Analisi di un DFN

Progetto di Gruppo

Programmazione e Calcolo Scientifico

Politecnico di Torino

Stefano Caprioli
Enrico Micalizio
Alberto Prino

Anno Accademico 2023/2024

1 Introduzione

In questo elaborato esamineremo un DFN (Discrete Fracture Network) prendendo in analisi alcuni file contenenti i vertici di un determinato numero di fratture.
Il progetto è costituito da due parti principali:

1. Determinazione delle tracce generate dalle intersezioni tra le fratture;
2. Determinazione dei sottopoligoni ottenuti tagliando le fratture lungo le tracce trovate nella parte 1.

2 Parte 1: Determinazione delle Tracce

In questa sezione descriveremo il processo di identificazione delle tracce per ciascuna frattura nel DFN. Differenzieremo tra tracce passanti e non-passanti e ordineremo questi sottoinsiemi per lunghezza in ordine decrescente.

2.1 Le structures

Per la memorizzazione, abbiamo scelto le seguenti strutture dati.

- **struct Fractures**: struct contenente tutte le fratture presenti nel file di input. In essa viene salvato il numero totale di fratture. Inoltre, per ogni frattura abbiamo: le coordinate dei punti sottoforma di vector di Vector3d, il rispettivo id, il numero di vertici, la normale al piano contenente la frattura e la sfera associata generata da calcsphere.
- **struct Traces**: struct contenente tutte le tracce del DFN. Oltre al numero di tracce abbiamo, per ogni traccia: il rispettivo id, la coppia di ids di fratture che la generano, la sua lunghezza e il flag "passante" (traccia inserita nel vector di array di ids $TipsTrue$) oppure "non passante" (traccia inserita nel vector di array di ids $TipsFalse$).

All'interno della struct $Fractures$ la scelta dei container è spesso ricaduta sui vectors. In particolare, data la necessità di accedere in modo casuale agli elementi, e non conoscendo a priori le dimensioni del DFN (in termini di numero di vertici per fratture e numero di fratture stesse), è risultato ottimale identificare ogni frattura come un vector di Vector3d (coordinate di ogni vertice della frattura). Ogni frattura è stata inserita, a sua volta, all'interno di un vector (di ids e di vector di Vector3d).
Per quanto riguarda, invece, le tracce, essendo ogni traccia univocamente definita da due punti è stato possibile servirsi degli arrays (dato che la compilazione è nota a tempo di compilazione). Le tracce sono state salvate in vector (poichè il numero di tracce non è noto a priori) di arrays di Vector3d (per salvare le coordinate degli estremi) e di ids di fratture (per associare ogni traccia alle due fratture che la generano).

2.2 Metodologie e Algoritmi Utilizzati

La prima parte ha richiesto l'utilizzo di un gran numero di algoritmi, che abbiamo diviso in tre namespace.

Sommariamente, gli step che abbiamo seguito sono:

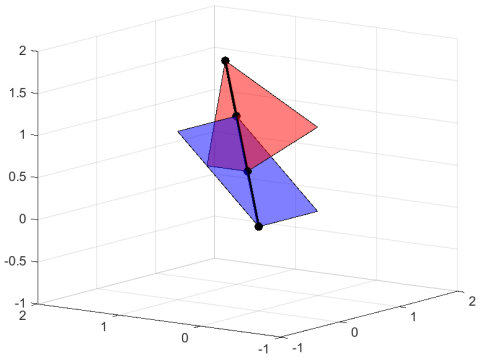
- scrematura delle fratture parallele o troppo distanti fra loro. Questo evita che tali fratture vengano prese in considerazione per la generazione di una traccia, diminuendo considerevolmente il tempo richiesto per la compilazione.

- identificazione delle tracce. Dopo aver trovato la retta generata dall'intersezione tra i due piani e averla fatta intersecare con i lati delle fratture in questione, abbiamo costruito un algoritmo in grado di constatare se un dato punto appartenga o meno a un certo poligono. In questo modo abbiamo trovato gli estremi delle tracce, che abbiamo memorizzato nell'apposita struct.

