Progetto PCS - 2024

Bisson Gabriele, Ferrero Enrico, Vigna Simone

8 luglio 2024

Abstract

E' stato richiesto di elaborare un algoritmo che deve analizzare un Discrete Fracture Network (DFN), un sistema di poligoni planari (chiamate fratture) che si intersecano e formano delle tracce. In particolare si vuole identificare le tracce, distinguere tra tracce passanti e non-passanti per ogni frattura, ordinare queste tracce in base alla lunghezza decrescente e infine effettuare il taglio della frattura seguendo un ordine specifico di attraversamento delle tracce.

Parte 1 - DETERMINARE LE TRACCE DI UNA DFN

INTRODUZIONE

In questa sezione del progetto vengono letti da un file input i dati relativi ad una DFN da cui si determinano le tracce della DFN e, per ciascuna frattura, si differenziano le tracce passanti e non passanti. Infine per ogni frattura questi due sottoinsiemi sono stati ordinati in modo decrescenti per lunghezza della traccia.

STRUTTURA DATI

Uno dei punti più importanti del progetto è stata la definizione della struttura dati con le relative scelte dei 'contenitori' da utilizzare in modo da sviluppare un codice il più efficiente possibile. Abbiamo deciso di scrivere la struttura dati della DFN in modo molto compatto, nel senso senza definire una struttura specifica per le tracce e per le fratture, ma di implementarne un'unica. Questo è stato voluto perchè più efficiente in termini di memoria, di accesso di dati e riduce la complessità del codice. Ciononostante il codice è più difficile da leggere, specialmente al crescere del numero di informazioni relative alla DFN. Poichè nel nostro caso la struttura contiene un numero limitato di campi e non ci sono relazioni troppo complesse tra di loro abbiamo definito una struttura unica.
Per tutti i dati in cui abbiamo dovuto accedere e modificare più volte (quali vertici delle fratture, estremi tracce, ecc) abbiamo deciso di utilizzare array quando la dimensione era fissa e vettori quando la dimensione era incognita. In generale abbiamo scelto `std::vector` poichè offre una buona combinazione di prestazioni, facilità d'uso e adatto per memorizzare dati da file.
Per quanto riguarda la tipologia di dati ammissibili per ogni dati abbiamo utilizzato quelle più logiche:

- `unsigned int` per numeri interi senza segno. In particolare per il numero delle fratture, delle tracce e dei vertici e per i codici identificativi (ID);
- `bool` per i tips in quanto dobbiamo memorizzare (e poi stampare) false (0) se passante o true (1) se non passante;
- `double` per tutti quei dati derivanti da manipolazione (come la lunghezza della traccia) oppure per le coordinate degli estremi e dei vertici.

```
namespace LibraryDFN{
struct DFN {

    //fratture
    unsigned int numFratture; //numero delle fratture
    std::vector<unsigned int> idFratture; //vettore degli id dei vettori
    std::vector<unsigned int> numVertici; //vettore di numeri di vertici di ogni frattura
    std::vector<std::vector<Eigen::Vector3d>> vertici;
    std::vector<Eigen::Vector3d> versori; //versori normali alle fratture
    std::vector<std::vector<unsigned int>> tracceNonPassanti;
    std::vector<std::vector<unsigned int>> traccePassanti;

    //tracce
    unsigned int numTracce; // numero delle tracce
    std::vector<unsigned int> idTracce; //vettore degli id delle tracce
    std::vector<std::array<unsigned int,2>> tracce; // idFratture1, idFratture2
```

```
17      std::vector<std::array<double, 6>> estremiTracce; // x1, y1, z1, x2, y2, z2
18      std::vector<bool> tips; // vero se non passante, falso se passante
19      std::vector<double> lunghezze; // lunghezza della traccia
20
21  };
22  }
```

Code 1. Struttura dati della DFN.

