Report

s301956
s294481
s297324

18 giugno 2024

1 Parte 1

1.1 Strutture create

Abbiamo creato due strutture per questo progetto:

1. **Fracture** La struttura per descrivere le fratture è caratterizzata da:
   - Un ID identificativo (*unsigned int*)
   - L'insieme dei suoi vertici (*MatrixXd*)
   - Due vettori per contenere tutte le tracce (gli identificativi) della frattura, separate tra passanti e non passanti (*vector<unsigned int>*)
2. **Trace** Le caratteristiche di una traccia sono
   - Un ID identificativo (*unsigned int*)
   - La sua lunghezza (*double*)
   - I suoi estremi (*Matrix<double, 3, 2>*)
   - Le due fratture che la generano (*unsigned int*)

Queste due strutture rappresentano rispettivamente una singola fratture e una traccia. Si è scelto di utilizzare un vettore di strutture personalizzate, invece che una struttura generalizzata contente dei vettori a sua volta, per leggibilità del codice e per facilitare la gestione in memoria.

1.2 Lettura dal file

Le fratture lette dal file vengono salvate in un vettore di **Fracture**. La scelta del vettore è dovuta al fatto che

- Possiamo ridimensionare il vettore appena letto il numero di fratture dal file.
- Quando andremo a cercare le tracce intersecando ogni frattura con tutte le altre avremo bisogno di una struttura nella quale l'accesso ai dati sia immediato.

1.3 Tolleranza

Per gestire le tolleranze, abbiamo deciso di definire una tolleranza di default (*standard*), e di chiederne due all'utente, una per il caso 1D (tol1D_user) e l'altra per il caso 2D (tol1D_user). Per decidere quale tolleranza utilizzare, abbiamo scelto

- tol1D = $\max\{standard; \text{tol1D\_user}\}$
- tol2D = $\max\{standard; \text{tol2D\_user}; (\text{tol1}D)^2\}$

1.4 Cicli sulle fratture

Per calcolare le tracce iniziamo col ciclare sul vettore delle fratture, confrontando ogni elemento con tutti quelli successivi. Questo garantisce che non si ripeta più volte lo stesso controllo ma che ogni coppia di fratture venga controllata una sola volta. Per gestire efficacemente la memoria, prima di iniziare il ciclo, vengono inizializzati due vettori di liste: uno conterrà le tracce passanti per una data frattura, mentre l'altro quelle non passanti. Non conoscendo quante fratture hanno tracce al loro interno entrambi, i vettori vengono inizializzati con una dimensione pari alle fratture totali; se la lista contenuta nell'i-esima posizione del vettore sarà vuota, allora la frattura i-esima non avrà tracce. Successivamente, per semplicità e poiché sarà necessario ordinare le tracce di ogni frattura, andiamo a salvare ogni lista in un vettore ed aggiungerlo alla struttura della Frattura.

A questo punto, in ogni iterazione del ciclo, andiamo a calcolare, dove presente, un'intersezione. Per escludere a priori alcune casistiche, prima di calcolare le intersezioni adiamo a definire delle condizioni di esclusione.

1.5 Condizioni di esclusione

Per evitare di fare controlli superflui, andando a cercare intersezioni tra fratture distanti tra loro, abbiamo studiato due tipi di condizione di esclusione, confrontandone poi l'efficienza.

1.5.1 Bounding box

La bounding box è (nel caso di figure piane) il rettangolo di area minore che le contiene. Questo viene ottenuto prendendo di tutte le componenti le coordinate minime e massime della frattura (combinando poi per ottenere anche gli altri vertici). Confrontiamo quindi le bounding box delle due fratture: se il minimo di una si trova sopra il massimo dell'altra, sicuramente non saranno possibili intersezioni. Questo procedimento si usa simmetricamente prima per il massimo della prima e il minimo della seconda e poi viceversa.

1.5.2 Sfere circoscritte

Per entrambe le fratture consideriamo i rispettivi baricentri e la maggiore distanza dai vertici, e chiamiamo questi valori max_1 e max_2. Se la distanza tra i baricentri è maggiore della somma tra max_1 e max_2, non proseguiamo nella ricerca delle intersezioni.