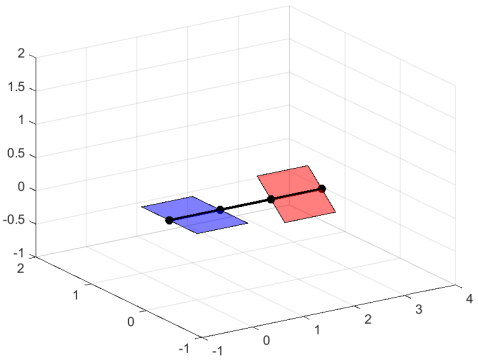
Entriamo adesso più nel dettaglio, descrivendo le principali caratteristiche di ogni algoritmo.

- **Analytics**
  Sono algoritmi puramente matematici, che sfruttano elementi di geometria computazionale e di algebra lineare essenziali per il progetto. In particolare, abbiamo:

  - **Vector4d calcsphere(const vector<Vector3d>& vertexdata)**: riceve in input un vector di Vector3d contenente i vertici di una generica frattura $\mathcal{F}$. Calcola il baricentro $B \in \mathbb{R}^3$ della frattura come media delle posizioni dei vertici e, successivamente, calcola, per ogni vertice $V \in \mathcal{F}$, la distanza d dal baricentro. La funzione restituisce in output un Vector4d contenente, nelle prime tre posizioni, le coordinate del baricentro B e nell'ultima contiene $R = \max_d\{d = ||V - B||_2^2 : V \in \mathcal{F}\}$, dimodoché si possa associare ad ogni frattura una sfera centrata in B avente come raggio R.
  - **double distance(const Vector3d& point1, const Vector3d& point2)**: restituisce un double $d = ||P_1 - P_2||_2$ dati due $P_1, P_2 \in \mathbb{R}^3$ ricevuti in input sotto forma di Vector3d.
  - **Vector3d normal(const vector<Vector3d>& vertexdata)**: dato un vector di Vector3d contenente almeno 3 elementi, calcola la normale al piano contenente i primi tre elementi P_0, P_1, P_2 svolgendo il seguente prodotto vettoriale: $(P_1 - P_0) \times (P_2 - P_0)$.
  - **bool intersectrettaretta(const double& epsilon, const Vector3d& point, const Vector3d& dir,const Vector3d& point1, const Vector3d& dir1, Vector3d& inter)**: in input riceve due rette (ognuna di queste è univocamente determinata da un punto $P_i \in \mathbb{R}^3$ e una direzione $v_i \in \mathbb{R}^3$). Risolve il sistema parametrico tra le due rette salvando il valore dei due parametri in s e t. Nel nostro caso, la prima retta è identificata dalla retta di intersezione tra i due piani contenenti le fratture, mentre la seconda retta è in realtà un segmento di una frattura. Per questo motivo il calcolo dell'intersezione è seguito da una verifica sul parametro s: $\epsilon < s < 1 + \epsilon$, dove ϵ è la tolleranza scelta. Se viene trovata un'intersezione che verifica la condizione sul parametro s, allora la funzione restituisce un bool $true$ e inserisce all'interno del Vector3d $inter$ il punto trovato. Viceversa restituisce un bool $false$ e assegna un punto arbitrario L a $inter$ (in questo caso $L = (1000, 1000, 1000)$).



(a) La retta di intersezione tra i piani interseca entrambe le fratture in due punti. Due punti di intersezione sono comuni a tutte e due le fratture: le fratture si intersecano.

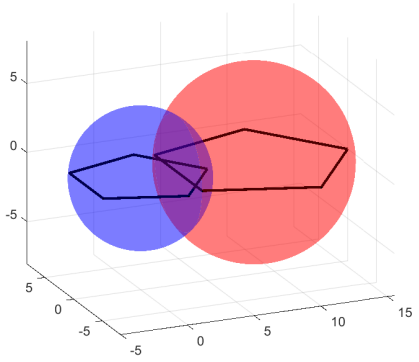


(b) La retta di intersezione tra i piani interseca entrambe le fratture in due punti. Non vi sono punti di intersezione comuni tra le fratture: le fratture non si intersecano.

