,l2,l3;/*l1 punt. al lato da c2 a c3*/
                     /*l2 punt. al lato da c3 a c1*/
                     /*l3 punt. al lato da c1 a c2*/
 listat_ptr t1,t2,t3;/*ti punt. al triangolo che ha il lato li
                       in comune, e' NULL se li e' di bordo
                       nella mesh creata*/
 listat_ptr next;    /*Punt. al triangolo successivo*/
```

```
} listat;

typedef struct listalatitriangoli * listalt_ptr;
typedef struct listalatitriangoli
{
 listal_ptr l;
 listat_ptr t;
 listalt_ptr next;
}listalt;

/*DEFINIZIONE DELLE FUNZIONI USATE*/

/*Funzioni di lettura e stampa*/
listac_ptr leggidafile(FILE *,listac_ptr,int);
void stampalistac(listac_ptr);
void stampalati(listal_ptr);
void stampatri(FILE *,listat_ptr);
void stampaXgid(listac_ptr,listat_ptr);

/*Funzioni di utilità per le liste*/
listac_ptr inscoo(listac_ptr,double,double,listac_ptr *);
listat_ptr instri(listac_ptr,listac_ptr,listac_ptr,
            listal_ptr,listal_ptr,listal_ptr,listat_ptr,
            listat_ptr ,listat_ptr);
listal_ptr inslato(listac_ptr,listac_ptr);
listalt_ptr inslt(listalt_ptr,listal_ptr,listat_ptr);
void cancella_listalt(listalt_ptr);
listalt_ptr aggiustalv(listalt_ptr,listac_ptr);
listal_ptr latointri(listac_ptr,listac_ptr,listat_ptr);
void aggiornastatol(listal_ptr,listat_ptr,listat_ptr);
void aggiornatri(listalt_ptr,listac_ptr);
listalt_ptr toglidalv(listal_ptr,listalt_ptr);
void cancella_triangolo(listat_ptr);
void cancella_lato(listal_ptr);
listac_ptr trovaprecedente(listac_ptr,int);
listac_ptr cercavicino(listat_ptr,listac_ptr,listal_ptr,
                      int *);

/*Funzioni algebriche*/
double circumc(listac_ptr,listac_ptr,listac_ptr,double *,double *);
int iocerchio(listac_ptr,double,double,double);
int iotri(listat_ptr,listac_ptr,listal_ptr *);
void orientatri(listac_ptr *,listac_ptr *,listac_ptr *);
void ordinapti(listac_ptr);
int inout(listac_ptr,int,double pt[]);
int confronta(double pt[],listac_ptr,listac_ptr);
listac_ptr dminima(double pt[],double d1[],double d2[],
                  listac_ptr,int *,int  *, double *);
double cross(double,double,double,double);
double dot(listac_ptr,listac_ptr);
```

```
/*Funzioni principali*/
void primotri(listac_ptr);
void checkpoint(listac_ptr,listat_ptr,listalt_ptr *,
                listalt_ptr *,listat_ptr *,listal_ptr *);
void crea_triangoli(listac_ptr);
int visibile(listal_ptr,listat_ptr,listac_ptr);
void smooth(void);

/*VARIABILI GLOBALI*/
FILE *g,*f,*h1,*h,*l;
int stop;
int ntri=0;          /*label dei tri. nella mesh*/
int nvert;
double tol=1.0e-10;
listac_ptr rootc,rootcpol;
listat_ptr roottri; /*testa e coda della lista dei tri.*/
listal_ptr rootlati;/*Testa e coda della lista dei lati*/
listalt_ptr triout; /*Testa della lista dei tri. che cadono
                    fuori dal poligono iniziale*/

/*PROGRAMMA PRINCIPALE*/
main()
{
 int oa;
 listac_ptr appoc;
 listalt_ptr lt;
 listat_ptr tri;
 rootc=NULL;
 triout=NULL;
 g=fopen("punti.dat","r");
 f=fopen("lati.dat","wr");
 h1=fopen("tridopo.dat","wr");
 h=fopen("tri.dat","wr");
 l=fopen("poligono.in","r");
 rootcpol=NULL;
 fscanf(l,"%d\n",&nvert);
 fscanf(l,"%d\n",&oa);
 rootcpol=leggidafile(l,rootcpol,1);
 appoc=rootcpol;
 if (appoc==NULL) printf("Errore\n");
 while(appoc->next!=NULL)
 {
  appoc=appoc->next;
 }
 appoc->next=rootcpol; /*Lista chiusa*/
 rootc=leggidafile(g,rootc,1);
 ordinapti(rootc);
 stampalistac(rootc);
 scanf("%d",stop);
 roottri=NULL;
 rootlati=NULL;
```

```
 primotri(rootc);
 crea_triangoli(rootc->next->next->next);
 stampatri(h,roottri);
 printf("fatta prima stampa tri\n") ;
 tri=roottri;
 while(tri!=NULL)
 {
  listac_ptr c1,c2,c3;
  double bar[2];
  c1=tri->c1;
  c2=tri->c2;
  c3=tri->c3;
  bar[0]=(c1->x+c2->x+c3->x)/3.0;
  bar[1]=(c1->y+c2->y+c3->y)/3.0;
  if(inout(rootcpol,nvert,bar)==-1)
          triout=inslt(triout,NULL,tri);
  tri=tri->next;
 }
 lt=triout;
 while(lt!=NULL)
 {
  if((lt->t->t1) != NULL) aggiornastatol(lt->t->l1,lt->t->t1,NULL);
  else  cancella_lato(lt->t->l1);
  if((lt->t->t2) != NULL) aggiornastatol(lt->t->l2,lt->t->t2,NULL);
  else  cancella_lato(lt->t->l2);
  if((lt->t->t3) != NULL) aggiornastatol(lt->t->l3,lt->t->t3,NULL);
  else  cancella_lato(lt->t->l3);
  cancella_triangolo(lt->t);
  lt=lt->next;
 }
 smooth();
 stampalati(rootlati);
 stampatri(h1,roottri);
 stampaXgid(rootc,roottri);
 fclose(f);fclose(g);fclose(h);fclose(l);fclose(h1);
 return;
}

