Progetto PCS 2024

Simone Domenico Morandi, Elisabetta Pautasso

Parte I

Determinazione delle tracce di un DFN

I.1 Scelta della tolleranza geometrica

Per tutto il codice si è scelto di utilizzare due tolleranze geometriche. Per le distanza in una dimensione si è scelta come tolleranza $\epsilon = 10^{-14}$, in modo da tenersi abbastanza lontano dalla precisione di macchina ed evitare eventuali cancellazioni numeriche. Per le distanze in due dimensioni (ad esempio i calcoli dei prodotti vettoriali), si è scelta invece una tolleranza $\tau = 10^{-12}$.

I.2 Definizione delle strutture

Prima di poter calcolare le tracce di un DFN, è necessario definire le strutture dati all'interno delle quali andremo a memorizzare i dati necessari. In questo modo si può facilmente accedere ai dati in modo efficiente, chiamando poi le varie strutture tramite reference. Si definiscono due strutture, una per le fratture e una per le tracce.

I.2.1 Prima struttura: Fractures

Questa struttura contiene tutte le informazione contenute nel file iniziale.

```
struct Fractures
{
    unsigned int N_frac;
    vector<unsigned int> frac_id;
    vector<unsigned int> N_vert;
    vector<vector<Vector3d>> frac_vertices;
```

```
    vector<pair<vector<unsigned int>,vector<unsigned int>>>
    trace_type;
};
```

Come si evince dai nomi dei componenti della struttura, vengono salvati in appositi vettori: l'id di ciascuna frattura, il numero di vertici che possiede, le coordinate dei vertici e il numero di fratture totali, che è dato direttamente nel file iniziale. La posizione del vettore esterno corrisponde all' n-esima frattura. L'ultimo attributo della struttura rappresenta un oggetto che è alquanto utile per la stampa dei risultati di questa parte del progetto, ma anche per la sezione iniziale della seconda parte. Il vettore esterno ha dimensione pari al numero delle fratture, per ogni frattura viene salvata una coppia contenente due vettori di interi; il primo vettore (**Id**) contiene l'id delle tracce che appartengono all'n-esima frattura, mentre il secondo (vettore **Tips)** contiene la caratteristica della traccia corrispondente.
Si riconosce che tale oggetto può essere leggermente inefficiente per quanto riguarda l'ordinamento e la sua struttura in generale, ma rende di più facile accesso e comprensione le caratteristiche legate a ciascuna traccia riferita a ciascuno poligono a cui appartiene.

I.2.2 Seconda struttura: Traces

Questa struttura ha il compito di contenere tutte le informazioni relative alle tracce che verranno calcolate.

```
struct Traces
{
    vector<unsigned int> traces_id = {};
    vector<vector<unsigned int>> traces_gen = {};
    vector<vector<Vector3d>> traces_points = {};
    vector<double> traces_length = {};
}
```

In ordine, vengono memorizzati: gli id delle tracce, la coppia di fratture generatrici in base al loro id, le coordinate di inizio e di fine della traccia e la lunghezza.
Per entrambe le strutture si è scelto di utilizzare i vettori della libreria *Eigen* per gestire le coordinate tridimensionali, in quanto sono già implementate funzioni essenziali per la manipolazione geometrica quali la differenza fra vettori, il prodotto scalare e vettoriale. Per gestire il resto, invece, si è scelto di utilizzare i vettori della libreria standard.

I.3 Lettura del file

Ora è possibile leggere il file contenente le informazioni relative alle fratture e riempire la struttura *Fractures*. Nel *main* vengono inizializzate la struttura per

le fratture e la stringa contenente il percorso del file; vengono poi passate alla funzione *importFractures* che eseguirà la lettura del file, riempiendo la struttura. Conoscendo il formato del file, è stato possibile leggere riga per riga tramite *getline* e ricavare tutte le informazioni richieste. All'interno di *importFractures* viene anche inizializzato l'attributo *trace_ type* con un numero di vettori pari al numero delle fratture. È stato scelto di fare ciò in quanto, in generale, è improbabile che in un DFN una frattura non possieda nessuna traccia, perciò rimane più conveniente inizializzare tutte le posizioni del vettore di tuple. Alla fine, la funzione restituisce *true* se tutto è andato a buon fine.