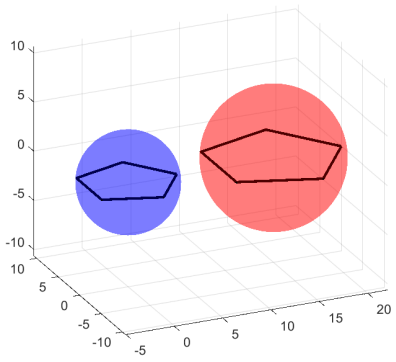
  - **bool intersectrettasemiretta(const double& epsilon, const Vector3d& point, const Vector3d& dir, const Vector3d& point1, const Vector3d& dir1,Vector3d& control)**: svolge lo stesso compito della funzione intersectrettaretta con la differenza che $t \geq 0$. All'interno del programma trova le eventuali intersezioni tra una semiretta ed un segmento.

- **Polygons**
  Funzioni chiave del programma, che agiscono direttamente sui poligoni e che permettono, grazie agli algoritmi in Analytics, di leggere il file e trovare le tracce.

  - **bool importdfn(const string& filename, Fractures& fractures)**: elabora il file preso in input e popola *struct Fractures* contenente tutte le informazioni sulle fratture che compongono il DFN.
  - **list<vector<unsigned int>> checkspheres(const Fractures& fractures)**: riceve in input la struct *Fractures* e restituisce una lista di coppie di ids di fratture le cui sfere (calcolate in calcsphere) si intersecano. In particolare, salva le coppie di fratture le cui sfere $S_i = \{||x - C_i||_2^2 = {R_i}^2 : x \in \mathbb{R}^3\}$ soddisfano $||C_1 - C_2||_2^2 < (R_1 + R_2)^2$ (è stato utilizzato il metodo $.squaredNorm()$ invece di $.norm()$ al fine di ridurre i costi computazionali). Le sfere "eccessivamente lontane" tra loro vengono così escluse, consentendo al programma di non risolvere sistemi lineari in eccesso e di diminuire il tempo totale di calcolo delle tracce.

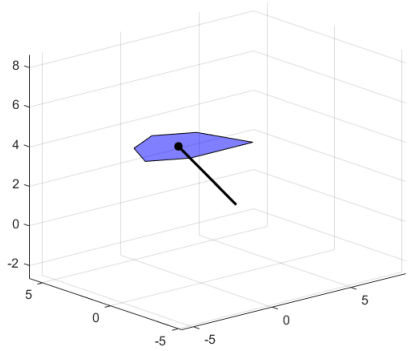


(a) Le sfere contenenti le due fratture si intersecano: la coppia di fratture viene inserita in goodcouples.

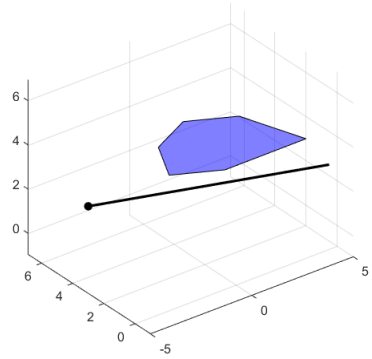


(b) Le sfere contenenti le due fratture non si intersecano: la coppia di fratture viene scartata.

