Immagine che contiene Elementi grafici, Carattere, grafica, logo

Descrizione generata automaticamente

Politecnico di Torino

Corso di Laurea in Matematica per l’Ingegneria

**Discrete Fracture Network**

Progetto Programmazione e Calcolo Scientifico 2024

|  |  |
| --- | --- |
| Relatori: | Candidati: |
| S. Berrone  M. Cicuttin  G. Teora  F. Vicini | M. Buffoblin  G. Calabrese  S. Silvestro |

# **Sommario**

1. INTRODUZIONE AL PROBLEMA 2

2. STRUTTURE DATI E DOCUMENTAZIONE UML 3

3. FUNZIONALITÀ ED IMPLEMENTAZIONE 5

4. TESTING 9

5. CONCLUSIONI 10

# **INTRODUZIONE AL PROBLEMA**

Un Discrete Fracture Network (DFN) è un metodo utilizzato in geologia per studiare i modelli di flusso attraverso le fratture presenti nelle rocce al cui interno possono scorrere dei fluidi.

Nell’analisi di una popolazione di fratture si devono individuare le fratture isolate, ovvero quelle che non si intersecano con altre fratture, in modo da escluderle a priori nella valutazione delle intersezioni. In questo contesto si è tralasciata l’analisi delle fratture che presentano come intersezione un segmento di dimensione nulla, vale a dire un singolo punto.

Un DFN è costituito da un insieme di fratture, in questo particolare caso rappresentate da una serie di poligoni planari che potrebbero intersecarsi nello spazio tridimensionale. Queste intersezioni sono chiamate tracce, corrispondenti a segmenti di diversa lunghezza, eventualmente nulla o ridotta ad un unico punto.

Si possono classificare le tracce suddividendole in due tipologie:

1. Traccia passante: un segmento con entrambi gli estremi giacenti sul bordo della frattura.
2. Traccia non passante: segmento che presenta almeno un estremo appartenente alla superficie interna della frattura stessa.

Questo tipo di classificazione per le tracce è riferito ad ogni poligono: una traccia può risultare passante per uno dei due poligoni che la genera e non passante per il secondo poligono che contribuisce alla sua creazione.

In questa prima parte del progetto proposto si ha l’obiettivo di identificare e classificare le tracce presenti all’interno di ciascuna frattura. Inizialmente si legge un file contenente una serie di poligoni identificati da un numero intero e descritti dai vertici che li compongono nello spazio.

Si è poi utilizzato il concetto di baricentro e di sfera circoscritta per valutare la posizione reciproca tra poligoni.

Successivamente si utilizzano delle funzioni specifiche per il calcolo dell’equazione del piano su cui giace un poligono, dell’equazione di una retta generata dall’intersezione tra due piani e dell’equazione di una retta passante per due punti, corrispondenti a due dei vertici di un poligono.

Un’ulteriore funzione è stata implementata per calcolare l’ascissa curvilinea delle rette ricavate in precedenza, sfruttando il metodo di risoluzione dei sistemi lineari con decomposizione QR.

Dopo aver popolato le strutture dati relative a frattura e traccia, si salva tutto nel file *‘Traces.txt’*. Ogni file derivante dall’esecuzione relativa ai diversi file di input forniti dal progetto, viene poi riportato all’interno di una cartella che raccoglie i risultati finali.

Infine, per la validazione delle singole unità logiche vengono utilizzati i test della libreria dei GoogleTest per verificare la correttezza delle singole funzioni implementate.

Per ogni traccia si memorizza la coppia degli identificativi delle fratture che la generano definiscono e le coordinate tridimensionali dei punti vertici che la identificano. Invece Per ciascuna frattura, si differenziano le tracce in passanti e non passanti, tramite un booleano: se la traccia risulterà passante si avrà ‘false’ al contrario le tracce non passanti riporteranno il valore true. Si ordinano i due sottoinsiemi distinti separatamente per lunghezza, in ordine decrescente. (SOTTO)

PARTE 2 – IN SOSPESO

Il secondo obiettivo del progetto riguarda la determinazione dei sotto-poligoni generati dal taglio delle fratture con le loro tracce. Ciascuna frattura, dato il suo sottoinsieme delle tracce, sarà inizialmente tagliata dalle tracce passanti. Per ciascuna traccia passante la frattura sarà divisa in due parti, di conseguenza l’elenco delle fratture sarà aggiornato con i nuovi sotto-poligoni generati. Una volta completato il taglio con le tracce passanti, si procederà con le tracce non-passanti, che verranno prolungate fino al bordo della frattura, generando così altri sotto-poligoni.