1.5.3 Differenze di efficienza tra condizioni di esclusione

Avendo spiegato l'implementazione di queste due condizioni, è d'obbligo domandarci quale delle due escluda più casi e, inoltre, quale dei due risulti essere il più efficiente a livello di costo computazionale. Per quanto riguarda l'efficienza, le "bounding box" sono meno costose rispetto all'altro metodo. Questo perché, per usare i baricentri, è necessario trovare le diagonali di lunghezza maggiore e questo richiede due cicli sulla prima frattura considerata e altrettanti sulla seconda. Vediamo ora il numero di casi esclusi dalle due condizioni:

Numero fratture	Bounding box	Baricentri
3	1	0
10	10	2
50	392	279
82	3291	867
200	5255	3009
362	65016	16982

Come si può notare dalla tabella riportata, il primo metodo esclude, in generale, molti più casi del secondo. Per questo motivo abbiamo scelto di adottarlo per il nostro codice. Un esempio di un caso in cui "bounding box" esclude e "baricentri" no è riportato all'interno di uno dei GoogleTest.

1.6 Intersezioni tra fratture

Per risolvere il problema di verificare se due fratture si intersecano e, nel caso, calcolare gli estremi della traccia, abbiamo fatto i seguenti passaggi:

1. Per ciascuna frattura, consideriamo il piano che la contiene calcolando il vettore normale al piano normalizzato. Per fare questo, calcoliamo il prodotto vettoriale tra i vettori che collegano il primo vertice con il secondo e con il terzo vertice.

2. Se i due piani non sono paralleli, andiamo a considerare la retta di intersezione in forma vettoriale (scriviamo $r : P = pointOnLine + \overline{direction} * k$ con $k \in \mathbb{R}$). La direzione della retta può essere calcolata come prodotto vettoriale tra le normali ai piani trovate prima. Per trovare un punto sulla retta, intersechiamo il piano che contiene una delle due fratture (noi abbiamo scelto la prima) con una retta che giace sull'altro (e che non sia parallela al piano). L'intersezione piano-retta si ottiene nella seguente maniera: Sia $\pi : ax + by + cz + d = 0$ l'equazione del piano con $n = (a, b, c)$ vettore normale al piano e $r : P = P_0 + \vec{v} * t$ dove $P_0 = (x_0, y_0, z_0), v = (v_x, v_y, v_z)$ l'equazione della retta. Il punto di intersezione è l'unico punto della retta che soddisfa l'equazione del piano

$$
\begin{aligned}
a(x_0 + v_x t) + b(y_0 + v_y t) + c(z_0 + v_z t) + d = 0 \\
ax_0 + v_x at + by_0 + v_y bt + cz_0 + v_z ct + d = 0 \\
t(av_x + bv_y + cv_z) + ax_0 + by_0 + cz_0 + d = 0 \\
t = -\frac{n \cdot P_0 + d}{n \cdot v}
\end{aligned}
$$

poiché $d = -n \cdot Z$ dove Z è un punto del piano, otteniamo infine

$$t = -\frac{n \cdot (P_0 - Z)}{n \cdot v} \tag{1}$$

Nel nostro caso quindi abbiamo

- n normale al piano della prima frattura
- A è un punto qualsiasi del piano, prendiamo il primo vertice della prima frattura
- P_0 è un punto della seconda frattura
- v è il vettore direzione da P_0 al vertice successivo

Dobbiamo assicurarci che n e v non siano paralleli ($n \cdot v \neq 0$), quindi partiamo considerando il primo vertice della seconda frattura, verifichiamo se è parallelo alla retta di intersezione e, nel caso, prendiamo il lato successivo. In caso contrario, il calcolo di ($n \cdot v \neq 0$) ci sarà comunque utile per il calcolo in (1). Poiché le fratture non sono degeneri, due lati consecutivi non possono essere paralleli.

3. Se le due fratture si intersecano, l'intersezione avverrà sicuramente sulla retta che abbiamo trovato. Andiamo quindi ad intersecare la retta con i lati della frattura, fermandoci non appena troviamo due intersezioni. Se non abbiamo trovato intersezioni, concludiamo che le fratture non si intersecano, altrimenti ripetiamo lo stesso procedimento sulla seconda frattura. Nuovamente, se non abbiamo trovato due intersezioni, concludiamo che non ci sia la traccia, in caso contrario possiamo proseguire. Vediamo come trovare il punto di intersezione tra due rette complanari: siano $r_1 : P = PR + \vec{dir} * k$ e $r_2 : Q = A + \vec{v} * t$, e cerchiamo il valore di t tale per cui $P = Q$.