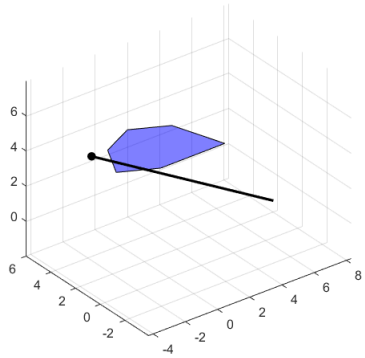
– **void tracesfinder(const Fractures& fractures, const list<vector<unsigned int>>& goodcouples, Traces& traces, const double& epsilon)**: riceve in input la struct contenente le fratture, la lista di coppie di ids la cui intersezione è probabile, la struct delle tracce ed una tolleranza ϵ presa dalla command line. Per ogni coppia di fratture $(\mathcal{F}_1, \mathcal{F}_2)$ trova la retta di intersezione *tangent* tra i piani su cui le fratture giacciono (dopo aver verificato che i piani non siano paralleli). Si cercano prima eventuali intersezioni tra tale retta e i segmenti di $\mathcal{F}_1$. Per ogni intersezione *inter* trovata viene utilizzato il seguente criterio per verificare che effettivamente si tratti dell'intersezione tra due fratture e non semplicemente l'intersezione tra una frattura e *tangent* : si genera una semiretta s in direzione casuale che parte da *inter* giacente sul piano contenente $\mathcal{F}_2$ e si contano le intersezioni tra s e $\mathcal{F}_2$. Data la convessità delle fratture, se il numero di intersezioni trovate è dispari, allora *inter* è certamente interno a $\mathcal{F}_2$ e può dunque essere considerato un valido punto di intersezione tra $\mathcal{F}_1$ e $\mathcal{F}_2$. Se il numero di intersezioni è 0, allora *inter* è esterno a $\mathcal{F}_2$ e non è un valido punto di intersezione. Se, invece, il numero di intersezioni è 2 si distingue il caso in cui *inter* è inclusa in tali intersezioni oppure non lo è: nel primo caso l'intersezione è valida poichè significa che un segmento di $\mathcal{F}_1$ interseca un segmento di $\mathcal{F}_2$, altrimenti significa che *inter* è esterno a $\mathcal{F}_2$ e le due fratture non si intersecano. Dopo aver trovato le eventuali intersezioni tra le due fratture, che si identificano con gli estremi della traccia, tracesfinder classifica le tracce in passanti e non passanti per ognuna delle due fratture della coppia. Affinchè una traccia sia passante per una frattura è necessario che entrambi gli estremi della traccia trovati giacciano su un lato. Per fare ciò, si controlla che la somma delle distanze tra l'estremo e i vertici sia uguale alla lunghezza del lato stesso. Dopo aver effettuato il controllo si popolano $TipsTrue$ e $TipsFalse$ a seconda che la traccia sia passante o non passante rispettivamente.



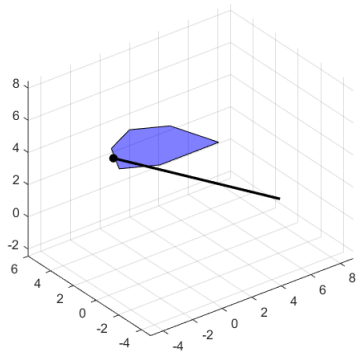
(a) La semiretta interseca un punto del poligono, il punto è interno



(b) La semiretta non interseca alcun punto del poligono, il punto è esterno.



(c) La semiretta interseca due punti del poligono, il punto è esterno.



(d) La semiretta interseca due punti del poligono, ma uno di questi appartiene al bordo del poligono, il punto è interno.

- **OutputFileTools**
  Sono le funzioni che terminano la prima parte del progetto, stampando quanto trovato sin'ora sui file di output nella giusta formattazione.

  - **bool printtraces(const string& tracesfileout, const Traces& traces)**: funzione che si occupa di creare il file di output nel formato richiesto. In particolare stampa l'id e le coordinate degli estremi di ogni traccia.
  - **bool printtips(const string& tipsfileout, Traces& traces, const Fractures& fractures)**: riceve in input il file di output "*tips_NumeroFratture.txt*", la *struct* contenente le tracce e relative informazioni e la *struct* delle fratture. Ordina le tracce in ordine decrescente di lunghezza. Successivamente, scorre sugli ids di ogni frattura e per ognuna di queste memorizza nel vector di unsigned int *idtraces* gli ids delle tracce su di essa. Dopo aver ricevato il numero di tracce totali su una certa frattura (e averlo stampato nel formato richiesto) la funzione stampa sul file di output prima le informazioni riguardanti le tracce passanti per quella frattura e, successivamente, le informazioni riguardanti quelle non passanti. Per entrambe le tipologie di tracce viene richiesto l'id della traccia, se è passante o non passante per la frattura (tramite il parametro *Tips*) e la lunghezza della traccia.

3 Parte 2: Determinazione dei Sotto-Poligoni

L'obiettivo della seconda parte del progetto è di analizzare i sottopoligoni generati dal taglio delle fratture con le relative tracce. Tali sottopoligoni sono stati memorizzati nella seguente struct e stampati su un file di output.