# **STRUTTURE DATI E DOCUMENTAZIONE UML**

**2.1 STRUTTURE DATI**

Le strutture dati scelte per contenere i dati necessari allo sviluppo del problema (vettori, matrici e array), influenzano direttamente le prestazioni computazionali e l’implementazione del programma.

Per cominciare abbiamo costruito un namespace chiamato *‘DFNLibrary’* al cui interno abbiamo definito due strutture distinte: *‘Fractures’* e *‘Traces’*.

La struttura *‘Fractures’* è stata progettata per rappresentare l’insieme di fratture all’interno del DFN. Le strutture dati che la compongono sono:

* *‘NumberFractures’*: variabile di tipo unsigned int che tiene traccia del numero totale di fratture contenute nel file in lettura
* *‘Id’*: variabile di tipo unsigned int che rappresenta l’identificatore univoco per ogni frattura
* *‘Vertices’*: vettore di matrici contenente tutti i vertici di ogni poligono. Le matrici hanno 3 righe che rappresentano le coordinate ‘x, y, z’ dello spazio e un numero dinamico di colonne, poiché il numero dei vertici che definiscono ogni frattura è variabile. Dunque ogni matrice contiene i vertici di una singola frattura
* *‘Plane’:* vettore di array di 4 elementi di tipo double che rappresentano i coefficienti ‘a, b, c, d’ dell'equazione del piano in forma parametrica su cui poggia il poligono.

La struttura *‘Traces’* è stata progettata per rappresentare l’insieme delle tracce generate dalle intersezioni tra le fratture.

Le strutture dati che la compongono sono:

* *‘NumberTraces’*: variabile di tipo unsigned int che tiene traccia del numero totale di tracce che vengono generate dall’intersezione tra poligoni
* *‘FracturesId’*: vettore di array di dimensione 2 contenente variabili di tipo unsigned int che contiene gli identificatori delle fratture che generano ogni traccia
* *‘Vertices’*: vettore di array di dimensione 2 contenente due vettori tridimensionali che rappresentano le coordinate dei due vertici della traccia. Per conseguenza, ogni array contiene i vertici di ogni traccia
* *‘Tips’*: vettore di array contenente due valori booleani: se la traccia risulterà passante si avrà ‘false’, al contrario le tracce non passanti riporteranno il valore ‘true’. In particolare si avrà un booleano per ciascuna delle due fratture che genera la traccia.

PARTE 2 – IN SOSPESO

Successivamente abbiamo costruito il namespace *‘PolygonalLibrary’* al cui interno è definita la struttura *‘PolygonalMesh’*, che è utilizzata per rappresentare una mesh poligonale composta da celle di dimensione 0 per i vertici, 1 per i lati e 2 per i poligoni:

Ogni cella è formata da una variabile *‘NumberCellND’* di tipo unsigned int che rappresenta il numero di celle per ciascuna dimensione e da un vettore di interi senza segno che contiene gli identificatori univoci per ciascuna cella di ogni dimensione. Inoltre ogni cella contiene delle strutture dati che la caratterizza e la distingue dalle altre:

* Cell0DCoordinates è un vettore di vettori di dimensione 3 che rappresentano le coordinate x, y, z di ciascun vertice della mesh.
* Cell1DIdVertices è un vettore di array di dimensione 2 che contengono gli identificatori dei vertici che rappresentano gli estremi di ciascun segmento.
* Cell2DVertices è un vettore di vettori che contengono gli identificatori dei vertici che formano ciascun poligono.
* Cell2DEdges è un vettore di vettori di dimensione variabile che contengono gli identificatori dei lati che compongono ciascun poligono.