 /*Legge la lista di punti iniziale da un file esterno*/
listac_ptr leggidafile(FILE *file,listac_ptr plc,int n)
{
 if (plc==NULL)
 {
  plc=(listac_ptr )malloc(sizeof(listac));
  if(plc==NULL)
       printf("Errore nell' alloc.leggidafile\n");
  plc->n=n;
  plc->next=NULL;
  if (fscanf(file,"%lf %lf\n",&(plc->x),&(plc->y))!=EOF)
  {
   plc->next=leggidafile(file,plc->next,n+1);
```

```
  }
  else {free(plc);plc=NULL;}
 }
 return plc;
}

/*Stampa la lista delle coordinate*/
void stampalistac(listac_ptr plc)
{
 if(plc!=NULL)
 {
  printf("%d %lf %lf\n",plc->n,plc->x,plc->y);
  stampalistac(plc->next);
 }
 return;
}

/*Stampa la lista dei lati*/
void stampalati(listal_ptr pl)
{
 if(pl!=NULL)
 {
  fprintf(f,"%lf %lf %lf %lf\n",pl->c1->x,pl->c1->y,pl->c2->x,
                                pl->c2->y);
  stampalati(pl->next);
 }
 return;
}

/*Stampa la lista dei triangoli*/
void stampatri(FILE *file,listat_ptr pt)
{
 if(pt!=NULL)
 {
  fprintf(file,"%lf %lf %lf %lf %lf %lf\n",
  pt->c1->x,pt->c1->y,pt->c2->x,pt->c2->y,pt->c3->x,pt->c3->y);
  stampatri(file,pt->next);
 }
 return;
}

/*Inserisce le coordinate dei punti*/
listac_ptr inscoo(listac_ptr q,double x,double y ,listac_ptr *appo)
{
 if (q==NULL)
 {
  q=(listac_ptr)malloc(sizeof(listac)) ;
  if(q==NULL)
      printf("Errore in inscoo\n");
  *appo=q;
  q->x=x;
```

```
  q->y=y;
  q->next=NULL;
 }
 else q->next=inscoo(q->next,x,y,appo);
 return q;
}

/*Inserisce un triangolo*/
listat_ptr instri(listac_ptr c1,listac_ptr c2,
                  listac_ptr c3 ,listal_ptr l1,listal_ptr l2,
                  listal_ptr l3,listat_ptr t1,listat_ptr t2,
                  listat_ptr t3)
{
 listat_ptr q=NULL;
 q=(listat_ptr)malloc(sizeof(listat)) ;
 if(q==NULL)
   printf("Errore in instri\n");
 if(roottri!=NULL) roottri->prev=q;
 ntri++;
 q->n=ntri;
 q->c1=c1; q->c2=c2; q->c3=c3;
 q->l1=l1; q->l2=l2; q->l3=l3;
 q->t1=t1; q->t2=t2; q->t3=t3;
 q->next=roottri;
 q->prev=NULL;
 return q;
}

/*Inserisce un lato in testa alla lista dei lati*/
listal_ptr inslato(listac_ptr c1,listac_ptr c2)
{
 listal_ptr q;
 q=NULL;
 q=(listal_ptr)malloc(sizeof(listal));
 if(q==NULL)
   printf("Errore in inslato\n");
 if(rootlati!=NULL) rootlati->prev=q;
 q->c1=c1;
 q->c2=c2;
 q->next=rootlati;
 q->prev=NULL;
 return q;
}

/*Dati 3 punt. a coord. mi da' il quadrato del raggio del
circumcerchio e le coordinate del centro.
Torna -1 se i tre punti sono allineati.*/
double circumc(listac_ptr pc1,listac_ptr pc2,listac_ptr pc3,
               double *xc,double *yc)
{
 double det;
```

```
 det=(pc2->x-pc1->x)*(pc3->y-pc1->y)-(pc2->y-pc1->y)*
                                        (pc3->x-pc1->x);
 if(fabs(det)<=tol) return -1.0;
 *xc=((pc3->y-pc1->y)*(dot(pc1,pc1)-dot(pc2,pc2))-
     (pc2->y-pc1->y)*(dot(pc1,pc1)-dot(pc3,pc3)))/det;
 *yc=(-(pc3->x-pc1->x)*(dot(pc1,pc1)-dot(pc2,pc2))+
     (pc2->x-pc1->x)*(dot(pc1,pc1)-dot(pc3,pc3)))/det;
 *xc=*xc/(-2.0);
 *yc=*yc/(-2.0);
 return (pc1->x-*xc)*(pc1->x-*xc)+(pc1->y-*yc)*(pc1->y-*yc);
}

/*Calcola il prodotto scalare*/
double dot(listac_ptr pc1,listac_ptr pc2)
{
 return (pc1->x*pc2->x)+(pc1->y*pc2->y);
}

/*Vede se un punto e' dentro (1), sul bordo(0) o fuori(-1 ) dal
cerchio di centro xc,yc e raggio  quadro r2*/
int iocerchio(listac_ptr plc,double xc,double yc,double r2)
{
 double dist2;
 dist2=(plc->x-xc)*(plc->x-xc)+(plc->y-yc)*(plc->y-yc);
 if(dist2<=r2-tol) return 1;
 if(dist2>=r2+tol) return -1;
 return 0;
}

/*Dato un tri. mi dice se un punto e' dentro(1) sul bordo(0)
  o fuori(-1) dal triangolo stesso */
int iotri(listat_ptr tri,listac_ptr pc,listal_ptr *lext)
{
 listac_ptr pc1,pc2,pc3;
 double pv,pv3;
 pc1=tri->c1;
 pc2=tri->c2;
 pc3=tri->c3;
 *lext=NULL;
 /*Lato pc1 pc2*/
 pv=cross(pc2->x-pc1->x,pc2->y-pc1->y,pc->x-pc1->x,pc->y-pc1->y);
 if(fabs(pv)<=tol) {*lext=tri->l3;return 0;}
 else if(pv<tol)  return -1;
 /*Lato pc2 pc3*/
 pv=cross(pc3->x-pc2->x,pc3->y-pc2->y,pc->x-pc2->x,pc->y-pc2->y);
 if(fabs(pv)<=tol) {*lext=tri->l1;return 0;}
 else if(pv<tol)  return -1;
 /*Lato pc3 pc1*/
 pv=-cross(pc3->x-pc1->x,pc3->y-pc1->y,pc->x-pc1->x,pc->y-pc1->y);
 if(fabs(pv)<=tol) {*lext=tri->l2;return 0;}
 else if(pv<tol)  return -1;
```

