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**

Il Discrete Fracture Network (DFN) è una metodologia utilizzata in geologia per modellare le fratture presenti nelle rocce.

Un DFN è costituito da un insieme di fratture rappresentate da poligoni planari che si intersecano nello spazio tridimensionale. Queste intersezioni sono chiamate tracce, segmenti di diverse lunghezze che possono essere classificate per passanti o non passanti. Una traccia passante ha entrambi gli estremi del segmento che giacciono sul bordo della frattura, mentre una traccia non-passante ha almeno un estremo all'interno della frattura stessa. Questo progetto ha l’obiettivo di identificare e classificare le tracce per ciascuna frattura.

La parte 1 del progetto consiste nel leggere un DFN da un file e calcolarne le tracce. Per ogni traccia si memorizzerà la coppia degli identificativi delle fratture che la definiscono e le coordinate tri-dimensionali dei punti che la identificano. Invece per ciascuna frattura, si differenzieranno le tracce in passanti e non-passanti, tramite un booleano [passante (false) oppure non-passante (true)]. Si ordineranno i due sottoinsiemi separatamente per lunghezza (decrescente). Il tutto verrà stampato in un file traces.txt.

La parte 2 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 sottopoligoni.

Un’attenzione particolare andrà nell'introdurre delle condizioni che escludano la probabilità che due fratture si intersechino, evitando così il costo relativo alla risoluzione dei sistemi lineari necessari per il calcolo delle tracce.

Infine, per la validazione dei risultati, verranno utilizzati dei test, implementando GoogleTest per verificare le varie unità logiche del codice.

# **STRUTTURE DATI E DOC. UML**

**2.1 STRUTTURE DATI**

In questa sezione verranno descritte le strutture dati utilizzate nel progetto, che influenzano direttamente le prestazioni computazionali e la facilità di implementazione del programma.

Per il progetto 1, nel namespace FractureLib sono definite due struct: Fractures e Traces.

La struct Fractures è stata progettata per rappresentare un insieme di fratture all’interno del DFN. Le sue componenti sono:

* NumberFractures: variabile di tipo unsigned int che tiene traccia del numero totale di fratture.
* Id: variabile di tipo unsigned int che rappresenta l’identificatore univoco per ogni frattura.
* Vertices: vettore di matrici contenente tutti i vertici. La matrice ha 3 righe (coordinate x, y, z) e un numero dinamico di colonne (il numero dei vertici è variabile).
* Piano: vettore di array di 4 elementi di tipo double, che rappresentano i coefficienti a, b, c, d dell'equazione del piano ax + by + cz + d = 0 su cui è situato il poligono.

La struct Traces è stata progettata per rappresentare le tracce generate dalle intersezioni delle fratture. Le sue componenti sono:

* NumberTraces: variabile di tipo unsigned int che tiene traccia del numero totale di tracce.
* Id: variabile di tipo unsigned int che rappresenta l’identificatore univoco per ogni traccia.
* Vertices: vettore di array contenente due vettori tridimensionali (che rappresentano le coordinate dei due vertici della traccia).
* Tips: vettore di array contenente due valori booleani (che indicano se la traccia è passante [false] o non-passante [true] per ciascuna delle due fratture che definiscono la traccia).

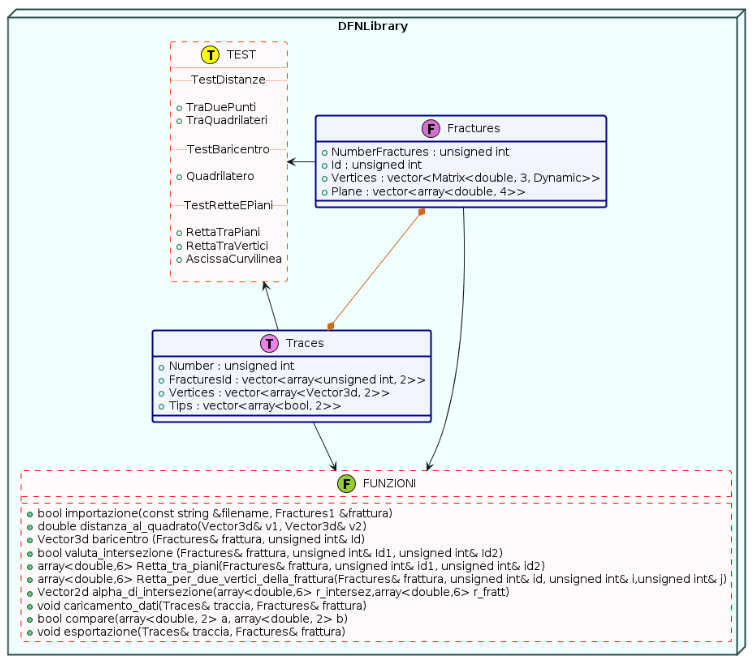
Per il progetto 2, nel namespace PolygonalLibrary è definita la struct PolygonalMesh, che è utilizzata per rappresentare una mesh poligonale composta da celle di diverse dimensioni:

* Cell0D: NumberCell0D, Cell0DId, Cell0DCoordinates (vettore composto da vettori che rappresentano le coordinate (x, y, z) di ciascun vertice).
* Cell1D: NumberCell1D, Cell1DId, Cell1DIdVertices (vettore composto da array che contengono gli identificatori dei vertici che compongono ciascun segmento).
* Cell2D: NumberCell2D, Cell2DId, Cell2DVertices (vettore composto da vettori che contengono gli identificatori dei vertici che compongono ciascun poligono), Cell2DEdges (vettore composto da vettori che contengono gli identificatori dei lati che compongono ciascun poligono).

Le strutture dati utilizzate sono state selezionate per i seguenti motivi:

* Vector: utilizzato quando la dimensione del vettore è dinamica, cioè può variare durante l’esecuzione del programma, e c’è bisogno di aggiungere o rimuovere elementi frequentemente.
* Array: utilizzato quando la dimensione dell’array è fissa. Utile perché offre accesso rapido ed efficiente ai suoi elementi.

**2.2 DOCUMENTAZIONE UML**



# **FUNZIONALITÀ ED IMPLEMENTAZIONE**

In questa sezione verranno descritte le principali funzionalità del progetto e le relative implementazioni.

Il codice, per il progetto 1, si divide in 3 parti principali, cioè, caricamento dati, calcolo delle tracce ed esportazione dei risultati ottenuti.

Le funzioni implementate sono:

* *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.

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

Questa funzione calcola la distanza al quadrato tra due punti tridimensionali v1 e v2 rappresentati da vettori Vector3d. È utile in molti algoritmi di geometria computazionale perché evita il calcolo della radice quadrata, rendendo il processo più efficiente.

* *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.

* *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:

x = coord\_retta[0] \* t + coord\_retta[3]

y = coord\_retta[1] \* t + coord\_retta[4]

z = coord\_retta[2] \* t + coord\_retta[5]

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 A\*x = b. 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 X = at+P dove t = (x2 – x1, y2 -y1, z2 -z1) 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 dimensioni 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 A\*x = b. 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.

* *bool compare(array<double, 2> a, array<double, 2> b);*

Questa è una funzione di confronto usata per ordinare array bidimensionali in base al secondo elemento.

* *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**

# **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.