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. Buffo Blin  G. Calabrese  S. Silvestro |

# **Sommario**

**1.** **INTRODUZIONE AL PROBLEMA** 2

**2.** **STRUTTURE DATI E DOCUMENTAZIONE UML** 3

**3.** **FUNZIONI E IMPLEMENTAZIONE** 6

**4.** **TESTING** 10

**5.** **CONCLUSIONI** 11

# **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.

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.

Inizialmente si era scelto di utilizzare una serie di mappe per tutti gli oggetti: gli identificatori erano la chiave e i dati di vertici o i coefficienti erano il valore della mappa. Successivamente si è preferito cambiare le strutture dati e utilizzare dei vettori, in quanto i vettori sono un oggetto meno costoso dal punto di vista computazionale in fase di compilazione quindi l’accesso ai dati risulta più rapido e meno costoso.

Il vettore è un oggetto della libreria standard che contiene dati memorizzati in maniera contigua. Si è deciso di utilizzare i vettori quando la dimensione di un oggetto varia durante l’esecuzione del programma, oppure quando è necessario aggiungere elementi in un momento successivo all’allocazione iniziale della memoria, sfruttando il metodo *push\_back*.

Sono stati utilizzati i vettori in quanto il costo di accesso al singolo elemento è un *O(1)* rispetto alle liste, che invece essendo memorizzate in maniera non contigua in memoria necessiterebbero dell’utilizzo dei puntatori e il relativo costo di accesso sarebbe un *O(n)*.

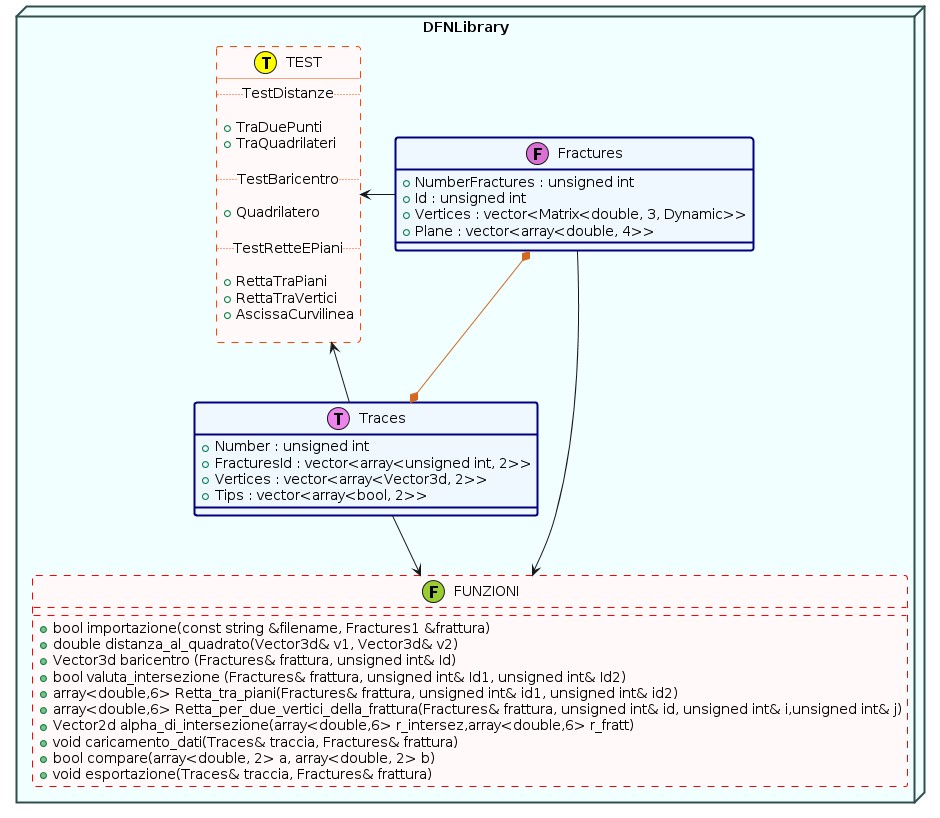
Gli array, anch’essi oggetti della libreria standard, presentano una dimensione statica, vale a dire che viene definita all’inizio del blocco di appartenenza e non varia nel tempo all’interno del codice, questo perché deve essere nota nel momento in cui avviene la compilazione. Inoltre visto che i dati memorizzati all’interno degli array risiedono in memoria in posizioni contigue l’accesso agli elementi di questa struttura risulta rapido ed efficiente.

Per queste motivazioni si è deciso di memorizzare alcuni oggetti, quali il numero dei coefficienti nell’equazione cartesiana del piano, il numero di fratture che partecipano alla creazione di una traccia e il numero di vertici di un segmento, all’interno di array di dimensione nota.

**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.



# **FUNZIONI E 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 sfruttando la funzione *getline()* per lavorare su una riga alla volta e poter gestire al meglio il salvataggio dei dati all’interno delle strutture dati scelte. Successivamente viene popolata la struct *Fractures*.