```
 return 1;
}

/*Calcola il prodotto vettoriale*/
double cross(double x1,double y1,double x2,double y2)
{
 return (x1*y2)-(x2*y1);
}

/*Controlla se un triangolo è memorizzato in senso antiorario,
se non lo è lo modifica in modo che diventi tale*/
void orientatri(listac_ptr *c1,listac_ptr *c2,listac_ptr *c3)
{
 if(cross((*c2)->x-(*c1)->x,(*c2)->y-(*c1)->y,
          (*c3)->x-(*c1)->x,(*c3)->y-(*c1)->y)<0.0)
 {
  listac_ptr appo;
  appo=*c1;
  *c1=*c2;
  *c2=appo;
 }
 return;
}

/*Ordina per ogni poligono i punti in ordine lessicografico*/
void ordinapti(listac_ptr q)
{
 listac_ptr p;
 double x,y;
 while(q!=NULL)
 {
  p=q->next;
  while(p!=NULL)
  {
   if((p->x-q->x<-tol) ||
     ((p->x-q->x<tol) && (p->x-q->x>=-tol) && (p->y-q->y<-tol)))
   {
    x=q->x;    y=q->y;
    q->x=p->x; q->y=p->y;
    p->x=x;    p->y=y;
   }
   p=p->next;
  }
  q=q->next;
 }
 return;
}

/*Crea il primo tri. unendo i primi due elem. della lista dei punti
con il primo punto della lista che formi con i primi due un tri.
 di area non nulla*/
```

```
void primotri(listac_ptr q)
{
 listac_ptr c1,c2;
 listal_ptr l1,l2,l3;
 if(q==NULL) {printf("ERRORE1!!\n"); return;}
 c1=q;
 if(c1->next==NULL) {printf("ERRORE2!!\n"); return;}
 c2=q->next;
 if(c2->next==NULL) {printf("ERRORE3!!\n"); return;}
 q=c2->next;
 for(;;)
 {
  if(fabs(cross(c1->x-q->x,c1->y-q->y,c2->x-q->x,c2->y-q->y))>tol)
   break;
  q=q->next;
  if(q==NULL) break;
 }
 if(q==NULL) printf("ERRORE TUTTI I PUNTI SONO ALLINEATI!\n");
 else
 {
  double x,y;
  x=q->x;           y=q->y;
  q->x=c2->next->x; q->y=c2->next->y;
  c2->next->x=x;    c2->next->y=y;
 }
 c2=c1->next;
 q=c2->next;
 orientatri(&c1,&c2,&q);
 rootlati=inslato(c1,c2);
 l3=rootlati;
 rootlati=inslato(c2,q);
 l1=rootlati;
 rootlati=inslato(q,c1);
 l2=rootlati;
 roottri=instri(c1,c2,q,l1,l2,l3,NULL,NULL,NULL);
 return;
}

/*Inserisce un elemento in fondo alla coda di listalt*/
listalt_ptr inslt(listalt_ptr q,listal_ptr l,listat_ptr t)
{
 if(q==NULL)
 {
  q=(listalt_ptr)malloc(sizeof(listalt));
  if(q==NULL) printf("Errore di alloc.in inslt\n");
  q->l=l;
  q->t=t;
  q->next=NULL;
 }
 else q->next=inslt(q->next,l,t);
 return q;
```

```
}

/*Cancella tutta listalt*/
void cancella_listalt(listalt_ptr q)
{
 if(q==NULL) return;
 else
 {
  listalt_ptr p;
  p=q->next;
  free(q);
  cancella_listalt(p);
 }
 return;
}

/*Dato il punto c e il tri. t, se c e' dentro t  mette t in text,
altrimenti vede se il circumc. di t contiene c nel qual caso mette
t nella lista tc.In ogni caso se un lato di t e' di bordo nella
triangolazione lo inserisce nella lista lv.*/
void checkpoint(listac_ptr c,listat_ptr t,listalt_ptr *lv,
                listalt_ptr *tc,listat_ptr *text,listal_ptr *lext)
{
 double x,y,r2;
 int ioc,iot;
 listal_ptr lat;
 r2=circumc(t->c1,t->c2,t->c3,&x,&y);
 if(r2>tol)
 {
  ioc=iocerchio(c,x,y,r2);
  if(ioc==0 || ioc==1)
  if((*text)==NULL)
  {
   iot=iotri(t,c,&lat);
   if(iot==0 || iot==1) {*text=t;*lext=lat;}
   else *tc=inslt(*tc,NULL,t);
  }
  else
   *tc=inslt(*tc,NULL,t);/*Abbiamo aggiornato tc e text*/
 }
  /*Ora vediamo i lati per vedere quali sono di bordo*/
  /*Quindi aggiorniamo lv.*/
 if(t->t1==NULL) *lv=inslt(*lv,t->l1,t);
 if(t->t2==NULL) *lv=inslt(*lv,t->l2,t);
 if(t->t3==NULL) *lv=inslt(*lv,t->l3,t);
}

/*Crea tutti i triangoli a partire dal primo*/
void crea_triangoli(listac_ptr pc)
{
 listat_ptr text,ptri;
```

