Progetto PCS: Discrete Fracture Network

S. Berrone, M. Cicuttin, G. Teora e F. Vicini

Azzolini Adriano 294253
Hniki Ayoube 299921
Trapanotto Biagio Luca 295894

A.A. 2023/2024

Indice

1 Definizione strutture dati

Per la risoluzione del problema si sono utilizzate due strutture distinte: Fractures, che memorizza i dati relativi alle fratture in esame, e Traces, che memorizza i risultati dei calcoli delle intersezioni tra le fratture.

1.1 struct Fractures

La prima struttura contiene i seguenti membri:

- FractureNumber: intero non negativo, descrive il numero di fratture del file in input;
- VerticeNumber: vettore di dimensione FractureNumber di interi non negativi, contiene alla posizione i il numero di vertici della frattura i-esima;
- Id: vettore di dimensione FractureNumber di interi non negativi, contiene alla posizione i l'identificativo della frattura i-esima;
- Coordinates: vettore di dimensione $\sum_{i=0}^{\text{FractureNumber}}$ VerticeNumber(i) di vettori Eigen 3x1 di double, contiene le coordinate dei vertici di ogni frattura;
- Segments: vettore di dimensione FractureNumber di vettori di dimensione VerticeNumber(i) di vettori Eigen 3x1, suddivide per ogni frattura e per ogni segmento le rette direzionali;
- ListVertices: vettore di dimensione FractureNumber di vettori di dimensione VerticeNumber(i) di interi non negativi, contiene per ogni frattura una elenco degli identificativi dei suoi vertici;
- VerticesCoordinates: matrice di dimensione 3x$\sum_{i=0}^{\text{FractureNumber}}$ VerticeNumber(i) di double. Ogni colonna rappresenta le coordinate di un vertice.

1.2 struct Traces

Contiene i membri:

- TracesNumber: intero non negativo, descrive il numero di tracce trovate;
- CoupleIdTips: mappa con chiavi gli Id delle tracce e valori "True" se traccia passante, "False" se interna;
- TracesPoints: Vettore di dimensione (2*TracesNumber) di vettori Eigen 3x1 di double, contiene le coordinate degli estremi delle tracce;
- IdFractures: vettore di dimensione NumberTraces di vettori Eigen 2x1 di double, alla posizione i contiene gli identificativi delle due fratture che formano la traccia i-esima;
- CoupleFracturesTraces: mappa che ha per chiavi gli identificativi delle fratture e per valori gli Id delle tracce che forma;
- LengthsTrace: vettore di dimensione TracesNumber di double, alla posizione i contiene la lunghezza della traccia i-esima.

1.3 Considerazioni sulla scelta delle strutture

Dei costrutti disponibili in C++, si è scelto di lavorare con due *struct* poiché, a differenza delle *class* l'accessibilità predefinita è pubblica. La scelta di utilizzare due struct Fractures e Traces distinte separa e mette in risalto le caratteristiche delle due entità e permette una più ordinata risoluzione del problema.
Per la scelta dei membri si sono adottate le seguenti strategie:
Si sono preferiti i vettori agli array, poiché le dimensioni, dovendo essere lette da file, non sono note

a priori; inoltre, a differenza delle liste, la posizione degli elementi può essere sfruttata in modo che coincida con l'identificativo della frattura, facilitando l'accesso ai dati da manipolare.
Si è inoltre deciso di memorizzare le coordinate dei vertici sia in un vettore che in una matrice: il primo memorizza vettori Eigen, che permettono la risoluzione dei sistemi lineari, mentre la seconda è utilizzata come input in Paraview.
Si sono utilizzate infine mappe dove serviva sfruttare la relazione chiave-valore.