3.1 Le structure

- **struct PolygonalMesh**: struct contenente tutte le informazioni del DFN dopo che sono stati effettuati i tagli in sottopoligoni. In particolare, abbiamo:
  - **Celle0d**: tutti i vertici sono inseriti all'interno di un vector e ognuno di essi è identificato in modo univoco da un id e dalle sue coordinate nello spazio 3d.
  - **Celle1d**: i segmenti sono univocamente determinati da un proprio id e dagli ids delle due $Celle0d$ che ne costituiscono gli estremi.
  - **Celle2d**: poligoni univocamente determinati da un proprio id, da un vector contenente gli ids dei suoi vertici ($Celle0d$) e da un vector contenente gli ids dei suoi lati ($Celle1d$)

  Viene inoltre registrato il numero totale di ogni tipo di cella presente all'interno del DFN dopo il taglio in sottopoligoni.

- **struct pair_hash**: struct che agisce da mappa su pair<unsigned int, unsigned int> mediante operator(). In particolare definisce una funzione di hash ad-hoc sulle coppie di unsigned int, non presente di default, basato sulle funzioni hash di *unordered_set* in modo che queste possano essere effettivamente utilizzate come chiavi nel container. L'utilizzo dell'*unordered_set* è risultato essere fondamentale per controllare in modo computazionalmente efficiente ($O(1)$) le coppie di lati in precedenza inserite. Infatti in questa struttura dati ogni chiave è unica e non sono presenti duplicati. Si è preferito utilizzare l' *unordered_set* al posto di una più comune *map* perché quest'ultima è basata su alberi binari 1-bilanciati e l'accesso, l'inserimento e la cancellazione degli elementi hanno un costo $O(log(n))$.

3.2 Metodologie e Algoritmi utilizzati

Per la seconda parte abbiamo utilizzato una funzione tuttofare *meshcalc* al cui interno sono stati implementati gli algoritmi logici utili a risolvere il problema posto.
Brevemente, dopo aver controllato quali delle tracce memorizzate passassero per la frattura in questione, abbiamo tagliato la frattura lungo le suddette tracce, dando priorità a quelle passanti e ordinate per lunghezza decrescente. I vertici dei sottopoligoni così ottenuti sono stati memorizzati in senso antiorario nella struct PolygonalMesh, giungendo alla conclusione del progetto.
In particolare è risultato fondamentale il namespace *Analytics*, con cui abbiamo realizzato l'algoritmo di taglio e suddivisione in sottopoligoni. Per quanto riguarda l'ordinamento di vertici (e quindi anche di segmenti) in senso antiorario sono state aggiunte al namespace *Analytics* due funzioni: *calcolangolo* e *antiorario*.

- **Analytics**
  - **double calcolangolo(const Vector3d& v1, const Vector3d& v2, const Vector3d& normal)**: la funzione riceve in input due Vector3d complanari e la normale al piano su cui giacciono e restituisce un double che idenfitica l'angolo θ tra i due vettori. Calcola prima il prodotto vettoriale $v_1 \times v_2$ e il prodotto scalare $v_1 \cdot v_2 = ||v_1||||v_2|| \cos\theta$. Si serve successivamente della funzione $atan2(y, x)$ che, dato un punto (x, y) nel piano cartesiano, restituisce l'angolo che il raggio vettore che collega l'origine $(0, 0)$ e il punto (x, y) forma con l'asse x. In particolare $atan2$ riceve in input $||v_1 \times v_2|| = ||v_1||||v_2|| \sin\theta$ e $v_1 \cdot v_2$. Se il prodotto misto con i vettori v_1 e v_2 e la normale al piano dovesse risultare negativo risulta necessario rimappare l'angolo $\theta = atan2(||v_1 \times v_2||, v_1 \cdot v_2)$ nell'angolo $\theta \to 2\pi - \theta$ in modo tale che $atan2 : [-1, 1]^2 \to [0, 2\pi)$
  - **void antiorario(vector<Vector3d>& sottopol, const Vector3d& normal)**: riceve in input un poligono sottoforma di vector di Vector3d e la normale al piano su cui giace il poligono. La funzione calcola il baricentro del poligono come media pesata dei vertici ed associa ad ogni vertice l'angolo che il raggio vettore che collega il baricentro e il vertice forma con un vettore arbitrario (nel nostro caso $v = (1, 0, 0)$). I vertici vengono poi ordinati in ordine crescente in base all'angolo ad essi associato in modo tale che risultino ordinati in senso antiorario.