```
listalt_ptr tc,lv,lappo,tcappo,tcappo2,lvdel;
listal_ptr lext;
double r2,x,y;
int vis,io,n;
n=0;
while(pc!=NULL)
{
 n++;
 text=NULL;tc=NULL;lv=NULL;lext=NULL;
 ptri=roottri;
 while(ptri!=NULL)
 {
  checkpoint(pc,ptri,&lv,&tc,&text,&lext);
  ptri=ptri->next;
 }
 if(text!=NULL)
 {
  tc=inslt(tc,NULL,text);
  cancella_listalt(lv);
  lv=NULL;
  {
   tcappo=tc;
   lvdel=NULL;
   while(tcappo!=NULL)
   {
    /*Primo lato*/
    vis=visibile(tcappo->t->l1,tcappo->t,pc);
    if(vis==1) lvdel=inslt(lvdel,tcappo->t->l1,tcappo->t);
    else if(vis==0 && tcappo->t->l1!=lext)
    {
     if(tcappo->t->t1==NULL ) lv=inslt(lv,tcappo->t->l1,NULL);
     else
     {
      r2=circumc(tcappo->t->t1->c1,tcappo->t->t1->c2,
                 tcappo->t->t1->c3,&x,&y);
      io=iocerchio(pc,x,y,r2);
      if(io==-1) lv=inslt(lv,tcappo->t->l1,tcappo->t->t1);
     }
    }
    /*Secondo lato*/
    vis=visibile(tcappo->t->l2,tcappo->t,pc);
    if(vis==1) lvdel=inslt(lvdel,tcappo->t->l2,tcappo->t);
    else if(vis==0 && tcappo->t->l2!=lext)
    {
     if(tcappo->t->t2==NULL) lv=inslt(lv,tcappo->t->l2,NULL);
     else
     {
      r2=circumc(tcappo->t->t2->c1,tcappo->t->t2->c2,
                 tcappo->t->t2->c3,&x,&y);
      io=iocerchio(pc,x,y,r2);
      if(io==-1) lv=inslt(lv,tcappo->t->l2,tcappo->t->t2);
```

```
     }
    }
    /*Terzo lato*/
    vis=visibile(tcappo->t->l3,tcappo->t,pc);
    if(vis==1) lvdel=inslt(lvdel,tcappo->t->l3,tcappo->t);
    else if(vis==0  && tcappo->t->l3!=lext)
    {
     if(tcappo->t->t3==NULL) lv=inslt(lv,tcappo->t->l3,NULL);
     else
     {
      r2=circumc(tcappo->t->t3->c1,tcappo->t->t3->c2,
                 tcappo->t->t3->c3,&x,&y);
      io=iocerchio(pc,x,y,r2);
      if(io==-1) lv=inslt(lv,tcappo->t->l3,tcappo->t->t3);
     }
    }  /*Fine del controllo dei lati*/
    tcappo=tcappo->next;
   }/*FINE WHILE*/
  }/*fine tc==NULL*/
  /*Cancello i triangoli i cui circumc contengono pc*/
  tcappo=tc;
  while(tcappo!=NULL)
  {
   cancella_triangolo(tcappo->t);
   tcappo=tcappo->next;
  }
  /*Cancello i lati vis. dei tri. i cui circumc. contengono pc*/
  lappo=lvdel;
  while(lappo!=NULL)
  {
   cancella_lato(lappo->l);
   lappo=lappo->next;
  }
 if(lext!=NULL) cancella_lato(lext);
 aggiornatri(lv,pc);
}
else if(tc!=NULL ) /*Pto fuori dalla mesh ma in qualche circumc.*/
{
 lv=aggiustalv(lv,pc);
 tcappo=tc;
 lvdel=NULL;
 while(tcappo!=NULL)
 {
  if(tcappo->t->t1==NULL) lv=toglidalv(tcappo->t->l1,lv);
  if(tcappo->t->t2==NULL) lv=toglidalv(tcappo->t->l2,lv);
  if(tcappo->t->t3==NULL) lv=toglidalv(tcappo->t->l3,lv);
  /*Primo lato*/
  vis=visibile(tcappo->t->l1,tcappo->t,pc);
  if(vis==1) lvdel=inslt(lvdel,tcappo->t->l1,tcappo->t);
  else if(vis==0)
  {
```

```
   if(tcappo->t->t1==NULL) lv=inslt(lv,tcappo->t->l1,NULL);
   else
   {
    r2=circumc(tcappo->t->t1->c1,tcappo->t->t1->c2,
               tcappo->t->t1->c3,&x,&y);
    io=iocerchio(pc,x,y,r2);
    if(io==-1) lv=inslt(lv,tcappo->t->l1,tcappo->t->t1);
   }
  }
  /*Secondo lato*/
  vis=visibile(tcappo->t->l2,tcappo->t,pc);
  if(vis==1) lvdel=inslt(lvdel,tcappo->t->l2,tcappo->t);
  else if(vis==0)
  {
   if(tcappo->t->t2==NULL) lv=inslt(lv,tcappo->t->l2,NULL);
   else
   {
    r2=circumc(tcappo->t->t2->c1,tcappo->t->t2->c2,
               tcappo->t->t2->c3,&x,&y);
    io=iocerchio(pc,x,y,r2);
    if(io==-1) lv=inslt(lv,tcappo->t->l2,tcappo->t->t2);
   }
  }
  /*Terzo lato*/
  vis=visibile(tcappo->t->l3,tcappo->t,pc);
  if(vis==1) lvdel=inslt(lvdel,tcappo->t->l3,tcappo->t);
  else if(vis==0)
  {
   if(tcappo->t->t3==NULL) lv=inslt(lv,tcappo->t->l3,NULL);
   else
   {
    r2=circumc(tcappo->t->t3->c1,tcappo->t->t3->c2,
               tcappo->t->t3->c3,&x,&y);
    io=iocerchio(pc,x,y,r2);
    if(io==-1) lv=inslt(lv,tcappo->t->l3,tcappo->t->t3);
   }
  }  /*Fine controllo lati*/
  tcappo=tcappo->next;
 }/*FINE WHILE*/
 /*Cancello i triangoli i cui circumc contengono pc*/
 tcappo=tc;
 while(tcappo!=NULL)
 {
  cancella_triangolo(tcappo->t);
  tcappo=tcappo->next;
 }
 /*Cancello i lati vis. dei tri. i cui circumc contengono pc*/
 lappo=lvdel;
 while(lappo!=NULL)
 {
  cancella_lato(lappo->l);
```

```
   lappo=lappo->next;
  }
  aggiornatri(lv,pc);
 }
 else /*Pto fuori dall triangolazione e dai circumcerchi*/
 {
  lv=aggiustalv(lv,pc);
  aggiornatri(lv,pc);
 }
 pc=pc->next;
 cancella_listalt(lv);
 cancella_listalt(tc);
}
}