2 Importazione dati: ImportData(filepath, fracture)

Per l'importazione dei dati si è definito il metodo "*ImportData*", chiamato nel main con argomenti *filepath* la stringa del percorso del file di input e *fracature* un'istanza della struttura Fractures. In quest'ultimo metodo le coordinate vengono inizialmente inserite nell'ordine in cui sono scritte nel file, facendo attenzione agli elementi che non sono seguiti da ";".
Ottenuto il vettore che contiene tutte le coordinate dei vertici, vengono utilizzati due iteratori per suddividerlo: l'iteratore interno "m" incrementa a ogni ciclo di un valore pari al numero dei vertici associato alle fratture. L'iteratore esterno "d" incrementa ad ogni ciclo di un'unità. Le terne così ottenute vengono appese al vettore *Coordinates*.

3 Definizione delle tracce

Una volta memorizzate e catalogate le fratture, si passa allo studio di come queste si intersecano e successiva caratterizzazione delle tracce che formano: si distingueranno casi di tracce passanti e non passanti.

3.1 DefineTraces(fileOutput, fracture, trace)

Per lo studio delle intersezioni tra fratture, si costruiscono e si risolvono sistemi lineari.
Il primo passo è il calcolo dei vettori u e v, generatori dei piani che contengono le fratture. Da questi è possibile calcolare il versore normale al piano n, definito come $n = \dfrac{u \times v}{\|u\|\|v\|}$, e lo scalare d, definito come $d = n \cdot \mathrm{P}_0$, con P_0 punto di intersezione tra u e v.
Si passa ora al calcolo di t, vettore tangente alla retta di intersezione tra piani, definito come $t = n_i \times n_j$, con i e j gli indici di due fratture.
Si hanno ora tutti i dati per la costruzione del sistema lineare: la matrice A ha per righe i due versori normali e il vettore tangente: $A = \begin{pmatrix} \cdots & n_i & \cdots \\ \cdots & n_j & \cdots \\ \cdots & t & \cdots \end{pmatrix}$ e il vettore d ha come componenti d_i, d_j e 0 $b = \begin{pmatrix} d_i \\ d_j \\ 0 \end{pmatrix}$ Si passa ora alla risoluzione di tale sistema: per verificare che ci sia intersezione tra i due piani si controlla che il determinante della matrice A sia diverso da 0. Tuttavia, essendo in algebra finita, la condizione diventa det(A)<-tol || det(A)>tol, dove tol rappresenta una tolleranza, scelta pari a 1e-10. Sotto questa condizione si passa al calcolo del sistema lineare $Ax = b$ dove x rappresenta un punto appartenente alla tangente, tramite fattorizzazione LU con pivoting parziale.
Successivamente bisogna controllare le intersezioni tra t e i segmenti s_k delle fratture, calcolati con il metodo *ComputeSegments*(trattato in seguito). Per far ciò si definiscono:

- r punto appartenente alla tangente trovato sfruttando l'equazione parametrica r=x+c·t con c pari a 3
- w direzione della retta contenente x e un estremo del segmento.

- A_3^t matrice 3x3 che contiene w, il segmento e il vettore r-x: $A_3^t = \begin{pmatrix} \cdots & w & \cdots \\ \cdots & s_k & \cdots \\ \cdots & r-x & \cdots \end{pmatrix}$

Per vedere che ci sia intersezione si verificano due condizioni:

- Complanarità: -tol<det(A)<tol
- Incidenza tra t e il segmento: $s_k \times (r-x) <$ -tol || $s_k \times (r-x) >$ tol

Nel caso in cui si verifichino entrambe le condizioni, si costruisce la matrice A_2 3x2, che contiene il segmento della frattura ed r-x: $A_2 = \begin{pmatrix} \vdots & \vdots \\ s_k & r-x \\ \vdots & \vdots \end{pmatrix}$ Si calcola il sistema lineare $A_2 \times$coeff=w tramite fattorizzazione LU con pivoting totale e si trovano i coefficienti α e β. Se tol<α<tol+1 il punto di intersezione appartiene al segmento in questione e viene considerato come un potenziale estremo della traccia. Sarà quindi memorizzato in un vettore *intersT*.
Una variabile ausiliaria "a" tiene conto del numero di punti trovati: nel caso se ne trovino due per la stessa frattura, il ciclo si interrompe, evitando il calcolo di ulteriori sistemi lineari, che potrebbero trovare intersezioni esterne alle fratture.
Se l'intersezione tra le due fratture genera 4 punti, si procede con lo studio della possibile traccia: si calcola la lunghezza dei segmenti aventi come estremi i punti trovati usando il metodo *ComputeLengths*(trattato in seguito).
Le lunghezze calcolate in precedenza sono del tipo l_{ij} dove i e j corrispondono alle posizioni dei punti nel vettore *intersT*, il quale ha nelle prime due posizioni i punti di intersezione trovati considerando la frattura i e nelle successive posizioni i punti di intersezione trovati considerando la frattura j.
A questo punto si distinguono i seguenti casi:

- Caso l_{01}<=tol || l_{23}<=tol: la lunghezza della possibile traccia è minore della tolleranza scelta e quindi non valutabile;
- Traccia Passante: l_{02}<=tol && l_{13}<=tol oppure l_{03}<=tol && l_{12}<=tol dove ogni punto di intersezione trovato considerando la frattura i coincide con un rispettivo punto di intersezione trovato considerando la frattura j. Gli estremi della traccia coincidono con i punti trovati considerando la stessa frattura e la lunghezza della traccia è equivalente alla lunghezza del segmento che ha generato i punti;
- Traccia interna: caso in cui vi è soltanto un punto di intersezione, trovato considerando la frattura i, che coincide solamente con un punto di intersezione trovato considerando la frattura j. Ad esempio l_{02}<=tol && l_{13}>tol;
  Per il calcolo degli estremi viene fatta attenzione alle posizioni dei punti: solo nella condizione in cui, preso un punto interno, i due punti più distanti si trovano entrambi alla sua destra o alla sua sinistra (condizione che si verifica valutando il segno del loro prodotto scalare) vi è una traccia con estremi i due punti interni;
- Un altro caso è quando i punti di intersezione trovati per le fratture i e j sono diversi. In questo caso si scartano gli estremi della possibile traccia di lunghezza massima: i restanti due delimitano la traccia di interesse se generano un segmento con lunghezza > tol;

Per ogni casistica, inoltre, viene aggiornata la mappa *CoupleFractureTraces*, utile a suddividere per ogni frattura gli id delle tracce che genera. In questo modo si ha per ciascuna frattura un vettore contenente le lunghezze delle tracce associate, successivamente ordinate tramite il metodo "*Sorting*" (trattato in seguito). Il metodo si conclude stampando i risultati nel modo richiesto sul file "Tracce".

3.2 ComputeSegments(fracture)

Per il calcolo dei segmenti si è utilizzato un vettore *StartIndex* che memorizza la posizione dell'inizio di ogni frattura: ad ogni ciclo viene inserito nel vettore un elemento che tiene conto del numero di vertici già considerati. A questo punto i segmenti vengono generati e appesi al vettore *Segments*, divisi per fratture.

3.3 ComputeLenghts(a,b)

Per il calcolo della lunghezza delle tracce, dati due estremi a e b, si è utilizzata la formula:

$$l_{ab} = \|a - b\|$$

4 Ordinamento tracce: MergeSort

Il mergeSort è un algoritmo di Ordinamento che usa la tecnica del "Divide et Impera". Consiste nel dividere il vettore ricorsivamente a metà fino ad ottenere degli elementi. Questi vengono poi ricombinati con procedimenti ricorsivi tramite il metodo *Merge*, che li ordina in modo decrescente. Tale algoritmo ha un costo computazionale pari a O(nlog(n)).
Questo costo è ricavato tramite il teorema fondamentale delle ricorrenze: nel caso del MeregeSort la relazione di ricorrenza è

$$T(n) \leq \begin{cases} c_0 & \text{se n} \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases} \text{[1]}$$