INPUT SU FILE

All'interno dei *SourceFile* del progetto, in particolare in un file dedicato chiamato *manipFile.cpp* abbiamo definito una funzione specifica per la lettura di file in input:*readDFNFromFile*.
Tale funzione legge e memorizza l'identificativo delle fratture, il numero e le coordinate di tutti i suoi vertici in un vettore che viene aggiunto al vettore *vertici* della struttura DFN.
Nella prima parte del codice dopo aver aperto il file in modalità di lettura. Se l'apertura fallisce, per esempio quando il file non esiste oppure non si hanno i permessi per aprirlo, viene stampato un messaggio di errore con il metodo `std::cerr`. In questa funzione abbiamo utilizzato il metodo `reserve` per risparmiare memoria infatti tale comando ci permette di preallocare memoria nei vettori (*idFratture*, *numVertici*, *vertici*) evitando riallocazioni multiple durante l'inserimento dei dati, migliorando così l'efficienza. Abbiamo inoltre cercato di evitare ridondanza nella lettura utilizzando un singolo stream *iss2* per leggere i vertici.
Alla fine del codice abbiamo chiuso con il metodo `close` il file in modo da assicurare assicurare la corretta gestione delle risorse, prevenire la corruzione dei dati, garantire la sincronizzazione dei dati e mantenere il tuo programma più affidabile.

ALGORITMO TRACCE

Abbiamo implementato l'algoritmo che gestisce le fratture e identifica le relative tracce della DFN nel file chiamato *Utils.cpp*, le dichiarazioni delle funzioni che costituiscono l'algoritmo sono state prima inserite nel file *Uilts.hpp*.

PRECISIONE FINITA

Questo algoritmo (come il successivo ALGORITMO MESH) lavora in precisione finita di calcolo, quindi ogni volta che è presente un'uguaglianza tra 2 oggetti di tipo double o vettori, questa è verificata se la norma della loro differenza, relativizzata all'intervallo [0,1], è minore della tolleranza relativa che viene fornita in input dall'utente (comunque non minore di 10 volte l'epsilon di macchina). Tutte le uguaglianze (e disuguaglianze) presenti nel codice tengono conto della precisione finita e quindi della tolleranza relativa.

PASSI DELL'ALGORITMO

Le funzioni che costituiscono l'algoritmo per il calcolo delle tracce sono:

scarta_fratture: Questa funzione scarta le fratture che sicuramente non si intersecano restituendo un vettore di coppie di indici di fratture candidate ad intersecarsi. I passi fondamentali di questa parte sono:

1. Calcolo per ogni frattura del punto medio tra i vertici di essa, sommando le coordinate dei vertici e dividendo per il numero di vertici:
$$V_m = \frac{1}{N}\sum_{i=1}^{N} V_i \tag{1}$$
2. Calcolo del raggio al quadrato massimo tra punto medio e i vertici:
$$r^2 = \max\left\{\|\mathbf{v_i} - \mathbf{v_m}\|^2 \,:\, i = 1, \dots, N\right\} \tag{2}$$
3. Primo test su tutte le coppie non ordinate di fratture in cui si verifica se la distanza al quadrato tra i punti medi è minore della somma dei 2 raggi. Se tale condizione è verificata, si effettua il secondo test.
Riguardo la foto a destra il primo test è verificato se $d(A,B)^2 > r^2 + R^2$ (si prende il quadrato, data la monotonia dell'elevamento al quadrato, per risparmiare conti e velocizzare il programma).

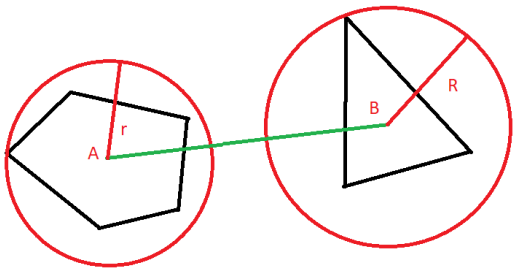


Figure 1. Primo test

4. Secondo test sulle coppie di fratture che hanno superato il primo test, sfruttando il Teorema dell'asse di separazione: si calcolano le proiezioni di F2 e F1 sull'asse normale a F1 e passante per il punto medio precedentemente calcolato, si controlla che si intersechino, se ciò avviene si ripete il controllo con le proiezioni di F2 e F1 sull'asse normale a F2 e passante per il punto medio di F2; il test è considerato superato se in tutti e due i casi l'intersezione tra le due proiezioni è non vuota. Nel caso della figura a destra il test non è superato dato che $d(b, x) > 0$
   Se viene superato anche questo test, la coppia di ID delle fratture viene salvata in un vettore che viene dato come output della funzione.