/*Aggiusta la lista dei lati visibili se pc non e' in nessun
circumcerchio della triangolazione*/
listalt_ptr aggiustalv(listalt_ptr lv,listac_ptr pc)
{
 int vis;
 listalt_ptr plv ,preplv;
 if(lv==NULL) return lv;
 plv=lv; preplv=NULL;
 while(plv!=NULL)
 {
  vis=visibile(plv->l,plv->t,pc);
  if(vis==0)
  {
   if(plv == lv) {lv=lv->next;free(plv);plv=lv;}
   else {preplv->next=plv->next; free(plv);plv=preplv->next;}
   /*L' ho tolto*/
  }
  else {preplv=plv; plv=plv->next;}
 }
 return lv;
}

/*Vede se il lato l del triangolo tri e' visibile o no
dal pto pc.
1- visibile
0- non visibile
-1 -l NULL*/
int visibile(listal_ptr l,listat_ptr tri,listac_ptr pc)
{
 listac_ptr c1,c2;
 double pvet;
 if(l!=NULL)
 {
  if(tri->l1==l)  {c1=tri->c2;c2=tri->c3;}
  else  if(tri->l2==l)  {c1=tri->c3;c2=tri->c1;}
  else  if(tri->l3==l)  {c1=tri->c1;c2=tri->c2;}
```

```
  else return -1;
  pvet=cross(pc->x-c1->x,pc->y-c1->y,c2->x-c1->x,c2->y-c1->y);
  if(pvet>=tol) return 1;
  else return 0;
 }
 return -1;
}

/*Data la lista di lati con cui unire pc,crea i nuovi tri.*/
void aggiornatri(listalt_ptr lv,listac_ptr pc)
{
 listat_ptr tri,t1,t2,t3;
 listac_ptr c1,c2,c3;
 int i,ntn=0;/*Numero di nuovi triangoli*/
 listal_ptr l,l1,l2,l3;
 c3=pc;
 while(lv!=NULL)
 {
  c1=lv->l->c1;
  c2=lv->l->c2;
  orientatri(&c1,&c2,&c3);
  if(lv->l==NULL) printf("ERRORE\n");
  l3=lv->l;
  t3=lv->t;
  l1=NULL; l2=NULL;
  /*Cerchiamo se l1 ed l2 esistono gia' come lati*/
  tri=roottri;
  for(i=1;i<=ntn;i++)
  {
   l=latointri(c2,c3,tri);
   if(l!=NULL) {l1=l;t1=tri;}
   l=latointri(c1,c3,tri);
   if(l!=NULL) {l2=l;t2=tri;}
   tri=tri->next;
  }
  if (l1==NULL)
  {
   rootlati=inslato(c2,c3);
   l1=rootlati;
   t1=NULL;
  }
  if(l2==NULL)
  {
   rootlati=inslato(c3,c1);
   l2=rootlati;
   t2=NULL;
  }
  roottri=instri(c1,c2,c3,l1,l2,l3,t1,t2,t3);
  ntn++;
  aggiornastatol(l1,t1,roottri);
  aggiornastatol(l2,t2,roottri);
```

```
  aggiornastatol(l3,t3,roottri);
  lv=lv->next;
 }
 return;
}

/*Da' il punt. al lato l del tri. t con estremi
c1 e c2 ,NULL se non trova niente.*/
listal_ptr latointri(listac_ptr c1,listac_ptr c2,listat_ptr t)
{
 listac_ptr vc[3];
 int i,j;
 vc[0]=t->c1;vc[1]=t->c2;vc[2]=t->c3;
 for (i=0;i<=2;i++)
 {
  for (j=0;j<=2;j++)
  {
   if(i!=j)
   {
    if(c1==vc[i] && c2==vc[j])
    {
     if ((i+j)==1) return t->l3;
     if ((i+j)==2) return t->l2;
     if ((i+j)==3) return t->l1;
    }
   }
  }
 }
 return NULL;
}

/*Dato il lato l di tri inserisce che l sta anche in altri*/
void aggiornastatol(listal_ptr l,listat_ptr tri,listat_ptr altri)
{
 if(tri!=NULL)
 {
  if(tri->l1==l) {tri->t1=altri;return;}
  if(tri->l2==l) {tri->t2=altri;return;}
  if(tri->l3==l) {tri->t3=altri;return;}
 }
}

/*Toglie da lv il lato l se c'e'*/
listalt_ptr toglidalv(listal_ptr l,listalt_ptr lv)
{
 listalt_ptr lvpre,lvappo;
 lvappo=lv;
 while(lvappo!=NULL)
 {
  if(lvappo->l==l)
  {
```

```
   if(lvappo==lv) {lv=lv->next;free(lvappo);return lv;}
   else {lvpre->next=lvappo->next;free(lvappo);return lv;}
  }
  lvpre=lvappo;
  lvappo=lvappo->next;
 }
 return lv;
}

/*Cancella un lato dalla lista dei lati*/
void cancella_lato(listal_ptr pl)
{
 listal_ptr lappo;
 if(pl!=NULL)
 {
  if(pl->prev==NULL)
  {
   rootlati=rootlati->next;
   if(rootlati!=NULL) rootlati->prev=NULL;
   free(pl);
   return;
  }
  else
  {
   pl->prev->next=pl->next;
   if(pl->next!=NULL) pl->next->prev=pl->prev;
   free(pl);
   return;
  }
 }
 return;
}

/*Cancella un triangolo dalla lista dei triangoli*/
void cancella_triangolo(listat_ptr pt)
{
 listat_ptr tappo;
 if(pt!=NULL)
 {
  if(pt->prev==NULL)
  {
   roottri=roottri->next;
   if(roottri!=NULL) roottri->prev=NULL;
   return;
  }
  else
  {
   pt->prev->next=pt->next;
   if(pt->next!=NULL) pt->next->prev=pt->prev;
   return;
  }
```