Considerando $\alpha = 2$ (perché vengono eseguiti due procedimenti ricorsivi), $\beta = 2$ (perché il problema viene suddiviso in due parti, ciascuna di dimensione $\frac{n}{\beta}$) e $f(n) = n$ (perché vengono eseguiti n confronti e i costi di decomposizione e di ricombinazione sono costanti). Si verifica la condizione aggiuntiva del teorema: $\alpha f(\frac{n}{\beta}) = \gamma f(n)$.
Si ha quindi $2\frac{n}{2} = \gamma n$, vera soltanto per $\gamma = 1$.
Si conclude che la complessità del problema non cambia, e quindi il costo è pari a O(nlog(n)) come affermato.
Per la risoluzione del problema si è scelto l'algoritmo MergeSort poiché il suo costo computazionale è il migliore tra i costi degli algoritmi di ordinamento e risulta particolarmente efficiente per grandi insiemi di dati.

5 Google Tests

Per i Test da effettuare è stata generata una serie di dati, elaborati su carta in modo da ottenere dei valori aspettati da confrontare con i risultati dei metodi del codice.
Si sono effettuati i seguenti test :

- *TestComputesLengths*: verifica la correttezza del metodo *ComputeLengths*.
  Esempio:

```
Vector3d a(0.0, 0.0, 0.0);
Vector3d b(1.0, 0.0, 0.0);
double length = ComputeLengths(a, b);
EXPECT_DOUBLE_EQ(length, 1.0);
```

- *TestSorting*: verifica il metodo *Sorting*. Di seguito il codice del Test:

```
vector<double> C = {1, 4, 3, 2, 6 ,5};
vector<double> vet = {6,5,4,3,2,1};
Sorting(C);
EXPECT_EQ(C, vet);
```

- *BasicTest*: Verifica la correttezza del metodo *DefineTraces*. Di seguito il codice del Test:

```
Fractures fracture;
Traces trace;
trace.TracesNumber = 0;
fracture.FractureNumber = 4;
fracture.VerticeNumber = {4, 4, 4, 4};
fracture.Id = {1, 2, 3, 4};
fracture.Coordinates = {Vector3d(0, 0, 0), Vector3d(0, 0.5, 0),
Vector3d(-0.5, 0.5, 1), Vector3d(-0.5, 0, 1), Vector3d(-0.5, 0, 0.5),
Vector3d(1, 0, 0.5), Vector3d(1, 0.5, 0.5), Vector3d(-0.5, 0.5, 0.5),
Vector3d(-1, -0.5, 1.5), Vector3d(1.5, -0.5, 1.5), Vector3d(1.5, 1, 1.5),
Vector3d(-1, 1, 1.5), Vector3d(1, 0, 1), Vector3d(1, 1.5, 1),
Vector3d(1, 1.5, 1.5), Vector3d(1, 0, 1.5)};

ComputeSegments(fracture);
DefineTraces("output.txt", fracture, trace);
EXPECT_EQ(trace.TracesNumber, 2);
EXPECT_EQ(trace.LengthsTrace[0], 0.5);
EXPECT_EQ(trace.LengthsTrace[1], 1.0);
```

- *TestParaview*: Verifica la correttezza dell'elaborazione dei dati da passare per l'esportazione su Paraview, generando un file che rappresenta le fratture che vengono date in input. Di seguito il codice del Test:

```
Fractures fracture;
fracture.ListVertices = {{0,1,2,3\textbraceright, {4,5,6,7}, {8,9,10,11},
{12,13,14,15}};
vector<vector<unsigned int>> triangles; VectorXi materials;
fracture.FractureNumber = 4;
fracture.VerticeNumber = {4, 4, 4, 4};
fracture.Coordinates = {Vector3d(0, 0, 0), Vector3d(0, 0.5, 0),
Vector3d(-0.5, 0.5, 1), Vector3d(-0.5, 0, 1), Vector3d(-0.5, 0, 0.5),
Vector3d(1, 0, 0.5), Vector3d(1, 0.5, 0.5), Vector3d(-0.5, 0.5, 0.5),
Vector3d(-1, -0.5, 1.5), Vector3d(1.5, -0.5, 1.5), Vector3d(1.5, 1, 1.5),
Vector3d(-1, 1, 1.5), Vector3d(1, 0, 1), Vector3d(1, 1.5, 1),
Vector3d(1, 1.5, 1.5), Vector3d(1, 0, 1.5)}
Gedim::UCDUtilities exporter;
GedimInterface(fracture, triangles, materials);
exporter.ExportPolygons("./polygons\textunderscore4TEST.inp",
fracture.VerticesCoordinates, triangles,{},{}, materials);
```