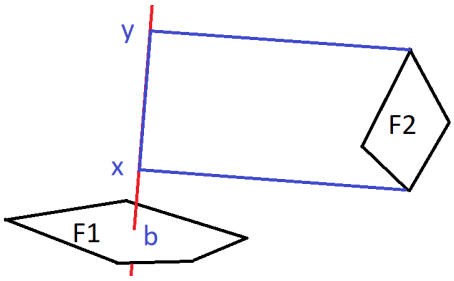


Figure 2. Secondo test

triangola_frattura: Questa funzione ausiliaria divide una frattura in triangoli mediante i seguenti passi:

1. Creazione di un vettore per memorizzare i triangoli;
2. Per ogni vertice dalla seconda posizione alla penultima si crea un triangolo utilizzando il primo vertice e i due adiacenti;
3. Il triangolo creato viene aggiunto al vettore che viene poi restituito.

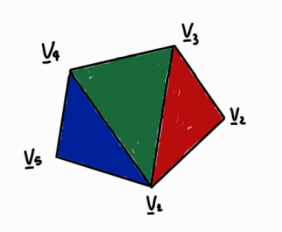


Figure 3. Esempio poligono triangolato

versore_normale: La funzione determina il versore normale al piano che contiene il poligono della frattura, utilizzato successivamente per orientare correttamente le fratture durante il confronto e l'intersezione.

memorizza_tracce: Questa parte dell'algoritmo memorizza le tracce delle intersezioni tra le fratture nella struttura DFN utilizzando le funzioni specificate nei punti precedenti e le ordina per passanti, non passanti e per frattura.

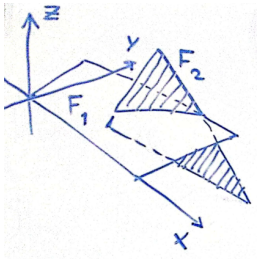


Figure 4. Roto-traslazione "immaginaria" della coppia di fratture

Concentrandosi su una coppia di fratture che si intersecano (F1,F2), gli estremi del segmento di intersezione derivano dall'intersezione di lati di F1 con F2 e/o lati di F2 con F1, quindi l'idea è stata quella di cercare tutte le intersezioni dei lati di F2 con F1 e/o cercare tutte le intersezioni dei lati di F1 con F2.

Per ogni coppia di fratture (F1,F2), ottenuta dalla funzione *scarta fratture*, si effettua una rotazione rigida delle 2 fratture in modo da allineare F1 al piano xy. Ciò serve per controllare in maniera rapida se un lato della frattura F2 interseca il piano che contiene la frattura F1. Infatti per ogni lato di F2 di estremi (x_1, y_1, z_1) e (x_2, y_2, z_2) si controlla se $z_1 z_2 \leq 0$ e,in tal caso il lato, interseca il piano contenente F1.
In vista di ciò non sono roto-traslati tutti i vertici di tutte le 2 fratture, ma vengono "roto-traslate" solo le z dei vertici della frattura F2.

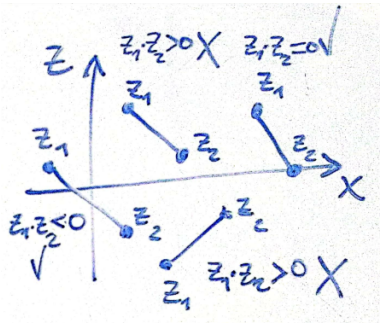


Figure 5. Esempi di situazioni di intersezione di un lato con il piano che contiene la frattura F1

Tra i lati di F2 che intersecano il piano contenente F1 si trova il punto di intersezione con tale piano e si verifica se è interno a F1, controllando l'intersezione di questi lati con ogni triangolo ottenuto dalla triangolazione di F1 attraverso *triangola frattura*.
Per ogni triangolo, in riferimento alla figura a destra, si ha che:

$$\underline{X} = \underline{A} + \alpha(\underline{C} - \underline{A}) + \beta(\underline{B} - \underline{A}) = \gamma(\underline{e}_2 - \underline{e}_1) + \underline{e}_1 \tag{3}$$

Ciò determina un sistema lineare di incognite (α,β,γ) che viene risolto con la fattorizzazione $PA = LU$. Si nota che α e β sono le coordinate baricentriche, da ciò si ottiene che:

- Se $\alpha \geq 0,\ \beta \geq 0 \wedge \gamma \in [0,1) \wedge \alpha + \beta \leq 1 \implies$ il lato interseca il triangolo.
- Se $\alpha + \beta = 1$ o $\beta = 0$ o $\alpha = 0 \implies$ il lato interseca il bordo del triangolo, da ciò con un ulteriore controllo si può verificare se il punto di intersezione sta sul bordo del poligono.