I.4 Calcolo delle tracce

A questo punto si hanno tutte le informazioni relative alle fratture ed è possibile calcolare le tracce. Nel *main* viene chiamata la funzione *Find_ Traces*, che prende come parametri di input una struttura del tipo *Fractures* e una del tipo *Traces*. La funzione *Find_ Traces* opera nel seguente modo:

1. Per prima cosa viene inizializzato lo spazio in memoria per la struttura delle tracce. Dato N il numero delle fratture, al più tutte le fratture avranno un'intersezione con le $N-1$ fratture rimanenti e ci saranno al massimo $\frac{N^2}{2}$ tracce. Successivamente viene fatta una scrematura delle fratture prima di incominciare il calcolo delle tracce. Si verifica che i *Bounding Box* delle due fratture prese in esame si intersechino. Il Bounding Box è definito come il parallelepipedo formato dalla coordinata massima e dalla coordinata minima di un poligono. In questo modo si evita di incominciare il calcolo della traccia tra due fratture che in primis non si intersecano. Ovviamente il Bounding Box dà una stima approssimata per difetto del poligono che contiene e la verifica dell'intersezione di due Bounding Box può dare origini a falsi positivi; tuttavia queste casistiche saranno individuate e scartate più avanti nel codice.

2. Se le due fratture si intersecano, si procede a calcolare i piani che contengono le fratture e memorizzarli in un *Vector4d* di Eigen, tale vettore contiene i parametri a, b, c, d del piano. Viene fatto un controllo per verificare che i due piani trovati non siano paralleli, evitando così il caso di intersezione di due fratture complanari.

3. Si calcola la traccia. Per ogni lato della prima frattura, si calcola la sua retta di estensione e si risolve un sistema lineare formato dalle due equazioni dei piani contenenti le fratture e dai due piani la cui intersezione rappresenta la retta di estensione. Se il sistema ha soluzione e la soluzione ricade all'interno del Bounding Box del primo poligono, essa viene salvata in un vettore di *Vector3d*. Si procede allo stesso modo per il secondo poligono. Dato che la

retta contenente la traccia seziona la frattura in due parti, ci si aspetta di trovare sempre quattro punti in totale. Se i punti sono uguali a due a due, allora la traccia è passante per entrambe le fratture. Se i punti trovati sono tutti distinti, si procede ordinando il vettore di *Vector3d* secondo l'ascissa curvilinea e considerando come traccia quella formata dai punti centrali (quelli in posizione 1 e 2), questa operazione equivale al calcolo della differenza tra due segmenti: quello che ha come vertici i primi due punti appartenenti alla prima frattura e quello contenente gli ulteriori due punti appartenenti alla seconda frattura. Dopo aver controllato che entrambi i punti appartengano a entrambi i Bounding Box delle fratture, si può procedere a verificare se la traccia è passante per una delle due fratture. Ciò viene fatto calcolando se il punto appartiene a uno dei lati della traccia tramite la funzione *check_pass*. In base alle varie casistiche viene completata la struttura *Traces* e l'attributo *trace_type*.