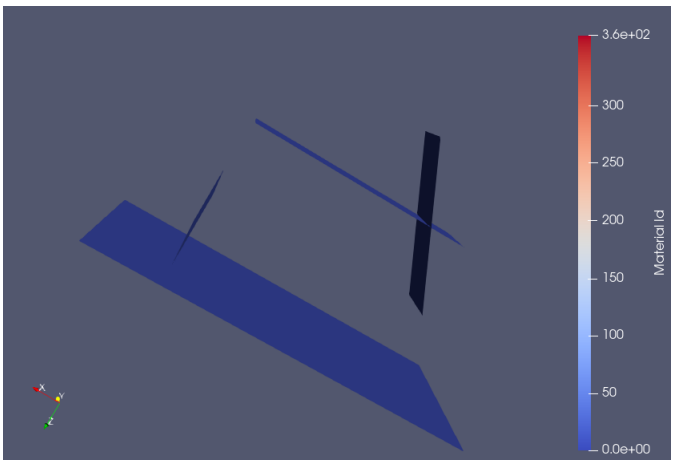
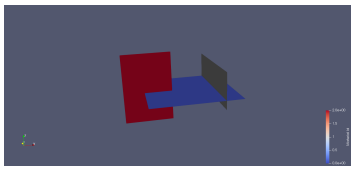


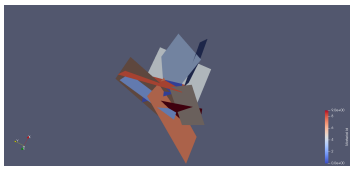
Figura 1: Test Paraview

6 Paraview

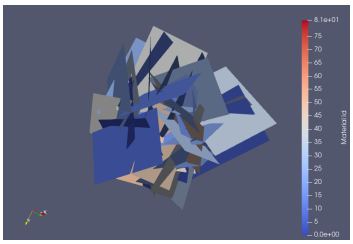
Per un'interpretazione visiva dei risultati ottenuti si è infine scelto di esportare il dataset sul software Paraview. Per ottenere un formato compatibile con il software, i dati sono stati manipolati attraverso il metodo *GedimInterface*. In particolare, avviene una triangolazione delle fratture e per ciascuna viene memorizzato in quanti triangoli è stata scomposta. Inoltre, per ogni triangolo si memorizzano gli Id dei suoi vertici. I dati generati vengono passati alla funzione *ExportPolygons*, che genera un file .inp leggibile da Paraview.
Di seguito le rappresentazioni visive dei dataset proposti:



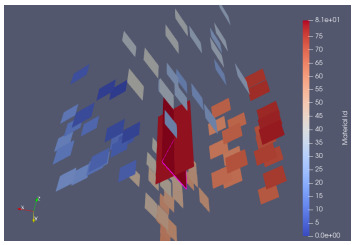
(a) 3 fratture



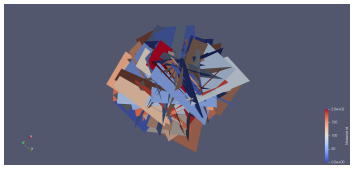
(b) 10 fratture



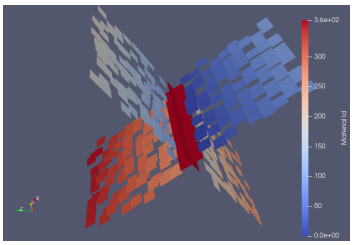
(c) 50 fratture



(d) 82 fratture



(e) 200 fratture



(f) 362 fratture

Riferimenti bibliografici

[1] P. Crescenzi. *Strutture di dati e algoritmi. Progettazione, analisi e visualizzazione*, page 61. Pearson, 2012.