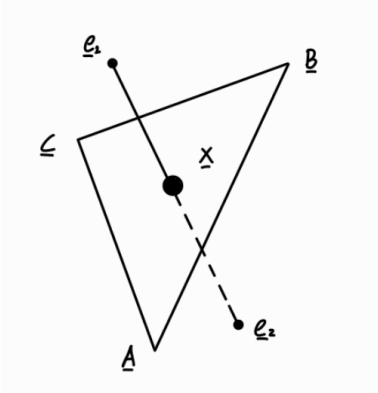


Figure 6. Intersezione tra triangolo e lato

In caso di intersezione effettiva tra un lato e la frattura F1 si ha un estremo della traccia, si aumenta così di +1 una variabile intera temporanea relativa alla frattura F2, detta tip intermedio di F2 (se è sul bordo si aumenta di +1 anche il tip intermendio di F1).
Si procede in questo modo fino a trovare tutti e 2 gli estremi della traccia, eventualmente considerando le intersezioni dei lati di F1 con la frattura F2 nello stesso modo con cui si sono cercate le intersezioni dei lati di F2 con F1.
Trovati entrambi gli estremi della traccia, vengono aggiornate tutte le info della struttura DFN legate alle tracce.
In particolare nel memorizzare i tip, si passa dal tip intermedio al tip effettivo con la trasformazione: $tip_{intermedio} \in \{0,1\} \implies tip_{effettivo} = 1, tip_{intermedio} = 2 \implies tip_{effettivo} = 0$.

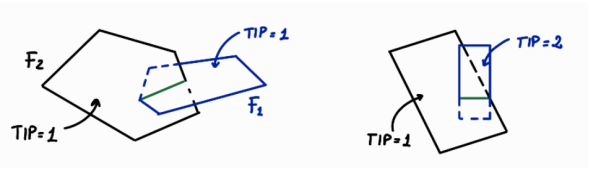


Figure 7. Due esempi di intersezioni tra fratture in cui si possono alcuni casi di tip intermedi delle fratture

◆ OTTIMIZZAZIONI E GESTIONE DELLA MEMORIA

In generale abbiamo utilizzato il metodo `reserve` per vettori in modo da minimizzare le riallocazioni e migliorare le prestazioni del codice. Inoltre si è cercata la massima riduzione del numero di cicli annidati e la migliore ottimizzazione delle operazioni condizionali per migliorare la velocità di esecuzione e la chiarezza del codice.

■ OUTPUT SU FILE

All'interno del progetto abbiamo scritto due funzioni di stampa che scrivono l'output su due relativi file. Per entrambe le funzioni abbiamo utilizzato il metodo `ofstream` che fornisce una interfaccia semplice per scrivere su file di testo. Inoltre tale metodo gestisce automaticamente le conversioni di tipo per scrivere variabili di diversi tipi in formato testuale nel file ed è progettato per essere efficiente nel tempo di esecuzione, consentendo la scrittura di grandi quantità di dati su file in modo rapido. Dopo l'apertura di ogni file in modalità di scrittura il codice verifica che siano aperti correttamente, in caso contrario viene stampato un messaggio di errore con il metodo `std::cout`. Quando la scrittura dei file viene terminata ogni file di output viene esplicitamente chiuso usando il comando `close`. La chiusura del file è importante per evitare la perdita di dati e per garantire l'integrabilità dei dati.

◆ STAMPA TRACCE: printTraces

printTraces è una funzione dedicata alla stampa delle tracce, in particolare viene visualizzato in un file output: numero di tracce, identificativo delle tracce e delle fratture coinvolte e le coordinate dei punti che la identificano.

