Laurea Triennale in "Matematica per l'Ingegneria"

Anno Accademico: 2023-2024

Insegnamento: PROGRAMMAZIONE E CALCOLO SCIENTIFICO (03NMVMQ)

Titolare: Stefano BERRONE
Collaboratori: Matteo CICUTTIN, Gioana TEORA, Fabio VICINI

REPORT DI GRUPPO

Discrete Fracture Network (DFN)

CONSEGNA: 11 luglio 2024

A cura di

Matilde INGLESE (s286431)
Alessia MAINA (s297296)
Lorenzo NERI (s283148)

Politecnico di Torino

SOMMARIO

INTRODUZIONE

Il progetto assegnato ha come obiettivo quello di adottare strutture dati volte all'ottimizzazione della memoria e dell'efficienza computazionale, sviluppando un codice in c++ in grado di determinare le tracce di un DFN (vedere PRIMA PARTE) e, conseguentemente, di determinare i sotto-poligoni che si generano a partire da ogni frattura (vedere SECONDA PARTE).

Con DFN si intende il Discrete Fracture Network, ovvero un modello matematico e geometrico utilizzato per rappresentare le fratture in un materiale, tipicamente di roccia o un suolo, in modo dettagliato. Si tratta di fratture discrete di dimensione finita in 2D (segmenti di linea) e in 3D (dischi planari). Le intersezioni che si originano tra le fratture sono denominate tracce e si categorizzano in passanti, ovvero un segmento di intersezione che ha tutti e due gli estremi sul bordo, e non passanti, cioè un segmento che ha almeno un estremo collocato all'interno del poligono (e quindi 1 estremo sul bordo e 1 estremo interno al poligono oppure entrambi gli estremi all'interno del poligono). Tutti poligoni trattati nell'esecuzione del progetto sono convessi e si possono intersecare in due modi:

- Punto
- Segmento, ovvero il caso che sarà oggetto di studio

1 PRIMA PARTE : Determinazione delle tracce di un DFN

1.1 Definizione delle strutture del DFN

In primo luogo, è fondamentale definire due strutture dati distinte, nelle quali verranno memorizzate le informazioni necessarie sia per le fratture sia per le tracce, al fine di poter eseguire il calcolo delle tracce. Così facendo si ottiene un rapido accesso ai dati, invocando le varie strutture tramite reference.

1.1.1 Struttura delle FRATTURE

- ***unsigned int NumberOfFractures = 0***
  È una variabile di tipo intero privato del segno e serve per memorizzare il numero totale di fratture.
- ***vector<unsigned int> FractureId = { }***
  È un vettore di numeri interi privati del segno che memorizza gli identificatori univoci per ciascuna frattura.
- ***vector<unsigned int> NumberOfVertices = { }***
  È un vettore di numeri interi privati del segno che memorizza il numero totale di vertici per ciascuna frattura. Ogni elemento del vettore rappresenta il numero di vertici di una specifica frattura.
- ***vector<vector<Vector3d>> VerticesOfFractures = { }***
  E' un vettore che per ogni frattura memorizza un vettore, al cui interno vengono memorizzate le coordinate 3d che identificano i vertici della frattura , ovvero x, y e z

La scelta ritenuta più opportuna è stata quella di utilizzare i vettori rispetto agli array statici, indotta dal fatto che i vettori possono crescere o ridursi dinamicamente in base al numero di fratture e vertici.
Mediante la definizione di tale struttura, si può conseguire una efficiente elaborazione dei dati forniti in input, che sono presenti all'interno della directory denominata DFN.