Inizialmente si utilizza un oggetto ifstream per aprire il file. Si legge poi il numero di fratture presenti e lo si memorizza nella variabile *numberFractures*. Per ogni frattura la funzione legge l’identificatore della frattura, il numero di vertici che la definiscono e memorizza le coordinate di quest’ultimi in una matrice.

Successivamente vengono calcolati i coefficienti dell'equazione del piano su cui giace ciascun poligono, sfruttando un’altra funzione, e memorizzandoli nell’oggetto *Plane*.

Dopo aver letto ed elaborato tutte le fratture presenti, la funzione chiude il file e restituisce *true* per indicare che l'operazione è stata completata con successo.

***array<double, 4> EqPiano (Fractures& frattura, unsigned int& Id)***

Grazie a questa funzione si ricavano i coefficienti reali ‘a, b, c, d’ dell’equazione cartesiana del piano in cui giace ogni poligono del sistema considerato.

L’equazione si presenta nella forma .

A questo scopo si utilizza la formula dell’equazione del piano passante per tre punti che impiega il calcolo del determinante della matrice A, poi posto uguale a zero. I tre punti considerati sono i primi tre vertici di ogni frattura, da cui il pedice nella matrice seguente.

***double distanza\_al\_quadrato (Vector3d& v1, Vector3d& v2)***

La funzione restituisce un numero reale che quantifica la distanza tra due punti nello spazio tridimensionale servendosi del Teorema di Pitagora. In questo caso si è scelto di non utilizzare l’elevamento a potenza, per lo stesso motivo per cui non si è calcolata la radice quadrata per non subire il costo computazionale associato a queste operazioni.

La formula utilizzata risulta essere:

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

Questa funzione serve per calcolare il baricentro delle fratture trattate. Dato *N* il numero di vertici, si lavora separatamente sulle singole coordinate per ciascuna delle tre dimensioni (x, y, z): ogni vertice contribuisce alla posizione del baricentro *G*. Sommando le singole coordinate di ogni vertice si procede poi dividendo per il numero di vertici, ottenendo così le coordinate del baricentro, poi salvate in un Vector3d. In formula matematica si è svolta la seguente operazione:

|  |  |  |
| --- | --- | --- |
|  |  |  |

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

Questa funzione valuta se due fratture possono intersecarsi, basandosi sulle posizioni reciproche che coppie di poligoni assumono tra loro.

Si comincia individuando 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.

Prendendo in esame coppie di poligoni, si calcolerà la loro distanza relativa al quadrato testando la distanza al quadrato tra i baricentri dei due poligoni contro il quadrato della somma dei raggi delle sfere circoscritte le due fratture, tenendo conto della tolleranza.

Studiando la distanza relativa si determina se le fratture potrebbero intersecarsi, in tal caso la funzione restituirà *true*, altrimenti le fratture sicuramente non si intersecheranno e la funzione restituirà *false*.

In questo modo si escludono 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 i coefficienti della retta di intersezione tra due piani definiti da fratture distinte e li salva all’interno di un array di dimensione 6, cioè pari al numero di coefficienti nell’equazione della retta. Essa è la retta su cui giacciono le tracce.

Si considerino la retta in forma parametrica:

e le equazioni di due piani: e

La direzione della retta di intersezione tra i due piani, cioè i coefficienti *‘a, b, c’*, sono determinati dal prodotto vettoriale dei loro vettori normali.

Considerata la matrice si calcolano i coefficienti della retta utilizzando il calcolo

del determinante, da cui ,

Invece i coefficienti *‘d, e, f’* sono le coordinate di un punto appartenente alla retta di intersezione.

Tali coordinate sono contenute nel vettore e si ottengono risolvendo un sistema lineare del tipo mediante il metodo di decomposizione LU, dove la matrice

e il vettore

***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. Si calcola tale retta per poter, in un momento successivo, andare a cercare i vertici delle tracce tramite intersezione con la retta di intersezione tra piani precedentemente trovata.

Dati i vertici e l’equazione della retta cercata è nella forma

Le componenti sono: , mentre i restanti coefficienti sono le coordinate di . Il parametro *t* è l’ascissa curvilinea della retta.

L’insieme dei coefficienti viene memorizzato in ordine in un array di dimensione 6.

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

Questa funzione calcola le ascisse curvilinee della retta di intersezione

e della retta passante per due vertici di una frattura e li salva nel vettore

Questi parametri vengono trovati risolvendo un sistema lineare del tipo , dove la matrice

e il vettore

Il sistema lineare viene risolto usando la decomposizione QR.

Vengono calcolate queste due ascisse curvilinee per poter determinare la posizione dei punti di intersezione tra le rette trovate in precedenza e poter definire quali tra questi punti di intersezione sono vertici di una traccia.