◆ STAMPA TRACCE PASSANTI E NON PASSANTI: sortTracesAndPrintByFracture

sortTracesAndPrintByFracture per ogni frattura ordina separatamente le tracce divise in passanti e non passanti in ordine decrescente in base alla lunghezza.
Abbiamo fatto uso del comando `emplace_back` in quanto inserisce un nuovo elemento direttamente nel vettore senza la necessità di creare temporaneamente un oggetto prima di inserirlo, risparmiando tempo e memoria. Abbiamo utilizzato un algoritmo di ordinamento delle tracce in base alla lunghezza in ordine decrescente. Per quanto riguarda questo punto, abbiamo deciso di non utilizzare il comando interno a *C++* per l'ordinamento ovvero `std::sort`, allora abbiamo implementato un algoritmo di ordinamento decrescente del metodo *MergeSort*. Abbiamo quindi deciso di utilizzare `MergeSort` per rendere il codice il più stabile possibile. Infatti:

- `std::sort` ha un costo computazione che varia tra $O(n \log n)$ e $O(n^2)$. Il metodo `std::sort` utilizza un algoritmo *QuickSort* in cui ci possono essere casi particolari in cui il costo può esplodere e quindi il codice può risultare lento;

- *MergeSort* ha un costo computazionale garantito in tutti i casi di $O(n \log n)$, è stabile e mantiene la stessa efficienza anche nei casi pessimi.

DOCUMENTAZIONE UML

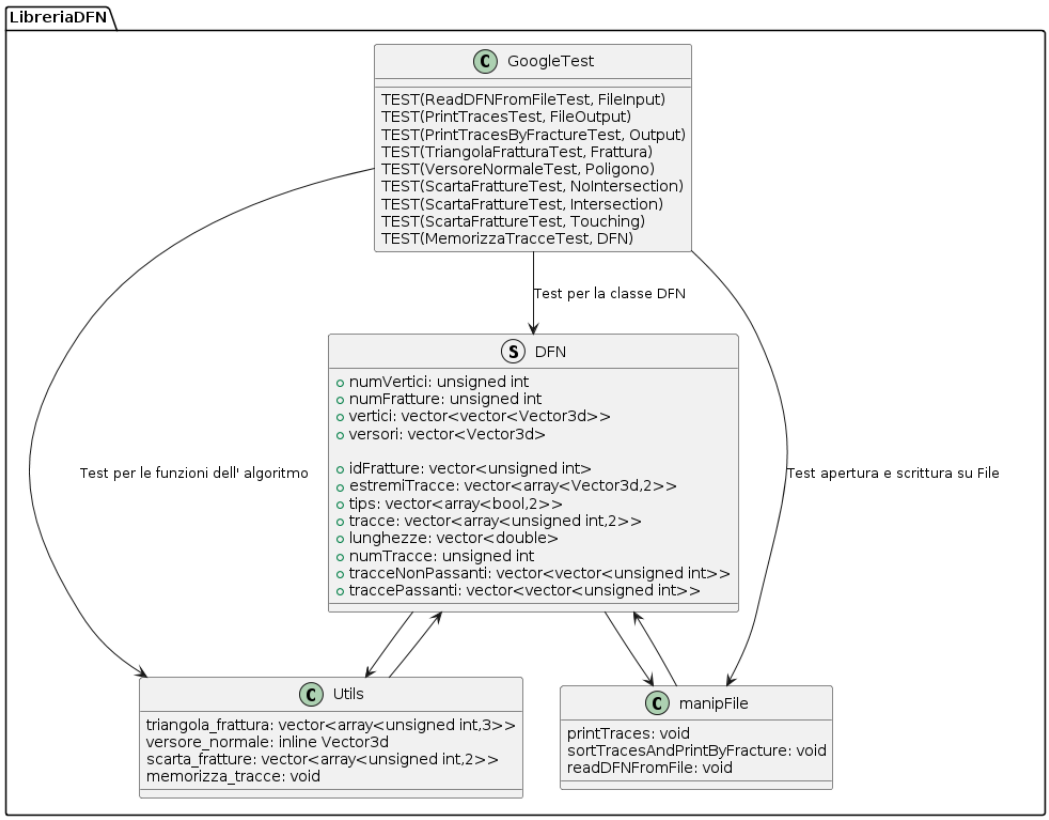
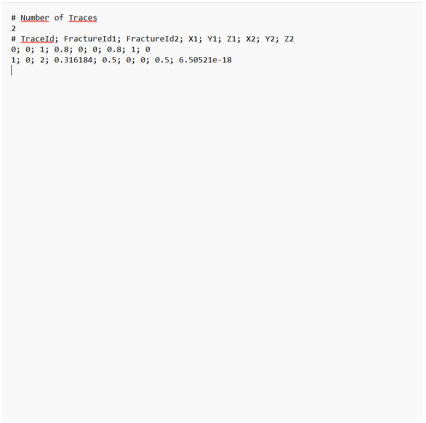