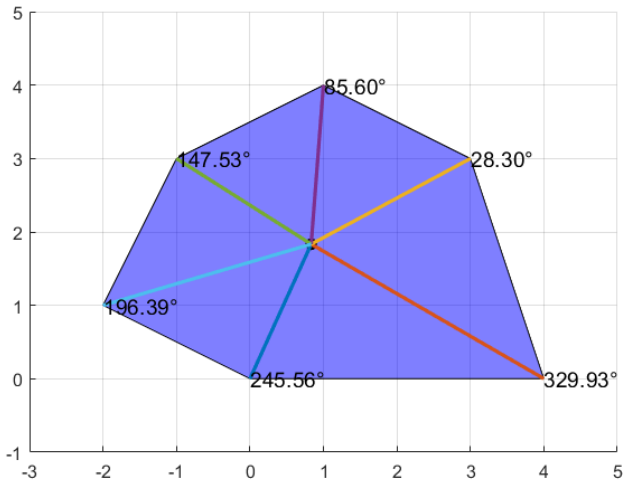


Figure 4: Angolo formato dai raggi che collegano i vertici e il baricentro con l'asse x. E' possibile ordinare i vertici in senso antiorario.

- **Meshlibrary**
  - **bool meshcalc(const double& epsilon,const Traces& traces, const Fractures& fractures, vector<PolygonalMesh>& mesh)**: la funzione riceve in input le *struct* delle tracce e delle fratture e modifica l'oggetto di tipo $PolygonalMesh$ in modo tale che questo contenga tutte le informazioni richieste sulle celle 0d, 1d e 2d. Per ogni frattura viene creato un vector di tracce popolato prima con tutte le tracce passanti e poi con tutte quelle

non passanti. Segue il calcolo ricorsivo dei sottopoligoni, realizzato nel modo seguente:

* **Step 1**: creazione di un vector di sottopoligoni *sottopoligoni* (inizializzato in modo tale che contenga da subito il poligono iniziale)
* **Step 2**: per ogni traccia crea un vector *copia* di *sottopoligoni* utile ad inserire in *sottopoligoni* i sottopoligoni generati dai tagli.
* **Step 3**: Per ogni estremo della traccia utilizziamo la tecnica (2.2) della semiretta per verificare che gli estremi della traccia siano interni al poligono corrente.
* **Step 4**: Per ogni sottopoligono verifichiamo se la traccia (intesa come segmento) interseca due segmenti del poligono corrente. In particolare controlliamo che la retta passante per la traccia intersechi il lato (e registriamo l'intersezione come nuovo vertice) e successivamente verifichiamo che questa sia interna alla traccia con il seguente criterio: dato un segmento $\bar{AB}$ e un punto P giacente sulla retta passante per il segmento, allora

$$P \in \overline{AB} \iff ||P - A|| + ||P - B|| - ||B - A|| = 0$$

(nel nostro caso è sufficiente che la differenza sia superiore alla tolleranza ϵ presa in input tramite la linea di comando).