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

La funzione lavora su tutte le fratture e ad ogni iterazione considera coppie di poligoni. Essa sfrutta tutte le funzioni implementate in precedenza per giungere allo scopo finale del codice: trovare e caratterizzare le tracce generate da una serie di poligoni nello spazio per poi memorizzarle nella struttura *‘Traces’.*

Dopo aver verificato la possibilità che vi sia intersezione tra le due fratture considerate, si procede calcolando la retta generata dall’intersezione tra i piani contenenti fratture, escludendo la condizione in cui i piani risultano paralleli.

Solo nel caso in cui questa retta esiste, si vanno a calcolare i coefficienti delle rette passanti per coppie di vertici di ciascuna delle due fratture, cioè viene costruita una retta su ogni lato dei poligoni considerati.

Successivamente si valuta l’intersezione tra tali rette: ogni retta generata dai lati dei poligoni viene testata con la retta di intersezione tra i piani.

Si è utilizzato fino a qui un contatore per inserire all’interno di un array tutte le ascisse curvilinee delle rette di intersezione tra piani.

Tenendo conto della tolleranza, visto che si sta lavorando su dati di tipo double, si controlla che l’ascissa curvilinea appartenga all’intervallo [0, 1]: ciò permette di stabilire se il punto di intersezione appartiene o meno al segmento coincidente con il lato del poligono.

Se il contatore assume valore pari a 4, significa che sono state trovate esattamente 4 rette che si intersecano con la retta tra piani, due per ogni poligono. Per questo valore del contatore si procede studiando in dimensione 1 la posizione reciproca degli intervalli generati dalle ascisse dei poligoni. Se questi si sovrappongono allora si è trovata una traccia generata dai due poligoni considerati.

In tal caso si riconsiderano nuovamente i medesimi poligoni, aggiungendo un contatore per ciascuno, il cui valore equivale al numero di punti di intersezione giacenti sui lati di ogni poligono.

Si sono studiati tre casi possibili, con tre iterazioni distinte e definite dal valore dei contatori sopra citati:

* + Caso 1a: la traccia risulta passante per entrambi i poligoni.
  + Caso 1b: la traccia risulta passante per il primo poligono e non passante per il secondo poligono
  + Caso 2: la traccia risulta non passante per il primo poligono e passante per il secondo
  + Caso 3 la traccia risulta non passante per entrambi i poligoni

In ciascuno di questi casi, vengono memorizzati il tipo di traccia per ogni poligono, considerando che il valore booleano 0 indica una traccia passante e il valore booleano 1 una traccia non passante, e le informazioni sulla traccia, quali vertici e identificatori dei due poligoni che la generano.

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

Questa funzione è stata creata al fine di esportare i dati ottenuti in precedenza su un file di testo denominato *‘Traces.txt’*.

Per ogni traccia, nel file viene stampato il suo identificatore, gli identificatori delle fratture coinvolte e le coordinate dei suoi due vertici.

Successivamente all’interno del medesimo file viene considerata una frattura alla volta riportando alcune informazioni: il numero complessivo di tracce in essa presenti, il suo identificatore e per ogni traccia vengono elencati identificatore, tipo e lunghezza. Vengono inserendo prima le tracce passanti poi quelle non passanti, in ordine di lunghezza decrescente servendosi della funzione ***bool compare (array<double, 2> a, array<double, 2> b)*** e il metodo sort.

# **TESTING**

Sfruttando la libreria esterna dei GoogleTest si sono scritti dei test di unità per verificare la correttezza delle funzioni implementate all’interno del codice.

Per costruire i test si è partiti dalle formule matematiche, procedendo con prove pratiche su carta eseguendo i passaggi su un caso matematico semplice. Successivamente si è confrontato il risultato dato dalla funzione con il risultato ricavato manualmente.

Si sono create tre suite: una contenente i test relativi alle funzioni che lavorano sulla distanza, un’altra testa la funzione che calcola il baricentro e la suite più popolata controlla il buon esito delle funzioni inerenti operazioni che lavorano su rette e piani.

Potendo scegliere tra le macro *‘EXPECT’* ed *‘ASSERT’* si è scelto di utilizzare la seconda opzione in quanto in caso di fallimento, di anche solo un test, si è potuto procedere immediatamente alla correzione dello stesso.

In base al tipo dei dati che ci si aspettava di avere in output si sono utilizzate le macro *‘ASSERT\_EQ’* per tutte le funzioni che restituiscono array o vettori, *‘ASSERT\_DOUBLE\_EQ’* per la funzione sulla distanza tra due punti che restituisce un numero reale, infine *‘ASSERT\_TRUE’* per la funzione che valuta la possibilità dell’intersezione tra due fratture.

Su carta sono state eseguite numerose prove per valutare casistiche diverse e solo alcune di esse sono state impiegate nella verifica con i GoogleTest. Al termine delle varie prove è stato scelto un caso da riportare nel codice definitivo.

# **CONCLUSIONI**

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

* Strutture dati:

Le strutture dati definite nel namespace DFNLibrary 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.