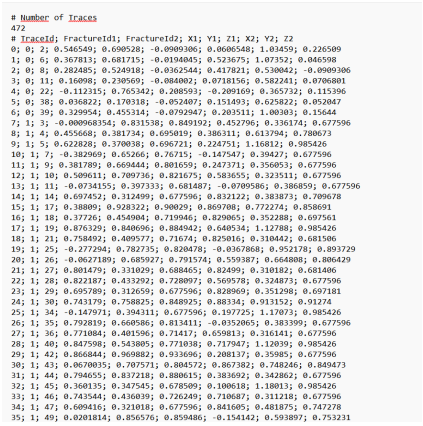
Figure 8. Diagramma delle classi della parte 1 del progetto in cui specifichiamo le relazioni tra le varie classi.

QUALCHE OUTPUT OTTENUTO



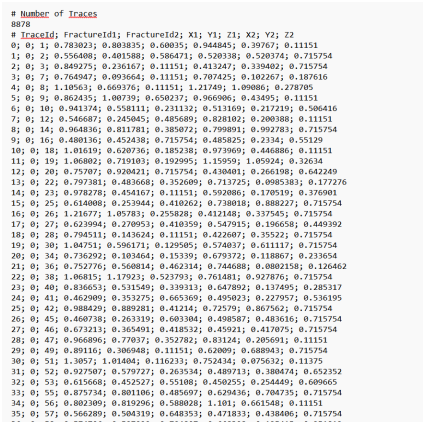
```
# Number of Traces
2
# TraceId; FractureId1; FractureId2; X1; Y1; Z1; X2; Y2; Z2
0; 0; 1; 0.8; 0; 0; 0.8; 1; 0
1; 0; 2; 0.316184; 0.5; 0; 0; 0.5; 6.50521e-18
```

Figure 9. FR3



```
# Number of Traces
472
# TraceId; FractureId1; FractureId2; X1; Y1; Z1; X2; Y2; Z2
0; 0; 2; 0.546549; 0.690528; -0.0909306; 0.0606548; 1.03459; 0.226509
1; 0; 6; 0.367813; 0.681715; -0.0194045; 0.523675; 1.07352; 0.046598
2; 0; 8; 0.282485; 0.524918; -0.0362544; 0.417821; 0.530042; -0.0909306
3; 0; 11; 0.160918; 0.230569; -0.0804002; 0.0718156; 0.582241; 0.0706801
4; 0; 22; -0.112315; 0.765342; 0.208593; -0.209169; 0.365732; 0.115396
5; 0; 38; 0.036822; 0.170318; -0.052407; 0.151493; 0.625822; 0.052047
6; 0; 39; 0.329954; 0.455314; -0.0792947; 0.203511; 1.00303; 0.15644
7; 1; 3; -0.000960354; 0.831538; 0.849192; 0.452796; 0.336174; 0.677596
8; 1; 4; 0.455668; 0.381734; 0.695019; 0.386311; 0.613794; 0.780673
9; 1; 5; 0.622828; 0.370038; 0.696721; 0.224751; 1.16812; 0.985426
10; 1; 7; -0.382969; 0.65266; 0.76715; -0.147547; 0.39427; 0.677596
11; 1; 9; 0.381789; 0.669444; 0.801659; 0.247371; 0.356053; 0.677596
12; 1; 10; 0.509611; 0.709736; 0.821675; 0.583655; 0.323511; 0.677596
13; 1; 11; -0.0734155; 0.397333; 0.681487; -0.0709586; 0.386859; 0.677596
14; 1; 14; 0.697452; 0.312499; 0.677596; 0.832122; 0.383873; 0.709678
15; 1; 17; 0.388089; 0.928322; 0.90029; 0.869708; 0.772274; 0.858691
16; 1; 18; 0.37726; 0.454904; 0.719946; 0.829065; 0.352288; 0.697561
17; 1; 19; 0.876329; 0.840696; 0.884942; 0.640534; 1.12788; 0.985426
18; 1; 21; 0.758492; 0.409577; 0.71674; 0.825016; 0.310442; 0.681506
19; 1; 25; -0.272294; 0.782735; 0.820478; -0.0367868; 0.952178; 0.893729
20; 1; 26; -0.0627189; 0.685927; 0.791574; 0.559387; 0.664808; 0.806429
21; 1; 27; 0.801479; 0.331029; 0.688465; 0.82499; 0.310182; 0.681406
22; 1; 28; 0.822187; 0.433292; 0.728097; 0.569578; 0.324873; 0.677596
23; 1; 29; 0.695789; 0.312659; 0.677596; 0.828969; 0.351298; 0.697181
24; 1; 30; 0.743179; 0.758825; 0.848925; 0.88334; 0.913152; 0.91274
25; 1; 34; -0.147971; 0.394311; 0.677596; 0.197725; 1.17073; 0.985426
26; 1; 35; 0.792819; 0.660586; 0.813411; -0.0352065; 0.383399; 0.677596
27; 1; 36; 0.771084; 0.401596; 0.71417; 0.659813; 0.316141; 0.677596
28; 1; 40; 0.847598; 0.543805; 0.771038; 0.717947; 1.12039; 0.985426
29; 1; 42; 0.866844; 0.969882; 0.933696; 0.208137; 0.35985; 0.677596
30; 1; 43; 0.0670035; 0.707571; 0.804572; 0.867382; 0.748246; 0.849473
31; 1; 44; 0.794655; 0.837218; 0.880615; 0.383692; 0.342862; 0.677596
32; 1; 45; 0.360135; 0.347545; 0.678509; 0.100618; 1.18013; 0.985426
33; 1; 46; 0.743544; 0.436039; 0.726249; 0.710687; 0.311218; 0.677596
34; 1; 47; 0.609416; 0.321018; 0.677596; 0.841605; 0.481875; 0.747278
35; 1; 49; 0.0201814; 0.856576; 0.859486; -0.154142; 0.593897; 0.753231
```