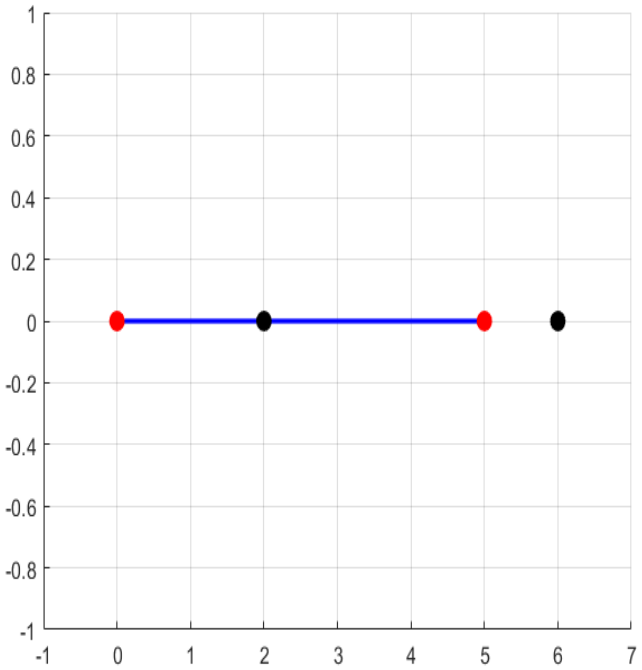


Figure 5: Punto interno e punto esterno al segmento.

* **Step 5**: Se non vi sono estremi della traccia interni al sottopoligono, nè la traccia interseca il sottopoligono in due punti, allora non è possibile tagliare il poligono e può essere inserito all'interno di copia.
* **Step 6** Dopo aver eliminato eventuali duplicati all'interno del vector contenente gli eventuali nuovi vertici trovati e verificato che il numero di nuovi

vertici trovati è effettivamente 2, è possibile effettuare il taglio del poligono corrente in due ulteriori sottopoligoni (sempre inizializzati sottoforma di vector di Vector3d). In ogni sottopoligono aggiungiamo i nuovi vertici v_1, v_2 trovati (necessariamente appartenenti ad entrambi) e suddividiamo i vertici u del poligono di partenza in due categorie, una per ogni sottopoligono: quelli per cui vale $(v_1 - v) \times (v_2 - v) \cdot n > 0$ e quelli per cui vale $(v_1 - v) \times (v_2 - v) \cdot n < 0$, dove n è la normale al piano contenente i sottopoligoni. Ordiniamo ogni vector utilizzando la funzione *antiorario* e inseriamo i sottopoligoni generati in *copia*.

* **Step 7**: Memorizza *copia* in *sottopoligoni* e svuota *copia*.

Inoltre, per ogni vertice trovato che si trovi su una traccia, controlliamo se è anche un vertice adiacente per ogni sottopoligono in *sottopoligoni* e, se lo è, lo aggiungiamo al sottopoligono (i cui vertici andranno nuovamente ordinato utilizzando la funzione *antiorario*).
Dopo aver terminato i tagli, *meshcalc*, per ogni frattura, popola un oggetto del tipo $PolygonalMesh$ associato alla frattura corrente. Il passaggio più computazionalmente difficile dell'intero programma è risultato essere l'inserimento nella mesh delle celle 1d a causa dell'elevato possibile numero totale (e di duplicati) di nuovi lati generati dopo il taglio. Per rendere il problema computazionalmente meno oneroso ci siamo serviti degli *unordered_sets* come descritto in 3.1. Abbiamo deciso di non utilizzare lo schema di popolamento della mesh delle celle 1d anche per le celle 0d e 2d, ma a causa dei controlli sulle distanze (effettuati utilizzando la tolleranza ϵ) è risultato più semplice utilizzare dei *vectors* (con un costo computazionale $O(n)$ per ogni controllo). L'implementazione dello schema con gli *unordered_sets* avrebbe richiesto la creazione di una funzione di hash ad-hoc anche per i Vector3d e un algoritmo logicamente molto più complesso per effettuare i controlli a meno della tolleranza.