$$\begin{aligned} A + \vec{v} * t &= PR + \vec{dir} * k \\ (A + \vec{v} * t) \times \vec{dir} &= (PR + \vec{dir} * k) \times \vec{dir} \\ A \times \vec{dir} + t * \vec{v} \times \vec{dir} &= PR \times \vec{dir} \\ t * \vec{v} \times \vec{dir} &= (PR - A) \times \vec{dir} \\ t &= \frac{((PR - A) \times \vec{dir}) \cdot (\vec{v} \times \vec{dir})}{\|\vec{v} \times \vec{dir}\|^2} \end{aligned}$$

4. Identifichiamo adesso le due coppie di intersezioni frattura-retta con la loro posizione sulla retta: poiché tutti i 4 punti soddisfano l'equazione della retta $r : P = pointOnLine + \overline{direction} * k$, andiamo a risolvere e otteniamo due segmenti, diciamo $[a, b]$ e $[c, d]$.

$$\begin{aligned} pointOnLine + k * \overline{direction} &= P \\ k * \overline{direction} &= P - pointOnLine \\ k &= \frac{(P - pointOnLine) \cdot direction}{\|direction\|^2} \end{aligned}$$

5. La traccia è l'intersezione tra i due segmenti. Rimane quindi solo da studiare la posizione relativa dei 4 punti per trovare gli estremi della traccia.

1.7 Ordinamento

Per ogni frattura, andiamo a considerare le sue tracce, prima le passanti e poi le non passanti. Dal momento che le tracce sono salvate in dei vettori, abbiamo accesso all'algoritmo sorting nella sua efficienza (mediamente) maggiore. La funzione sorting ha, al massimo, un costo computazione di $O(n\log(n))$, che offre la massima efficienza possibile per un algoritmo di ordinamento senza informazioni aggiuntive sui vettori. Poiché per ogni frattura è stato salvato solo l'ID delle tracce passanti prima di ordinare i vettori è necessario costruire dei vettori temporanei di pair, ID lunghezza, ordinando quest'ultimi è poi possibile ricostruire il vettore di soli ID ordinato.

2 Parte 2

Al fine di suddividere i compiti al meglio per sfruttare il tempo a disposizione, abbiamo deciso di trattare separatamente le due parti del progetto. Questa scelta, fatta per ragioni pratiche, ha influenzato l'ottimizzazione finale. In particolar modo, nella prima parte sarebbe stato possibile salvare i lati in cui avvenivano le intersezioni delle tracce passanti (informazione necessaria nella parte due per generare i sotto poligoni per queste ultime) al momento del calcolo delle intersezioni, senza dover ricalcolare molti valori. Anche per questo motivo la seconda parte risulta più grezza e meno ottimizzata nell'ottica del progetto complessivo, tuttavia su alcuni aspetti è stata riportata comunque la dovuta attenzione.

2.1 Taglio lungo le tracce

Abbiamo iniziato prendendo il vettore delle fratture e per ognuna di esse siamo andati a spostare le tracce passanti e non, in un unico vettore. Questa scelta, seppur non ottimale, ci ha permesso di poter creare un'unica funzione per il taglio.

A questo punto, per generare i sotto poligoni siamo andati a risolvere il problema adottando un approccio ricorsivo:

1. Prendiamo il poligono salvato in una matrice, un vettore di tracce e un contatore che segna la posizione nel vettore della traccia per cui vogliamo "tagliare".

2. Selezionata la traccia, andiamo a verificare se i bounding box della traccia e del poligono si intersecano (al fine di trascurare le tracce che non sono all'interno del sottopoligono), successivamente calcoliamo le intersezioni tra la retta passante per la traccia e i lati del poligono come fatto nella parte 1, salvando questa volta anche il lato del poligono su cui avviene questa intersezione. Come menzionato in precedenza, avremmo potuto ricavare questa informazione già nel punto 1, salvando, nel caso di tracce passanti, anche i lati agli estremi della traccia. Avendo trattato le due parti separatamente, è necessario ora ripetere i conti.

3. Affinché la traccia sia contenuta nel poligono, l'intersezione tra il prolungamento della traccia e il lato del poligono deve o avvenire sulla traccia stessa, oppure i due punti di intersezione devono essere da versi opposti rispetto agli estremi della traccia. Questo metodo per verificare che la traccia sia all'interno di un poligono risulta computazionalmente pesante, tuttavia permette di ricavare contemporaneamente i punti di intersezione. Inoltre risulta molto affidabile e robusto anche con valori di tolleranza molto piccoli.