AGGIUNGI

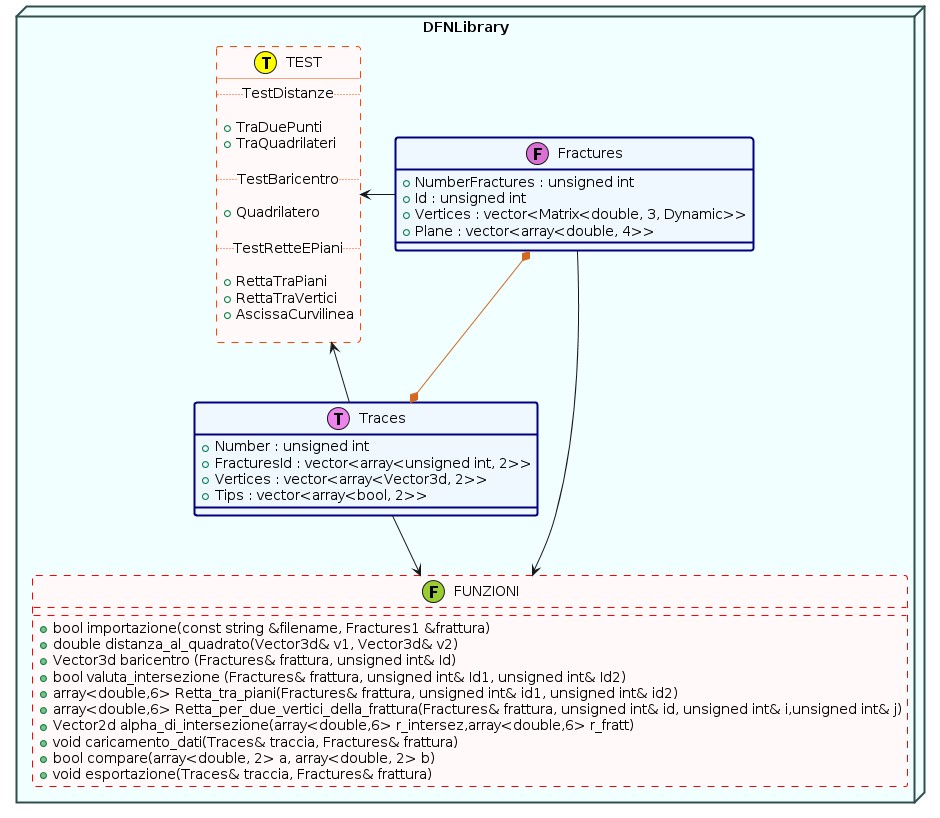
Abbiamo deciso di utilizzare i vettori quando la dimensione risulta dinamica e varia durante l’esecuzione del programma, oppure quando è necessario aggiungere o rimuovere elementi frequentemente.

Inoltre abbiamo utilizzato gli array quando la dimensione risulta fissa, dunque è utile perché offre accesso rapido ed efficiente ai dati in esso contenuti.

**2.2 DOCUMENTAZIONE UML**

Di seguito si riporta l’immagine relativa all’implementazione UML della struttura data al presente progetto.

Si riporta il namespace esterno contenente le strutture, evidenziate dal bordo a linea continua, e l’insieme dei test svolti e delle funzioni scritte, evidenziati dal bordo tratteggiato. Questa distinzione è stata pensata in quanto Test e Funzioni non sono classi o strutture, dunque non appartengono completamente a nessuna delle due. Per conseguenza si è scelto di mantenerle esterne perché attingono e comunicano da entrambe le strutture citate.



# **FUNZIONALITÀ ED IMPLEMENTAZIONE**

All’interno del codice si possono individuare tre sviluppi successivi per giungere all’obiettivo della classificazione delle tracce contenute in ogni frattura: il caricamento dei dati da file, l’individuazione e il calcolo delle tracce e l’esportazione dei risultati ottenuti.

Di seguito si descrivono le funzioni implementate con il relativo scopo e fondamento matematico di ciascuna.

***bool importazione(const string& filename, Fractures& frattura)***

Questa funzione legge i dati da un file e popola la struct Fractures con questi dati. Inizialmente si utilizza un oggetto ifstream per aprire il file contenente i dati delle fratture. Legge il numero di fratture presenti e lo memorizza nella variabile numberFractures. Per ogni frattura la funzione legge l’identificatore della frattura, il numero di vertici e memorizza le coordinate di quest’ultimi nella matrice Tab\_coord\_vertici. Successivamente la funzione calcola i coefficienti dell'equazione del piano di ciascun poligono, e questi vengono aggiunti alla variabile Piano della struct Fractures. Dopo aver letto ed elaborato tutte le fratture, la funzione chiude il file e restituisce true per indicare che l'operazione è stata completata con successo.