```
 }
 return;
}

/*Stampa per il software GID*/
void stampaXgid(listac_ptr plc,listat_ptr plt)
{
 FILE *fg;
 int cont=1;
 fg=fopen("meshn.msh","w");
 fprintf(fg,"MESH    dimension = 3 ElemType Triangle  Nnode = 3\n");
 fprintf(fg,"Coordinates\n");
 while(plc!=NULL)
 {
  fprintf(fg,"%d %lf %lf %lf\n",plc->n,plc->x,plc->y,0.0);
  plc=plc->next;
 }
 fprintf(fg,"end coordinates\n");
 fprintf(fg,"\n");
 fprintf(fg,"Elements\n");
 while(plt!=NULL)
 {
  fprintf(fg,"%d %d %d %d\n",cont,plt->c1->n,plt->c2->n,plt->c3->n);
  plt=plt->next;
  cont++;
 }
 fprintf(fg,"end elements\n");
 fclose(fg);
 return;
}
/* Vede se pt è fuori o dentro il pol. di cui fare la mesh.
Parametri di Input:
        pt[2] -  punto da indagare

        Valori di ritorno:
        INOUT - -2 Errore in input
                -1 Pto esterno
                 0 Pto sul bordo
                +1 Pto interno
                +2 Pto vertice
*/
int inout(listac_ptr pappo,int n,double pt[2])
{
 double  c1,c2,d1[2],d2[2],d[2],a1[2],a2[2],b[2];
 double dmin;
 int i,j,l,l1,l2,io,vsn;
 listac_ptr ph,ph1,ph2;
 io=-2;
 /*Cerca il punto sul bordo del pol. piu' vicino a pt */
 dmin = 1.0e+10;
 vsn=0;  /*il punto a distanza minima vsn=0
```

```
          non e' un vertice vsn=1 e' un vertice */
ph=NULL;
for(i=0;i<=n-1;i++)
{
 d1[0]=pappo->x;
 d1[1]=pappo->y;
 d2[0]=(pappo->next)->x;
 d2[1]=(pappo->next)->y;
 ph1=dminima(pt,d1,d2,pappo,&vsn,&io,&dmin);
 if(io==0 || io==2) return io;
 pappo=pappo->next;
 if(ph1!=NULL) ph=ph1;
}
if (dmin<1.e+10)
{
 if (vsn==0)
 {
  io=confronta(pt,ph,ph->next);
  return io;
 }
 else
 {
  ph1=ph->next;
  ph2=trovaprecedente(ph,n);
  d1[0]=ph1->x;
  d1[1]=ph1->y;
  d2[0]=ph2->x;
  d2[1]=ph2->y;
  d[0]=ph->x;
  d[1]=ph->y;
  if ((d1[1]-pt[1])*(d2[1]-pt[1])<-1.e-10)
  {
   io=confronta(pt,ph,ph1);
   return io;
  }
  if (fabs(d1[1]-pt[1])<1.e-10)
  {
   io=confronta(pt,ph2,ph);
   return io;
  }
  if (fabs(d2[1]-pt[1])<1.e-10)
  {
   io=confronta(pt,ph,ph1);
   return io;
  }
  if ((d1[1]-pt[1])*(d2[1]-pt[1])>1.e-10)
  {
   for(j=0;j<=1;j++)
   {
    a1[j]=d1[j]-d[j];
    a2[j]=d2[j]-d[j];
```

```
     b[j]=pt[j]-d[j];
    }
    c1=(a1[0]*b[0]+a1[1]*b[1])/sqrt((a1[0]*a1[0]+a1[1]*a1[1])*
                                (b[0]*b[0]+b[1]*b[1]));
    c2=(a2[0]*b[0]+a2[1]*b[1])/sqrt((a2[0]*a2[0]+a2[1]*a2[1])*
                                (b[0]*b[0]+b[1]*b[1]));
    if (c1>c2)
    {
     io=confronta(pt,ph,ph1);
     return io;
    }
    else
    {
     io=confronta(pt,ph2,ph);
     return io;
    }
   }
  }
 }
 else
 return -1;
 return -2;
}

/*Calcola il punto a distanza minima da un lato (per inout)*/
listac_ptr dminima(double pt[2],double d1[2],double d2[2],
                   listac_ptr pi,int *vsn,int  *io, double *dmin)
{
 double dist,t,xint;
 listac_ptr ph;
 ph=NULL;
 if (fabs(d2[1]-d1[1]) <= tol)
 {
  if (fabs(d2[1]-pt[1]) <= tol)
  {
   if ((d1[0]+tol<=pt[0] && pt[0]<=d2[0]-tol) ||
      (d2[0]<pt[0] && pt[0]<d1[0])) {*io=0;return ph;}
   if (fabs(d1[0]-pt[0])<=tol ||
       fabs(d2[0]-pt[0])<=tol) {*io=2;return ph;}
   if ( d1[0]<pt[0])
   {
    dist=pt[0]-d1[0];
    if (dist< *dmin)
    {
     *vsn=1;
     *dmin=dist;
     ph=pi;
    }
   }
  }
 }
```

```
 else
 {
  t=(pt[1]-d1[1])/(d2[1]-d1[1]);
  if (t>=-tol && t<1.-tol)
  {
   xint=(1-t)*d1[0]+t*d2[0];
   if (fabs(xint-pt[0])<=tol)
   {
    if (t>-tol) {*io=0;return ph;}
    else {*io=2; return ph;}
   }
   else if (xint<pt[0])
   {
    dist=pt[0]-xint;
    if (dist< *dmin)
    if (t>0.)
    {
     *vsn=0;
     *dmin=dist;
     ph=pi;
    }
    else
    {
     *vsn=1;
     *dmin=dist;
     ph=pi;
    }
   }
  }
 }
 *io=3;
 return ph;
}

/*
Parametri di Input:
pt -  punto da indagare
Valori di ritorno:
INOUT - -1 Pto esterno
    +1 Pto interno
*/
int confronta(double pt[2],listac_ptr pi1,listac_ptr pi2)
{
 double  d1[3],d2[3],det;
 int j;

 d1[0]=pi1->x-pt[0];
 d1[1]=pi1->y-pt[1];
 d2[0]=pi2->x-pt[0];
 d2[1]=pi2->y-pt[1];
 det=d1[0]*d2[1]-d1[1]*d2[0];
```