4. Nel caso in cui il poligono non contenga la traccia richiamo la funzione ricorsiva sul poligono, aggiornando il contatore e passando alla traccia successiva

5. Trovati i lati su cui avviene l'intersezione, definiamo due matrici in cui andiamo a salvare i vertici del poligono originale e i punti di intersezione. Per mantenere l'ordine anti-orario andiamo a salvare nella prima matrice tutti i punti precedenti al lato dell'intersezione nella prima matrice e quelli dopo il lato della seconda intersezione, mentre nella seconda i vertici compresi tra i lati delle due intersezioni. In questo modo abbiamo trovato i due sotto-poligoni generati dal taglio della traccia e possiamo andare a richiamare la funzione ricorsivamente su entrambi i nuovi sotto-poligoni generati, passandogli il contatore aumentato di uno per selezionare la traccia successiva.

6. Quando abbiamo controllato che il poligono non contenga nessuna traccia di quelle successive a quella che lo ha generato possiamo andare a salvare il poligono in una lista di matrici.

2.2 Mesh

Dopo aver ottenuto tutti i sotto-poligoni, siamo andati a salvare le informazioni ottenute in una mesh. Abbiamo quindi deciso di creare un'unica mesh per tutte le fratture invece di farne una per ogni frattura. Sebbene ciò differisca leggermente da quanto chiesto nella consegna, abbiamo pensato potesse essere significativo per il progetto, poiché funzionale a rappresentare lo scopo iniziale del modellizzare il *discrete fracture network*. Infatti, utilizzando un'unica mesh, è possibile salvare una sola volta punti e lati condivisi tra più fratture, evitando quindi una ridondanza. Questa è la struttura di mesh utilizzata:

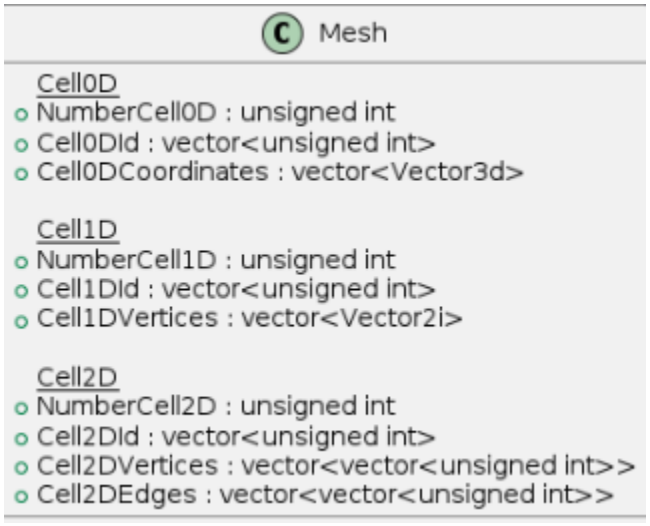

Figura 1: Struttura della mesh

Per fare questo abbiamo utilizzato le seguenti strutture:

- unordered map con chiave Vector3d e valore id

- unordered set con una pair di id

e definito le seguenti funzioni di hashing

- Vector3dHash
- pair_hash

Queste strutture, grazie all'unicità dei loro elementi e al metodo *insert*, ci hanno permesso di capire quando un punto non era ancora stato inserito nella mesh e, in caso contrario, di recuperare il suo id. La caratteristica peculiare di queste strutture che non salvano l'ordine di inserimento, garantisce un' elevata efficienza computazionale che, nel caso migliore (in cui non ci siano ripetizioni tra i valori del'hashing), permette di controllare che un elemento non sia già presente e, nel caso, di inserirlo con un costo di $O(1)$, arrivando, nel caso peggiore, ad un costo di $O(n)$. Questa scelta, seppur non sia la migliore per il file con poche fratture ed intersezioni, risulta molto efficiente nel file da 200 fratture, caso in cui altre strategie richiedevano tempi di esecuzioni notevolmente maggiori.

Dopo aver definito dei vettori per salvare le informazioni necessarie alla mesh e aver riservato in memoria una stima dello spazio necessario, abbiamo fatto un loop sulla lista dei sotto poligoni generati dal taglio e abbiamo inserito ogni punto nell'unordered map con un ID che corrisponde al contatore con cui il punto è stato trovato, e inserito il punto nella mesh solo quando insert.second risulta *true*. Un ragionamento analogo è stato fatto anche per i lati e l'unordered set. Per ogni sotto-poligono invece è stato aggiunto un Id nel vettore di Celle2D, e nella corrispettiva posizione del vettore Cell2DVertices, è stato aggiunto il contatore del punto se già presente, o in alternativa il valore a cui punta insert.first.