I.4.1 Ordinamento di *trace_type*

A questo punto è necessario ordinare per passante/non passante tutte le tracce trovate e, per ciascuna caratteristica, ordinare le tracce per lunghezza crescente. Ciò viene fatto dalla funzione *sort_Traces_type*. Per ogni coppia di vettore di interi contenuta in *trace_type*, a meno che non sia già ordinata, viene chiamata la funzione *sort_pair*, che è in grado di ordinare un vettore tramite il metodo *sort* e trasferire le modifiche effettuate su un secondo vettore. In questo modo la corrispondenza **Id-Tips** rimane invariata. La prima chiamata della funzione ordina il vettore *Tips* e trasferisce i cambiamenti sul vettore degli id, successivamente viene identificato l'indice dell'ultima traccia passante (identificate con 0), che chiameremo *lastZeroIndex*, e vengono creati dei vettori con le lunghezze corrispondenti agli id delle tracce passanti e di quelle non passanti. Infine viene di nuovo chiamata la funzione *sort_pair* che ordina il vettore delle lunghezze, trasferendo i cambi sul vettore **Id**.
Qui si può notare la non perfetta ottimizzazione del processo di ordinamento, ma tutto sarà riguadagnato nella stampa delle caratteristiche delle tracce e sopratutto nella determinazione dei sotto-poligoni.

I.5 Esportazione dei file

Successivamente al calcolo delle tracce, si chiamano nel *main* due funzioni che compilano i due file di output richiesti dalla consegna, tali funzioni sono *Export_traces_Info*, che prende le informazioni dalla struttura *Traces* e *Export_traces_Type*, che legge l'attributo *trace_type* delle struttura *Fractures* e stampa su file i suoi dati.

I.6 GoogleTest

Come richiesto per questa prima parte, sono stati eseguiti i *GoogleTest* per tutte le funzioni utilizzate, ad eccezione di *Export_Paraview* in quanto viene utilizzato

del codice esterno. Per quasi tutte le funzioni si sono testati casi validi e invalidi, controllando la corretta gestione di questi ultimi. Per testare le funzioni si sono utilizzati i dati contenuti nel file *FR3_ data.txt* poiché di immediata verifica grafica tramite Geogebra.

I.7 Risultati

Tutti i file con i risultati ottenuti e la relativa documentazione sono disponibili qui. Si riporta qui di seguito delle rappresentazioni grafiche create tramite il software *Paraview©* dei file che presentano più tracce in proporzione al numero di fratture e delle relative tracce.

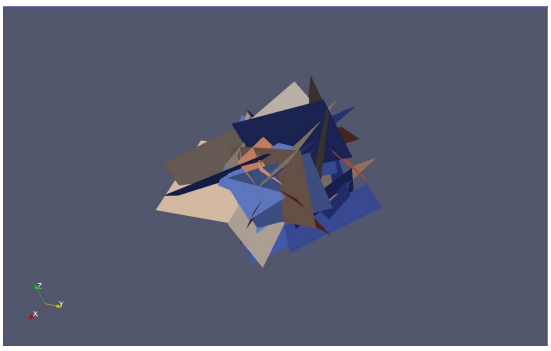

Figura 1: DFN con 50 fratture.

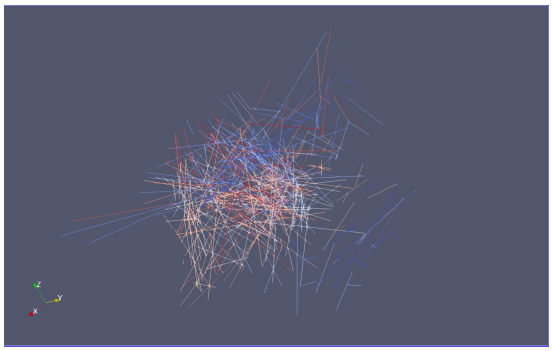

Figura 2: Tracce relative al DFN.

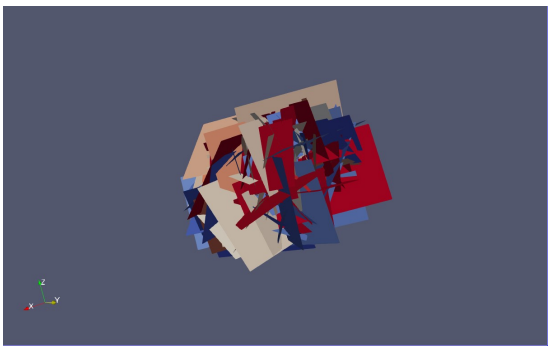

Figura 3: DFN con 200 fratture.