***Vector3d baricentro (Fractures& frattura, unsigned int& Id1)***

La funzione baricentro calcola il centroide di una frattura specificata dal suo identificatore. Ottenuto il numero di vertici, per ciascuna delle tre dimensioni (x, y, z) viene sommata la coordinata di ciascun vertice. Ottenuta la somma si divide per il numero di vertici, ottenendo così le coordinate del centroide, che vengono restituite tramite un Vector3d.

***bool valuta\_intersezione (Fractures& frattura, unsigned int& Id1, unsigned int& Id2)***

Questa funzione valuta, senza dover risolvere sistemi di equazioni lineari, se due fratture possono intersecarsi basandosi sulle loro posizioni e sui raggi delle sfere circoscritte ai loro vertici. Ottenuti il numero di vertici e il centroide delle due fratture, si calcolano i raggi circoscritti al poligono come le distanze al quadrato dai vertici al baricentro. Si determina il raggio massimo per ciascuna frattura. Successivamente si verifica se la distanza al quadrato tra i baricentri è minore o uguale alla somma dei raggi delle sfere circoscritte al poligono, meno una piccola tolleranza. Se questa condizione è soddisfatta allora le fratture potrebbero intersecarsi e la funzione restituisce true, altrimenti le fratture sicuramente non si intersecano.

Successivamente si individua il baricentro di ogni poligono e il raggio della sfera circoscritta ad ognuno, considerando come raggio la distanza tra il baricentro e il vertice del poligono da esso più distante. Si prenderanno coppie di poligoni, e si calcolerà la loro distanza relativa al quadrato. In questo modo si andranno ad escludere i casi in cui i poligoni sicuramente non si intersecano perché distanti tra loro più della distanza dei loro baricentri. Così procedendo si dovrà studiare un minor numero di possibili intersezioni.

***array<double,6> Retta\_tra\_piani(Fractures& frattura, unsigned int& id1, unsigned int& id2)***

Questa funzione calcola la retta di intersezione tra due piani definiti dalle fratture. La retta viene rappresentata in forma parametrica:

La direzione della retta di intersezione tra i due piani, quindi (coord\_retta[0], coord\_retta[1], coord\_retta[2]), è determinata dal prodotto vettoriale dei loro vettori normali, che sono definiti dai coefficenti dell’equazione del piano. Invece per trovare un punto che giace sulla retta di intersezione (coord\_retta[4], coord\_retta[5], coord\_retta[6]), risolviamo un sistema lineare del tipo . La matrice A contiene i coefficienti delle variabili (x, y, z) delle due equazioni dei piani e della direzione della retta. Il vettore b contiene i termini noti delle equazioni dei piani. Utilizziamo la decomposizione LU per risolvere il sistema lineare. Questa operazione ci fornisce il vettore x che contiene le coordinate del punto di intersezione dei piani. Le coordinate della direzione della retta e del punto di intersezione vengono memorizzate nell'array `coord\_retta` che viene poi restituito dalla funzione.

***array<double,6> Retta\_per\_due\_vertici\_della\_frattura(Fractures& frattura, unsigned int& id, unsigned int& i,unsigned int& j)***

Questa funzione calcola la retta passante per due vertici di una frattura. Quest’ultima è rappresentata da dove è la direttrice e il punto di partenza P è semplicemente uno dei due vertici. Le coordinate della direzione della retta e del punto di partenza vengono memorizzate in un array di dimensione 6, dove i primi tre elementi rappresentano la direzione e gli ultimi tre rappresentano il punto di partenza.

***Vector2d alpha\_di\_intersezione(array<double, 6> r\_intersez, array<double, 6> r\_fratt)***

Questa funzione calcola i parametri alpha e beta che determinano i punti di intersezione tra due rette in uno spazio tridimensionale. Questi parametri vengono trovati risolvendo un sistema lineare del tipo . Le colonne della matrice A sono composte dai vettori direzionali delle due rette:

A.col(0) = [r\_fratt[0], r\_fratt[1], r\_fratt[2]]

A.col(1) = [r\_intersez[0], r\_intersez[1], r\_intersez[2]]

Il vettore b rappresenta la differenza tra i punti di partenza delle due rette:

b = [r\_intersez[3] - r\_fratt[3],

r\_intersez[4] - r\_fratt[4],

r\_intersez[5] - r\_fratt[5]]

Il sistema lineare viene risolto usando la decomposizione QR. La funzione restituisce x = [alpha, beta] che sono rispettivamente l’ascissa curvilinea della retta passante per due punti e della retta generata dall'intersezione tra piani.