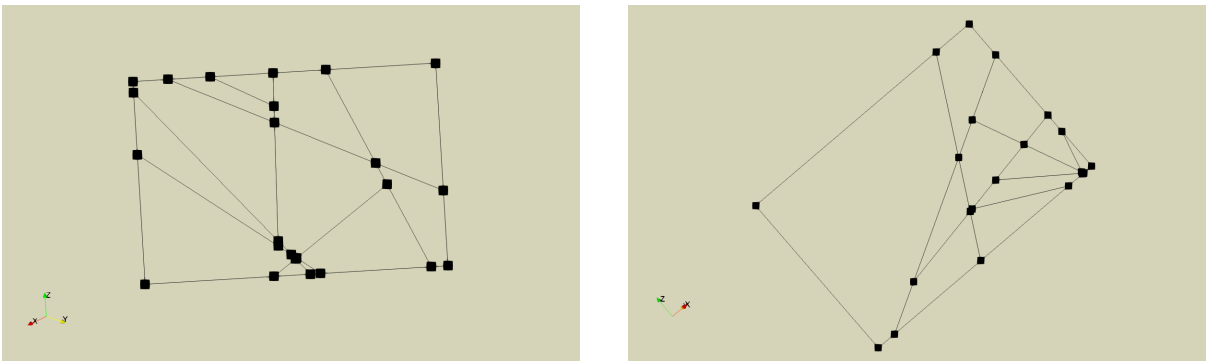


Figure 6: Visualizzazione su Praview della mesh di due fratture presenti nel file "FR10_data.txt".

.

4 Testing

Per la parte di testing ci siamo serviti dei google test. Abbiamo testato ogni funzione prima su casi semplici e poi su casi generali e più complessi, in modo da verificarne il corretto funzionamento per ogni possibile situazione.
Inoltre, abbiamo utilizzato la stessa tolleranza fornita da barra di comando in modo da rendere coerenti i test con il resto del programma: infatti, dato che più la tolleranza è alta più i risultati sono grezzi, ha senso testare le funzioni con lo stesso livello di precisione.

5 Tempi computazionali: una breve analisi

Utilizzando la libreria *chrono*, abbiamo raccolto alcuni dati sui tempi di calcolo totale del programma.
Si evince che i tempi di calcolo non aumentano in modo esponenziale in funzione del numero di tracce n(unico parametro veramente rilevante nella stima del numero di tagli e sottopoligoni generati). In particolare la dipendenza della funzione tempo $T(n)$ dalle tracce risulta essere all'incirca $T(n) \approx 0.001n^{1+\epsilon}$ suggerendo una dipendenza di tipo lineare per $0 < n \leq 10^4$. Nonostante ciò, la stima sembra sottostimare progressivamente $T(n)$ per $n >> 10^4$, suggerendo che per un numero di tracce molto elevato l'algoritmo di memorizzazione della mesh potrebbe risultare inefficiente.

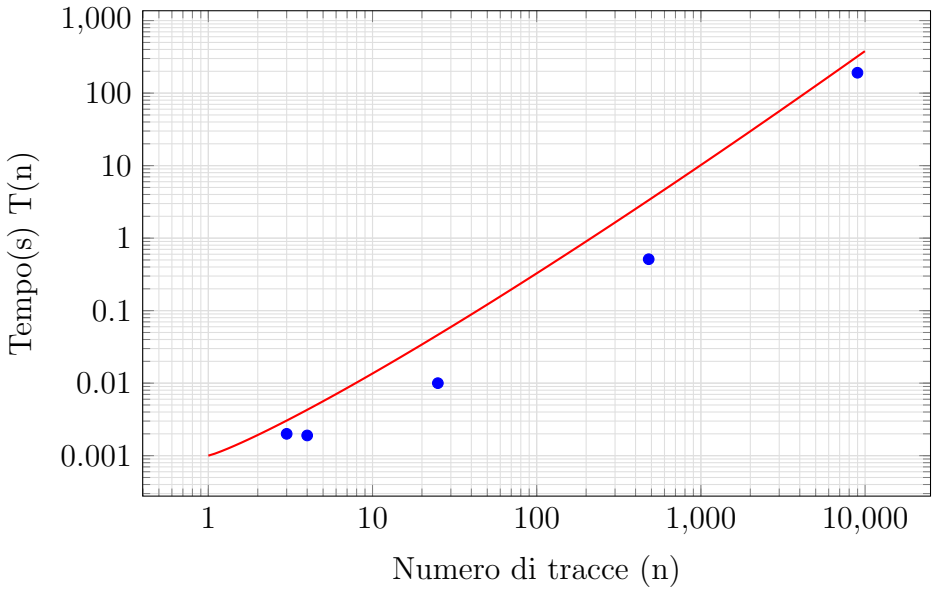


Figure 7: Grafico in scala logaritmica del tempo complessivo di esecuzione del programma in funzione del numero di tracce e comparazione con la funzione $T(n) = 0.001e^{log^{1.15}(n)}$