```
 if (det > 0.+tol) return 1;
 else return -1;
}

/*Cerca il punt. precedente a pl in una lista ciclica lunga n*/
listac_ptr trovaprecedente(listac_ptr pl,int n)
{
 int i;
 listac_ptr pappo;
 pappo=pl;
 for(i=1;i<=n-1;i++) pappo=pappo->next;
 return pappo;
}

/*Calcola quali dei triangoli creati abbiano angoli piccoli e
per la boccia di quel punto applica l'ottimizzazione*/
void smooth()
{
 double x,y,coseno,cosmax;
 listat_ptr tri,triin,ptri;
 listac_ptr pc,pcn;
 listal_ptr l;
 int spia,ind,n;
 tri=roottri;
 while(tri!=NULL)
 {
  cosmax=0.0;
  coseno=((tri->c2->x-tri->c1->x)*(tri->c3->x-tri->c1->x)+
         (tri->c2->y-tri->c1->y)*(tri->c3->y-tri->c1->y))/
         sqrt(((tri->c2->x-tri->c1->x)*(tri->c2->x-tri->c1->x)+
         (tri->c2->y-tri->c1->y)*(tri->c2->y-tri->c1->y))*
         ((tri->c3->x-tri->c1->x)
         *(tri->c3->x-tri->c1->x)+
         (tri->c3->y-tri->c1->y)*(tri->c3->y-tri->c1->y)));
  if(coseno>cosmax) {cosmax=coseno;pc=tri->c1;}
  coseno=((tri->c1->x-tri->c2->x)*(tri->c3->x-tri->c2->x)+
         (tri->c1->y-tri->c2->y)*(tri->c3->y-tri->c2->y))/
         sqrt(((tri->c1->x-tri->c2->x)*(tri->c1->x-tri->c2->x)+
         (tri->c1->y-tri->c2->y)*(tri->c1->y-tri->c2->y))*
         ((tri->c3->x-tri->c2->x)
         *(tri->c3->x-tri->c2->x)+
         (tri->c3->y-tri->c2->y)*(tri->c3->y-tri->c2->y)));
  if(coseno>cosmax) {cosmax=coseno;pc=tri->c2;}
  coseno=((tri->c2->x-tri->c3->x)*(tri->c1->x-tri->c3->x)+
         (tri->c2->y-tri->c3->y)*(tri->c1->y-tri->c3->y))/
          sqrt(((tri->c2->x-tri->c3->x)*(tri->c2->x-tri->c1->x)+
         (tri->c2->y-tri->c3->y)*(tri->c2->y-tri->c3->y))*
         ((tri->c1->x-tri->c3->x)
         *(tri->c1->x-tri->c3->x)+
         (tri->c1->y-tri->c3->y)*(tri->c1->y-tri->c3->y)));
  if(coseno>cosmax) {cosmax=coseno;pc=tri->c3;}
```

```
  if(cosmax>sqrt(3.0)/2.0)
  {
   triin=tri;
   ptri=tri;
   l=NULL;
   n=0;
   x=y=0.0;
   for(;;)
   {
    n++;
    pcn=cercavicino(ptri,pc,l,&ind);
    x=x+pcn->x; y=y+pcn->y;
    if(ind==1)  {l=ptri->l1; ptri=ptri->t1;}
    else if(ind==2)  {l=ptri->l2; ptri=ptri->t2;}
    else  {l=ptri->l3; ptri=ptri->t3;}
    if(ptri==NULL) {spia=0;break;}
    if(ptri==triin) {spia=1;break;}
   }
   if(spia==1) {pc->x=x/(double)n;pc->y=y/(double)n;}
  }
  tri=tri->next;
 }
 return;
}

/*Cerca il numero del lato del triangolo limitrofo a tri
che ha in comune il lato l*/
listac_ptr cercavicino(listat_ptr tri,listac_ptr c,
                       listal_ptr l,int *ind)
{
 if(tri->c1==c)
 {
  if(tri->l2==l) {*ind=3;return tri->c2;}
  else {*ind=2;return tri->c3;}
 }
 else if(tri->c2==c)
 {
  if(tri->l1==l) {*ind=3;return tri->c1;}
  else {*ind=1;return tri->c3;}
 }
 else if(tri->c3==c)
 {
  if(tri->l2==l) {*ind=1;return tri->c2;}
  else {*ind=2;return tri->c1;}
 }
 else printf("Errorein cerca vicino\n") ;
 return NULL;
}
```

Appendice C

Listati per Mathematica

Riportiamo quindi seguito il programma, scritto, per il software Mathematica, che è stato utilizzato per la visualizzazione delle mesh.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Acquisizione dei dati dal file costruito dal programma
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

b=ReadList["d:/compiler/lati.dat",Number];

%%%%%%%%%%%%%%%%%%%%%%%
Riassemblaggio dei dati
%%%%%%%%%%%%%%%%%%%%%%%

d=Dimensions[b]
c=Table[Line[{{b[[4*i+1]],b[[4*i+2]]},{b[[4*i+3]],b[[4*i+4]]}}]
       ,{i,0,d[[1]]/4-1}];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Linee di visualizzazione del grafico
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Show[Graphics[c],PlotRange->
    {Automatic,Automatic},Axes->False,AspectRatio->Automatic]
```

Ora, invece, riportiamo il programma utilizzato per la visualizzazione dei poligoni.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Acquisizione dei dati dal file costruito dal programma
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

b=ReadList["d:/compiler/albero.dat",Number];