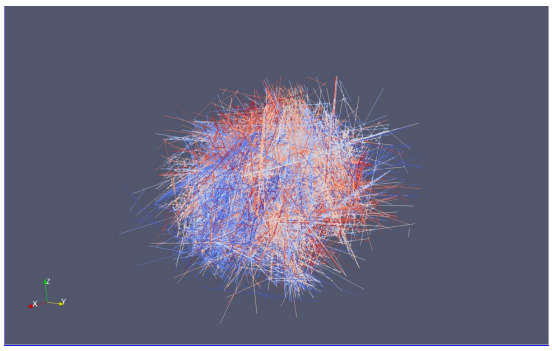

Figura 4: Tracce relative al DFN.

Parte II
Determinazione dei sotto-poligoni generati per ogni frattura

II.1 Taglio delle fratture

Una volta memorizzate tutte le tracce e il rispettivo Tips per ogni frattura, si può procedere con il taglio. Poiché il taglio di una frattura genera un albero binario, si è deciso di utilizzare una coda in modo tale da salvare i figli generati da un taglio e proseguire con le operazioni di taglio su questi ultimi. Una volta concluso il taglio, i sotto-poligoni coincideranno con le foglie dell'albero binario ottenuto. Tutto questo viene gestito dalla funzione *cutPolygons* che richiama al suo interno altre due funzioni *extendTracesToEdges* e *subPolygons*.

La funzione *cutPolygons* opera nel seguente modo:

1. Per ogni frattura, si utilizza una coda che viene inizializzata con i vertici della frattura solamente se questa ha almeno una traccia. Qui l'attributo *trace_type* risulta molto conveniente, in quanto si può accedere con costo costante a tutte le tracce appartenenti a una frattura.

2. Una volta inizializzata la coda, inizia un ciclo in modo tale da visitare tutte le tracce di quella frattura.

3. Da qui si inizia a visitare ogni elemento nella coda verificando ogni volta che la traccia sia interna e che non coincida con un lato del poligono.

4. Superati questi controlli, si può procedere al taglio vero e proprio richiamando la funzione *subPolygons* che agisce nel modo seguente:

   (a) Innanzitutto, si estende la traccia ai lati del poligono tramite la funzione *extendTracesToEdges* che restituisce un vettore di dimensione 6 che contiene al suo interno: gli estremi della traccia, le coordinate dei vertici del primo lato che contiene la prima coordinata della traccia e infine quelle del secondo lato che contiene la seconda coordinata della traccia, cioè

traccia[0]	traccia[1]	lato1[0]	lato1[1]	lato2[0]	lato2[1]

   (b) Dopodiché, si controlla che la dimensione del vettore restituito sia effettivamente 6, altrimenti si restituisce un vettore vuoto.

(c) Superato questo controllo, si passa alla costruzione dei nuovi sotto-poligoni tramite una variabile di flag: si parte costruendo il sotto-poligono 1 e si aggiungono le coordinate dei vertici del poligono fino a che non si arriva alla posizione di inizio del lato 1, in questo caso il flag diventa vero e vengono inserite, nel sotto-poligono 1, le coordinate della traccia nell'ordine corretto. Si procede poi alla costruzione del sotto-poligono 2 in maniera analoga al precedente e, una volta che si raggiunge la posizione di inizio lato 2, il flag torna ad essere falso e si completa la costruzione del sotto-poligono 1.

(d) Una volta completata la costruzione dei due sotto-poligoni, viene restituita la coppia formata dal sotto-poligono 1 e sotto-poligono 2.