1.1.2 Struttura delle TRACCE

- ***unsigned int NumberOfTraces = 0***
  Ugualmente a strFractures, è una variabile di tipo intero privato del segno che memorizza il numero totale di tracce.
- ***vector<unsigned int> TraceId = { }***
  È un vettore di numeri interi privati del segno che memorizza gli identificativi univoci per ciascuna traccia.
- ***vector<Vector2i> FractureIds = { }***
  È un vettore di 'vector2i, dove 'vector2i' rappresenta una coppia di interi. I due elementi rappresentano gli identificativi delle fratture che si intersecano per formare una traccia.
- ***vector< array<Vector3d, 2>> VerticesOfTraces = { }***
  È un vettore che per ogni traccia memorizza un array contenente al suo interno le coordinate 3d dei 2 estremi che identificano la traccia
- ***vector<array<bool, 2>> Tips = { }***
  È un vettore che per ogni traccia memorizza un array di dimensione 2 contenente i valori booleani che essa assume rispettivamente per la prima e per la seconda frattura
- ***vector<double> TraceLenght = { }***
  È un vettore di double che memorizza la lunghezza di ciascuna traccia. Ogni elemento del vettore rappresenta la lunghezza di una traccia specifica.
- ***vector<pair<double, unsigned int>> passing = { }***
  È un vettore che per memorizza le coppie lunghezza-id associate alle tracce passanti
- ***vector<pair<double, unsigned int>> notPassing = { }***
  È un vettore che memorizza coppie lunghezza-id associate alle tracce NON passanti

Mediante la definizione di tale struttura, si ricavano le informazioni principali che servono per definire tutti i parametri che una traccia deve possedere per eseguire le operazioni necessarie.

1.2 Lettura dei file contenenti informazioni inerenti alle fratture, del tipo *FR3_data.txt*

Avendo a disposizione i file di input, è possibile visualizzarne il formato e, di conseguenza, utilizzare le informazioni riportate al suo interno per definire le componenti delle fratture dichiarate nella strFractures. Per fare ciò, si utilizza la struttura **bool importListFractures**. Essa, dopo aver verificato che il file di input si apra correttamente e sia non vuoto, ha come obiettivo quello di importare il numero totale di fratture e, per ciascuna di essa, memorizzare l'Id in ordine crescente, il numero totale di vertici e, per ciascuno di essi, le relative coordinate in 3d. I differenti file sono

definiti all'interno della repository DFN e prendono in considerazione diverse casistiche di fratture (FR3, FR10, FR50, FR82, FR200, FR362).
Dopo aver effettuato l'importazione, per verificare che le fratture siano ben definite, si eseguono 2 test principali:

1. **bool testEdgesOfFracture:** per ogni frattura, verifica la presenza di lati tutti diversi da zero
2. **bool testVerticesOfFracture:** per ogni frattura, verifica la presenza di almeno 3 vertici (altrimenti se i vertici sono pari a 2 si tratta di un segmento, se il vertice è solamente 1 si tratta di un punto)

1.3 Calcolo delle tracce e relativa stampa su un file, del tipo *FR3_TracesResults.txt*

Come si nota nei file immagine forniti inizialmente, le fratture presentano una certa orientazione nello spazio e, di conseguenza, per verificare la loro effettiva intersezione, è utile avvalersi di una struttura particolare, denominata Bounding Sphere.
La **Bounding Sphere** o sfera circoscritta è una sfera minima che contiene completamente un oggetto geometrico tridimensionale, in questo caso una frattura. In particolare, fornito un insieme di punti, essa è definita come la sfera con il raggio minimo che contiene tutti i punti dell'insieme.
In questo caso di studio, si provvede a calcolare il centro come la media dei vertici forniti e successivamente, per ogni vertice, si stabiliscono la distanze tra i vertici e il centro, e si prende in considerazione la distanza massima.
L'applicazione di tale struttura, consente la definizione di **bool proximityOfFractures,** ovvero una funzione che testa se avviene o meno l'intersezione tra 2 fratture, contenute entrambe nella rispettiva Bounding Sphere: per intersecarsi, la distanza tra i loro centri deve essere minore o uguale alla somma dei loro raggi.