***void caricamento\_dati(Traces& traccia, Fractures& frattura)***

Questa funzione calcola e memorizza le intersezioni fra le fratture (cioè le tracce) nella struct Traces.

La funzione itera su tutte le fratture e verifica l'intersezione di ciascuna frattura con ogni frattura successiva. Inizialmente utilizziamo la funzione *valuta\_intersezione*, così da poter capire con certezza se le fratture non si intersecano. Se quest’ultime potrebbero intersecarsi calcoliamo la retta di intersezione tra piani delle fratture (tramite *Retta\_tra\_piani*), escludendo la condizione dei piani paralleli. Per ogni lato della frattura 1 calcoliamo la retta generata dai vertici (tramite *Retta\_per\_due\_vertici\_della\_frattura*). Verifichiamo se c’è intersezione tra la retta del lato del poligono e la retta di intersezione dei piani, anche qui escludendo la condizione di parallelismo. Se questa condizione è vera allora, tramite *alpha\_di\_intersezione*, verifichiamo se la retta di intersezione tra piani interseca la retta generata dal lato del poligono all’interno del segmento, cioè controlliamo che alpha sia compreso tra 0 e 1, tenendo conto di una piccola tolleranza. Se questo avviene incrementiamo il contatore che conta i punti di intersezione della frattura e memorizziamo beta. Ripetiamo il processo per la frattura 2.

Se cont è uguale a 4, cioè sono stati trovati due punti di intersezione validi per ciascun poligono, allora analizziamo l’intervallo di intersezione sull'asse delle ascisse, che è determinato dai punti di intersezione più a sinistra (sx) e più a destra (dx). Qualora gli intervalli si sovrapponessero (sx < dx) rianializziamo i due poligoni controllando se i punti di intersezione appartengono al segmento, all’altro poligono e all’intervallo di intersezione. Successivamente vengono considerati tre casi possibili in base alla configurazione dei punti di intersezione trovati:

* + Caso 1: Se entrambi i poligoni hanno due punti di intersezione validi all'interno dell'intervallo [sx, dx], allora la traccia viene considerata passante per entrambi i poligoni.
  + Caso 2: Se il primo poligono non ha punti di intersezione e il secondo poligono ne ha due all'interno dell'intervallo [sx, dx], la traccia viene considerata passante solo per il secondo poligono.
  + Caso 3: Se il primo poligono ha un punto di intersezione e il secondo poligono ne ha uno all'interno dell'intervallo [sx, dx], la traccia viene considerata non passante per entrambi i poligoni.

In ciascuno di questi casi, vengono impostati i tipi di traccia corrispondenti (Tipo[0] = passante e Tipo[1] = non passante) e le informazioni sulla traccia (vertici, ID dei poligoni e tipo) vengono memorizzate.

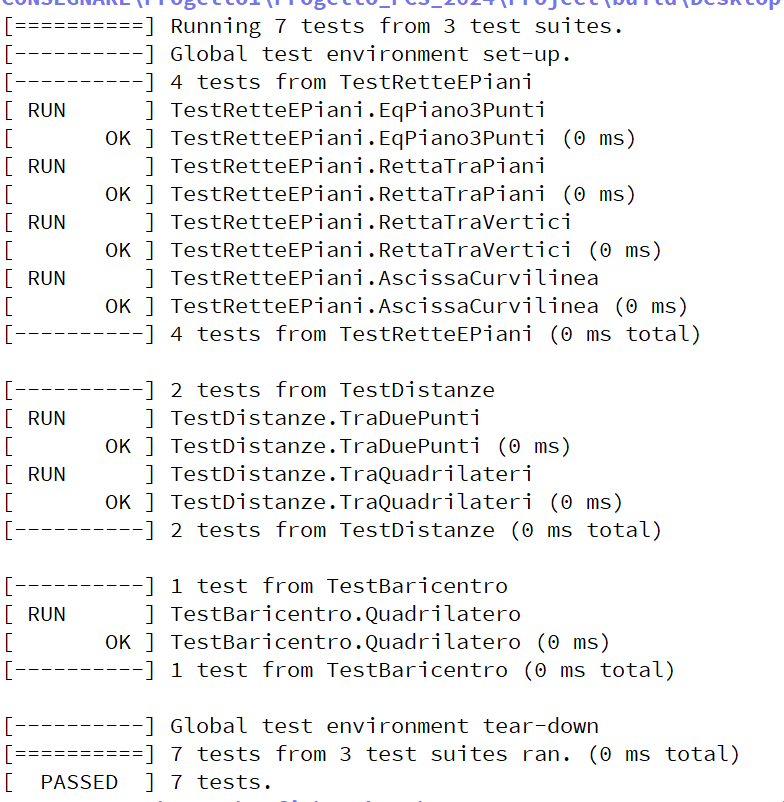
***void esportazione(Traces& traccia, Fractures& frattura)***