5. Si procede poi al controllo che i due sotto-poligoni non siano vuoti, in modo tale da poterli inserire nella coda e continuare a ripetere questo procedimento.

6. Una volta che si è concluso il ciclo sugli elementi della coda e sulle tracce relative a quella frattura, vengono salvati gli elementi rimasti nella coda che costituiscono i sotto-poligoni finali derivanti dal taglio.

II.2 Risultati

Tutti i file con i risultati ottenuti e la relativa documentazione sono disponibili qui. Si riporta qui di seguito delle rappresentazioni grafiche create tramite il software *Paraview©* dei DFN che presentano più tracce in proporzione al numero di fratture e dei relativi sotto-poligoni, insieme al DFN3, mostrato come caso base.

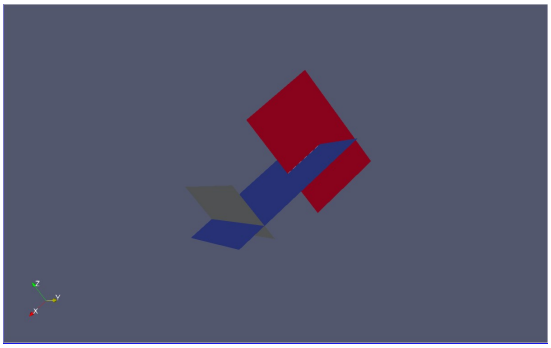

Figura 5: DFN con 3 fratture.

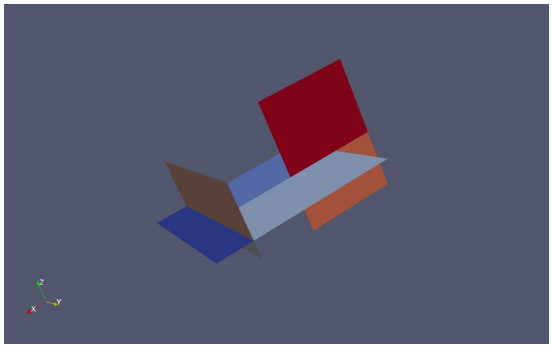

Figura 6: Sotto-poligoni relativi al DFN.

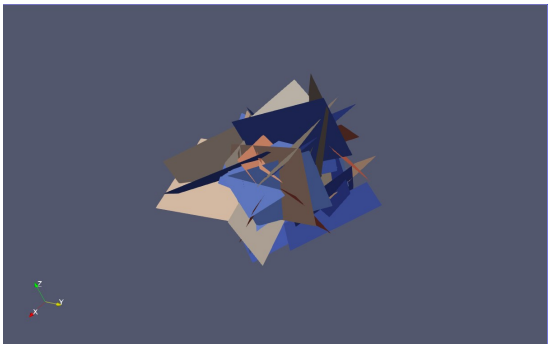

Figura 7: DFN con 50 fratture.

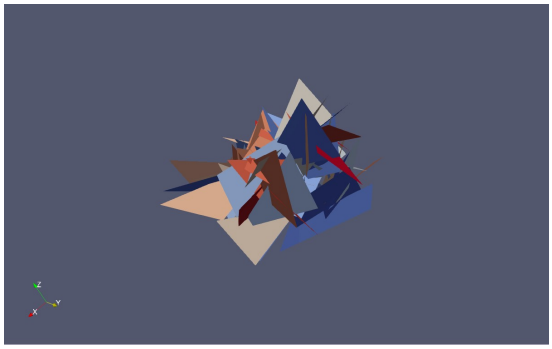

Figura 8: Sotto-poligoni relativi al DFN.