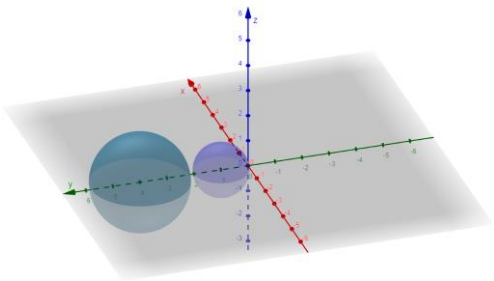

Figura 1: le bounding spheres non si intersecano

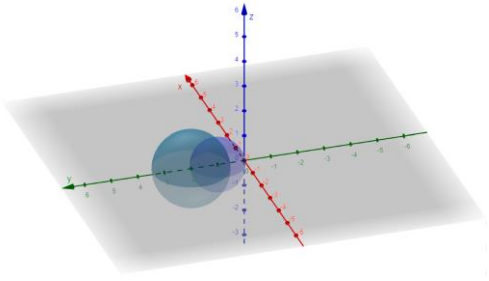

Figura 1: le 2 bounding spheres si intersecano

Più nel dettaglio, per controllare se un punto appartiene o meno ad una frattura, viene effettuata un'operazione caratteristica del Discrete Fracture Network, ovvero la **triangolazione**, che consente di rappresentare in modo dettagliato la geometria della frattura all'interno di questo modello.
Con essa, si fa riferimento alla suddivisione della superficie della frattura stessa in triangoli e per eseguire tale operazione, si definiscono 2 funzioni differenti:

1. **bool isPointInTriangle:** stabilisce che un punto P appartiene ad un triangolo ABC se la somma delle aree dei triangoli che esso forma con i vertici del triangolo (ovvero la somma delle aree di ABP, ACP e BCP) è pressoché pari all'area totale del triangolo

2. **bool isPointInFracture:** applica la funzione bool isPointInTriangle per ogni triangolo in cui è scomposta la frattura, creato a partire da un vertice comune e dall'iterazione degli altri due punti

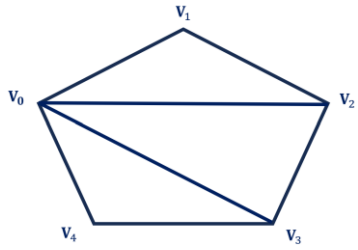

Figura 3: esempio di suddivisione di un poligono di n-vertici, il quale è scomposto in n-2 triangoli

Tali funzioni risultano utili al fine di determinare nel concreto la definizione della traccia.
Per la sua determinazione, si impiegano varie nozioni teoriche derivanti dalla Geometria Computazionale.
Facendo riferimento a quanto indicato durante l'insegnamento, due piani, $\boldsymbol{\pi_1}$ e $\boldsymbol{\pi_2}$, si intersecano in una retta $\boldsymbol{r}$ se si verificano le seguenti condizioni, in questo caso applicate al problema da risolvere:

- L'equazione del piano viene definita come di seguito:

$$\boldsymbol{\pi = \{x \in E^3 \mid x = \alpha_0 P_2 + \alpha_1 P_1 + (1 - \alpha_0 - \alpha_1) P_0 \, , \alpha = (\alpha_0, \alpha_1) \in \mathbb{R}^2\}} \quad (1)$$

- Si individua il versore tangente al piano per ogni piano che individua la frattura, ed è definito nel seguente modo a partire da 3 vertici consecutivi della frattura

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2} \quad (2)$$

  dove u è il vettore dal secondo punto al primo punto, mentre v è il vettore dal secondo punto al terzo punto. I vettori sono entrambi contenuti nel piano della frattura e sono perpendicolari tra di loro e, a loro volta, sono perpendicolari a **n**

- Per trovare la retta di intersezione $\boldsymbol{r}$, si deve calcolare il vettore tangente al piano individuato dai versori tangenti calcolati per ogni piano della frattura. Per fare ciò si usufruisce del prodotto vettoriale:

$$\mathbf{t} = \mathbf{n_1} \times \mathbf{n_2} \quad (3)$$

- Successivamente, si deve calcolare l'intersezione tra $\boldsymbol{\pi_1}$, $\boldsymbol{\pi_2}$ e un ulteriore piano $\boldsymbol{\pi_3}$, il quale è ortogonale a **t** e passante per l'origine del sistema di vettori:

  Definizione del **Punto**:

$$\mathbf{P}: \begin{cases} \mathbf{n_1^T P = d_1} \\ \mathbf{n_2^T P = d_2} \\ \mathbf{t^T P = 0} \end{cases} \quad (4)$$