Questa funzione esporta i dati delle tracce e delle fratture in un file di testo denominato "Traces.txt".

Apre un file di output chiamato "Traces.txt" e controlla se l'apertura ha avuto successo. Se fallisce, mostra un messaggio di errore e termina la funzione. Per ogni traccia, scrive nel file di output l'identificatore della traccia, gli identificatori delle fratture coinvolte e le coordinate dei due vertici della traccia. Organizza poi un vettore frattura\_traccia che memorizza il numero complessivo di tracce per ogni frattura. Per ogni frattura, se ha almeno una traccia associata, scrive nel file di output l'identificatore della frattura, il numero di tracce associate e per ogni tipo di traccia (elencate in passante o non passante in ordine di lunghezza decrescente), elenca gli identificatori, i tipi e le lunghezze.

# **TESTING**

Sfruttando la libreria esterna dei GoogleTest si sono implementati dei test di unità per verificare la correttezza delle funzioni implementate all’interno del codice. Sfruttando un caso matematico semplice si è confrontato il risultato dato dalla funzione con il risultato ricavato manualmente su carta. Si sono creati tre raccoglitori: uno contiene i test relativi alle funzioni che lavorano sulla distanza, un altro testa la funzione che calcola il baricentro e il raccoglitore più popolato controlla la buona riuscita delle funzioni inerenti operazioni che lavorano su rette e pieni



# **CONCLUSIONI**

Il progetto “Discrete Fracture Network” ha affrontato efficientemente la modellazione delle fratture nelle rocce. I principali risultati e osservazioni emerse dal progetto (parte 1) sono riportate di seguito:

* Strutture dati:

Le strutture dati definite nel namespace FractureLib sono state scelte per bilanciare efficienza computazionale e semplicità di implementazione. La struct Fractures rappresenta un insieme di fratture all'interno del DFN. La struct Traces rappresenta le tracce generate dalle intersezioni delle fratture. L'uso di vector e array ha garantito una gestione dinamica e statica dei dati rispettivamente, permettendo un accesso rapido agli elementi e un'allocazione della memoria efficiente.

* Modellazione delle Fratture e delle Tracce:

Il sistema sviluppato rappresenta le fratture come poligoni planari nello spazio tridimensionale. Il sistema calcola le intersezioni tra le fratture, cioè le tracce, che sono segmenti di diverse lunghezze che abbiamo classificato in passanti, con entrambi gli estremi del segmento che giacciono sul bordo della frattura, e in non passanti, con almeno un estremo all'interno della frattura. Questo calcolo avviene tramite un processo che richiede algoritmi ben strutturati per garantire precisione, affidabilità ed efficienza. Per esempio, la funzione *valuta\_intersezione* ha dimostrato di essere efficace nel determinare le intersezioni tra fratture senza risolvere complessi sistemi lineari ogni volta. Inoltre, l'uso della decomposizione LU e QR e l’esclusione di casi particolari, come il parallelismo, ha reso il calcolo delle intersezioni e delle rette molto più robusto e preciso.

* Implementazione, valutazione e risultati:

L'implementazione del progetto è suddivisa in importazione dei dati, calcolo delle tracce ed esportazione dei risultati, ed ognuna di queste ha un ruolo cruciale nel garantire che il sistema funzioni correttamente. Inoltre, il sistema di testing con GoogleTest ha garantito che le unità logiche del codice fossero verificate e convalidate, assicurando l'accuratezza dei risultati ottenuti. Infine, il file Traces.txt generato come output contiene tutte le informazioni sulle tracce ordinate per lunghezza e tipo, offrendo una visione chiara delle intersezioni tra le fratture.

In conclusione, il progetto ha raggiunto gli obiettivi prefissati per la parte 1, offrendo una soluzione robusta e efficiente per la modellazione e l'analisi delle fratture nelle rocce.