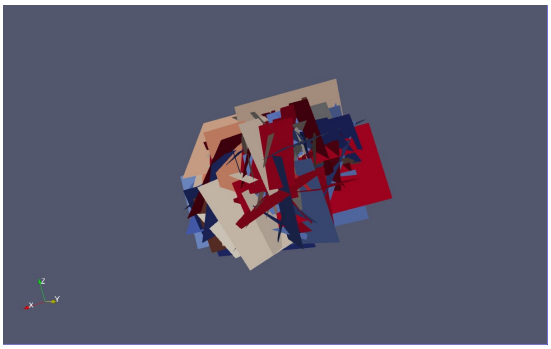

Figura 9: DFN con 200 fratture.

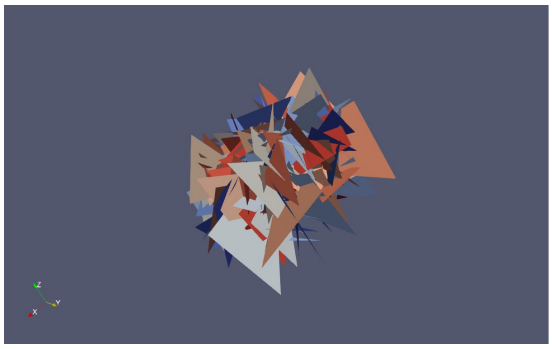

Figura 10: Sotto-poligoni relativi al DFN.

II.3 Polygonal Mesh

Per salvare i sotto-poligoni ottenuti si è utilizzata la seguente struttura dati:

```
struct PolygonalMesh
{
    // Cell0D
    unsigned int NumberOfCell0Ds = 0;
    vector<unsigned int> IdCell0Ds = {};
    vector<Vector3d> CoordinatesCell0Ds;

    // Cell1D
    unsigned int NumberOfCell1Ds = 0;
    vector<unsigned int> IdCell1Ds = {};
    vector<vector<unsigned int>> VerticesCell1Ds = {};

    // Cell2D
```

```
    unsigned int NumberOfCell2Ds = 0;
    vector<unsigned int> NumberOfVertices = {};
    vector<unsigned int> NumberOfEdges = {};
    vector<vector<unsigned int>> VerticesCell2Ds = {};
    vector<vector<unsigned int>> EdgesCell2Ds = {};
};
```

La sua definizione e il relativo riempimento avviene all'interno della funzione *cutPolygons*. Per evitare l'utilizzo di dizionari, si sono utilizzati dei vettori per gestire le coordinate delle celle $0D$ e gli identificativi delle celle 0D e 1D, nel seguente modo:

```
vector<Vector3d> map0D; // Coordinate celle 0D.
vector<pair<unsigned int, unsigned int>> map1D;
// Id celle 0D che delimitano i lati.
vector<unsigned int> id0; // Id vertici.
vector<unsigned int> id1; // Id lati.
unsigned int counterID0 = 0;
unsigned int counterID1 = 0;
```

Per ogni frattura si parte da un oggetto di tipo *PolygonaMesh* che viene riempito con le informazioni dei sotto-poligoni derivanti dal taglio, se la frattura possiede tracce, altrimenti, la mesh viene riempita con i dati iniziali. Nel primo caso, per ogni sotto-poligono e per ogni lato si verifica innanzitutto se le coordinate dei lati erano già presenti all'interno del vettore *map0D*, aggiungendole se necessario insieme al corrispondente identificativo, in questo modo non si formano duplicati. Fatto ciò si verifica se l'identificativo relativo a questo lato era già presente all'interno del vettore *map1D*, aggiungendolo se necessario, dopodiché si riempiono due vettori temporanei che tengono traccia dei lati e dei vertici visitati in modo tale da definire le celle 2D. Terminato il ciclo relativo al sotto-poligono considerato e quello relativo a tutti i sotto-poligoni della frattura considerata, si aggiorna la mesh. Tutto questo viene ripetuto per ogni frattura.

II.4 Documentazione UML

Il codice sorgente è correlato dalla relativa documentazione UML, che fa fede a ciò che è stato spiegato sopra. Per una maggiore comprensione, la documentazione è divisa nelle due parti del progetto ed è possibile visualizzarla qui.