%%%%%%%%%%%%%%%%%%%%%%%%
Riassemblaggio dei dati
%%%%%%%%%%%%%%%%%%%%%%%%

Clear[v,graf] v=Array[graf,Floor[Dimensions[b]/6]];
dim=Floor[Dimensions[b]/6]
cont=1
For[i=1,i<dim[[1]],i++,m=(b[[cont]]+1)*2+cont;
       graf[i]=Line[Partition[Take[b,{cont+1,m}],2]];
       cont=m+1;If [cont>Dimensions[b][[1]],Break[]]]
Clear[w]
w=Drop[v,{i+1,dim[[1]]}];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Linee di visualizzazione del grafico
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Show[Graphics[w,AspectRatio->Automatic,Axes->True,PlotRange->
{Automatic,Automatic}]]
```

Ringraziamenti

Il mio ringraziamento più grande va ai miei genitori che mi hanno permesso di arrivare a questo traguardo, grazie per la fiducia e la pazienza.
Vorrei anche cogliere l'occasione per ringraziare tutti i miei amici, quelli che avevo e quelli che ho conosciuto in quest'ultimo periodo di lavoro, in particolare: Andrea, Barbara, Patrizia, Sonia, Gianni, Checco e Fabio. Spero di non aver dimenticato nessuno.
Ma il mio ringraziamento maggiore va a Francesco "Bebo" Mancini ottimo amico e bravissimo informatico, non credo che senza il suo aiuto avrei potuto completare il mio lavoro. Un ringraziamento lo devo anche ai miei relatori, a Nadaniela Egidi e al Prof. Rosario Culmone per le capacità e l'esperienza che hanno sempre messo a mia disposizione.
E per ultimo, ma non perchè meno importante, vorrei ringraziare il mio Lorenzo per il suo amore, la sua pazienza e il suo grande incoraggiamento.

Bibliografia

[1] P.L.George, H.Borouchaki. *Delaunay Triangulation and Meshing. Application to Finite Elements*. Hermes, 1998.

[2] P.J.Frey,P.L.George. *Mesh Generation. Application to Finite Elements*. Hermes Science, 2000.

[3] M. Bern, D. Eppstein. *Mesh Generation and Optimal Triangulation*.In *Computing in Euclidean Geometry*, F. Hwang e D. Du, World Scientific 1995.

[4] M. Bern, P. Chew,D. Eppstein,J. Ruppert. *Dihedral Bounds for Mesh Generation in High Dimensions*. Proceedings of the 6th Annual Symposium on Discrete Algorithms. San Francisco, CA, USA, 1995. (Pag 189-196).

[5] S. Fortune. *Voronoy Diagrams and Delaunay Triangulations*. In *Computing in Euclidean Geometry*, F. Hwang e D. Du, World Scientific 1995.

[6] M. G. Coutinho Neto. *Computational Geometry Implementation of Topological Robust Control*. A dissertation presented to the Faculty of the Graduate School University of Southern California. 1996.
URL:www.isi.edu/people/coutinho/SimplicialVIEW/main.html

[7] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD Thesis, School of Computer Science, Carnagie mellon University, Pittsburg. 1997. URL:http://fano.ics.uci.edu/cites/Author/Jonathan-Richard-Shewchuk.html

[8] J. R. Shewchuk. *Tetrahedral Mesh Generation by Delaunay Refinement*. Proceedings of the Fourteenth Annual Symposium on Computational Geometry (Minneapolis, Minnesota), (Pag 86-95), Association for Computing Machinery, 1998.
URL:http://www.cs.cmu.edu/ jrs/jrspapers.html

[9] E. P. Mucke. *Shapes and Implementations in three-dimensional Geometry*. PhD Thesis. Technical report UIUCDCS-R-93-1836. Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.
URL:http://www.geom.umn.edu/ mucke/GeomDir/thesis93.html

[10] E. P. Mucke. *A Robust Implementation for three-dimensional Delaunay Triangulations*. Proceedings of the 1st International Computational Geometry Software Workshop, 1995.
URL:http://www.geom.umn.edu/ mucke/GeomDir/detri95.html

[11] B. Joe. *Three-dimensional triangulations from local trasformations*. SIAM Journal On Scientific Statistical Computing, 10(4):718-741, 1989.

[12] B. Joe. GEOMPACK. *A software package for the generation of meshes using geometric algorithms*. Advances in Engineering software and Workstations Computing. 47(1):43-49, 1991.

[13] M. Bern, P. Plassmann *Mesh Generation.* Handbook of Computational Geometry. Elsevier Science.

[14] S. J. Owen *A Survey of Unstructured Mesh Generation Technology.* URL:www.andrew.cmu.edu/user/sowen/survey/index.html

[15] M. Filipiak. *Mesh Generation.* Technology Watch Report. Edinburgh Parallel Computing Centre. Novembre 1996. URL:http://www-users.informatik.rwth-aachen.de/roberts/literature.html

[16] E. Seveno. *Towards an adaptive advancing front method.* INRIA, Project Gamma. Proceedings, 6th International Meshing Roundtable, Sandia National Laboratories, pp.349-360, Ottobre 1997. URL:www.andrew.cmu.edu/user/sowen/imr6.html

[17] P. J. Frey, H. Borouchaki, P. L. George. *Delaunay Tetrahedralization using an Advancing-Front Approach.* INRIA, Project Gamma. Proceedings, 5th International Meshing Roundtable, Sandia National Laboratories, pp.31-46, Ottobre 1996. URL:www.andrew.cmu.edu/user/sowen/abstracts/Fr241.html

[18] P. Fleischmann, S. Selberherr. *Three-Dimensinal Delaunay Mesh Generation Using a Modified Advancing Front Approach.* Proceedings, 6th International Meshing Roundtable, Sandia National Laboratories, pp.267-278, Ottobre 1997. URL:www.andrew.cmu.edu/user/sowen/imr6.html

[19] H. Borouchaki, F. Hecht, E. Saltel, P. L. George. *Reasonably efficient Delaunay based mesh generator in 3 dimensions*. Proceedings, 4th International Meshing Roundtable, pages 3–14. Sandia National Laboratories, Agosto 1995.
URL:http://citeseer.nj.nec.com/borouchaki95reasonably.html

[20] F. Aurenhammer, R. Klein. *Voronoi Diagrams*. Chapter 18, Textbook on Computational Geometry, J.Sack and G. Urrutia (eds). 1998.

[21] GID - A universal, adaptive and user-friendly graphical user interface for geometrical modelling, data input and visualisation of results for all types of numerical simulation programs.
URL:http://gid.cimne.upc.es/intro/index.html

[22] R. Pennesi. *Modellatore solido per la Generazione e la Visualizzazione di Mesh*. Tesi Sperimentale di Laurea in Calcoli Numerici e Grafici. Dipartimento di Matematica e Fisica. Università di Camerino. A.A. 1999/2000