  Definizione del **Sistema Lineare**:

$$\begin{pmatrix} \mathbf{n_{1,x}} & \mathbf{n_{1,y}} & \mathbf{n_{1,z}} \\ \mathbf{n_{2,x}} & \mathbf{n_{2,y}} & \mathbf{n_{2,z}} \\ \mathbf{t_x} & \mathbf{t_y} & \mathbf{t_z} \end{pmatrix} \begin{pmatrix} \mathbf{P_1} \\ \mathbf{P_2} \\ \mathbf{P_3} \end{pmatrix} = \begin{pmatrix} \mathbf{d_1} \\ \mathbf{d_2} \\ \mathbf{0} \end{pmatrix} \quad \text{ovvero } \mathbf{Ax = b} \quad (5)$$

dove la prima riga della matrice A è formata dal vettore $\mathbf{n_1}$, la seconda dal vettore $\mathbf{n_2}$ e la terza dal vettore $\boldsymbol{t}$.
dove il primo elemento del vettore b è costituito dal prodotto tra $\boldsymbol{n_1^T}$ e il centro della Bounding Sphere della prima frattura, il secondo elemento è costituito dal prodotto tra $\boldsymbol{n_2^T}$ e il centro della Bounding Sphere della seconda frattura e il terzo elemento è pari a 0.

- La soluzione di tale sistema esiste solamente nel caso in cui il determinante della matrice A sia diverso da 0, condizione che conseguentemente indica la non-complanarità dei vettori, ovvero che il prodotto misto deve essere diverso da 0:

$$\mathbf{t} \cdot (\mathbf{n_1} \times \mathbf{n_2}) \neq \mathbf{0} \tag{6}$$

- Ora, dopo aver stabilito che esiste l'intersezione tra i piani delle 2 fratture, è necessario, in base a essa, calcolare e verificare i due punti estremi della traccia per ogni frattura coinvolta.

1.4 Differenza tra le tracce passanti e NON passanti

Per identificare le 2 tipologie di tracce, è necessario introdurre l'utilizzo di una variabile booleana, denominata Tips, che assume valore FALSE se la traccia è PASSANTE, valore TRUE se la traccia è NON PASSANTE.
Più nel dettaglio, le tracce passanti e le tracce non passanti si distinguono nel seguente modo:

- **Traccia passante:** si riferisce a una linea che attraversa completamente una superficie o un oggetto; nel nostro caso, è identificata da un segmento risultante dall'intersezione di due fratture, avente entrambi gli estremi sul bordo della frattura in cui si trovano

- **Traccia non passante:** si riferisce a una linea che non attraversa completamente una superficie o un oggetto, ma rimane al suo interno; nel nostro caso, è identificata da un segmento risultante dall'intersezione di due fratture avente almeno un estremo collocato all'interno della frattura in cui si trovano, oppure entrambi gli estremi all'interno della frattura

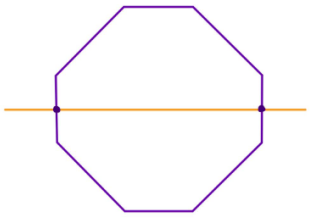

Figura 4: esempio PASSANTE

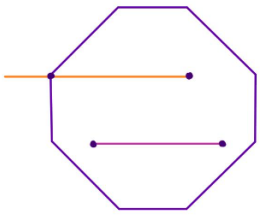

Figura 5: esempio NON PASSANTE

Per calcolare la tipologia delle tracce, viene utilizzata la funzione **void computeTypeTrace**, all'interno della quale si esegue un'iterazione su tutte le tracce, estraendone l'estremo iniziale e l'estremo finale.
Inizialmente, ogni traccia presa in considerazione viene definita "non passante", quindi il valore della variabile booleana Tips è impostato a True. Di seguito, per ciascuna traccia, si verifica se una delle estremità si trova su un lato della frattura e se l'altra estremità si trova su un lato diverso della stessa frattura, considerando tutti i lati delle frattura. Se entrambe le estremità si trovano su lati diversi, la traccia è considerata "passante", e quindi Tips assume il valore False.
Il risultato della classificazione (passante o non passante) viene memorizzato all'interno di vector<array<bool, 2>> Tips = { }.

Per calcolare la tipologia delle tracce, viene utilizzata la funzione **void computeTypeTrace**, all'interno della quale si esegue un'iterazione su tutte le tracce, estraendone l'estremo iniziale e l'estremo finale.
Inizialmente, ogni traccia presa in considerazione viene definita "non passante" sia per la prima frattura sia per la seconda frattura, quindi viene impostato array<bool, 2> currentTips = {true, true}. Di seguito, per ciascuna traccia, la funzione verifica se i suoi estremi sono su lati opposti, sia per la prima traccia, sia per la seconda traccia. Per farlo, si prende in considerazione il primo lato della frattura e si verifica se l'estremo della traccia si trova su questo lato. Se tale condizione è affermativa, allora si itera sui lati successivi della frattura per verificare se anche il secondo estremo si trova su uno dei lati. Nel caso in cui sia il primo estremo che il secondo estremo della traccia si trovano su lati differenti della frattura, allora la traccia è passante e il tips relativo a quella frattura assume valore "false".

1.4.1 Ordinamento delle 2 tipologie per lunghezza decrescente

Per ordinare ogni tipologia di traccia in modo decrescente, si ricorre all'implementazione dell'algoritmo **Merge Sort**, ovvero un algoritmo di ordinamento per fusione che opera in tempo $\boldsymbol{O(n\ log\ n)}$ utilizzando l'approccio divide et impera. Esso è caratterizzato da 3 particolarità: **decomposizione, ricorsione** e **ricombinazione**. Per questo, l'array viene suddiviso in sub-array di dimensione sempre più piccole. successivamente i sottoarray sono fusi tra loro fino a ricostruire l'array iniziale in ordine crescente.

Per implementare tale ordinamento, si esegue la funzione **void orderTraces**, nella quale:

- Tutte le tracce che hanno almeno un tips pari a false vengono inserite in vector<pair<double, unsigned int>> passing
- Tutte le tracce che hanno almeno un tips pari a true vengono inserite in vector<pair<double, unsigned int>> notPassing

Ogni traccia viene rappresentata come una coppia di lunghezza e indice. Entrambi i vettori, passing e notPassing, vengono ordinati per lunghezza decrescente utilizzando l'algoritmo di ordinamento Merge Sort, se contengono almeno un elemento.

1.4.2 Esecuzione stampa di risultati su file, uno del tipo *Traces_Info_FR3.txt* e uno del tipo *Traces_Tips_FR3.txt*

Traces_Info_FR3.txt

```
# Number of Traces
2
# TraceId; FractureId1; FractureId2; X1; Y1; Z1; X2; Y2; Z2
0; 0; 1; 8.0000000000000004e-01; 0.0000000000000000e+00; 0.0000000000000000e+00; 8.0000000000000004e-01; 1.0000000000000000e+00; 0.0000000000000000e+00
1; 0; 2; 0.0000000000000000e+00; 5.0000000000000000e-01; 0.0000000000000000e+00; 3.1618370000000001e-01; 5.0000000000000000e-01; 0.0000000000000000e+00
```

Traces_Tips_FR3.txt

```
# FractureId; NumTraces
0; 2
# TraceId; Tips; Lenght
0; false; 1.0000000000000000e+00
1; true; 3.1618370000000001e-01


# FractureId; NumTraces
1; 1
# TraceId; Tips; Lenght
0; false; 1.0000000000000000e+00


# FractureId; NumTraces
2; 1
# TraceId; Tips; Lenght
1; true; 3.1618370000000001e-01
```

1.5 Implementazione dei GoogleTest

Seguendo le istruzioni, si è provvisto ad effettuare i GoogleTest per testare le varie unità logiche presenti all'interno del codice. Come base per i test, sono stati utilizzati i dati forniti all'interno del file **FR3_data.txt**, in quanto permette dei controlli più accurati.

1.6 Definizione della tolleranza utilizzata

- ***'machineEpsilon'***
  numeric_limits<double>::epsilon() è una funzione che fa parte della libreria '<limits>' e identifica la cosiddetta epsilon di macchina, il cui valore corrisponde a 2.22045e-16.
  MachineEpsilon è la tolleranza utilizzata nella maggior parte dei casi in cui il codice ne richiede l'utilizzo. Essa è utile per verificare se due numeri 'double' sono uguali nella precisione della macchina. Inoltre è importante in quanto rappresenta la precisione minima della macchina per i numeri in virgola mobile.

- ***'defaultTolerance'***
  pow(10,-10) è una funzione che restituisce 10-10, ovvero 0.0000000001.
  defaultTolerance è la tolleranza utilizzata nello specifico per verificare se un punto si trova all'interno di un triangolo e per verificare se un punto si trova all'interno di un segmento. Viene usata per confronti geometrici, per avere una tolleranza più specifica che tiene conto degli errori di arrotondamento accumulati.

Entrambi questi valori servono per garantire che i calcoli siano eseguiti in modo accurato, limitando la presenza di errori all'interno delle funzioni del codice.