Figure 10. FR50



```
# Number of Traces
8878
# TraceId; FractureId1; FractureId2; X1; Y1; Z1; X2; Y2; Z2
0; 0; 1; 0.783023; 0.803835; 0.60035; 0.944845; 0.39767; 0.11151
1; 0; 2; 0.556408; 0.401588; 0.586471; 0.520338; 0.520374; 0.715754
2; 0; 3; 0.849275; 0.236167; 0.11151; 0.413247; 0.339402; 0.715754
3; 0; 7; 0.764947; 0.093664; 0.11151; 0.707425; 0.102267; 0.187616
4; 0; 8; 1.10563; 0.669376; 0.11151; 1.21749; 1.09086; 0.278705
5; 0; 9; 0.862435; 1.00739; 0.650237; 0.966906; 0.43495; 0.11151
6; 0; 10; 0.941374; 0.558111; 0.231132; 0.513169; 0.217219; 0.586416
7; 0; 12; 0.546687; 0.245045; 0.485689; 0.828102; 0.200388; 0.11151
8; 0; 14; 0.964836; 0.811781; 0.385072; 0.799891; 0.992783; 0.715754
9; 0; 16; 0.480136; 0.452438; 0.715754; 0.485825; 0.2334; 0.55129
10; 0; 18; 1.01619; 0.620736; 0.185238; 0.973969; 0.446886; 0.11151
11; 0; 19; 1.06802; 0.719103; 0.192995; 1.15959; 1.05924; 0.32634
12; 0; 20; 0.75707; 0.920421; 0.715754; 0.430401; 0.266198; 0.642249
13; 0; 22; 0.797381; 0.483668; 0.352609; 0.713725; 0.0985383; 0.177276
14; 0; 23; 0.978278; 0.454167; 0.11151; 0.592086; 0.170519; 0.376001
15; 0; 25; 0.614008; 0.253944; 0.410262; 0.738018; 0.888227; 0.715754
16; 0; 26; 1.21677; 1.05783; 0.255828; 0.412148; 0.337545; 0.715754
17; 0; 27; 0.623994; 0.270953; 0.410359; 0.547915; 0.196658; 0.449392
18; 0; 28; 0.794511; 0.143624; 0.11151; 0.422607; 0.35522; 0.715754
19; 0; 30; 1.04751; 0.596171; 0.129505; 0.574037; 0.611117; 0.715754
20; 0; 34; 0.736292; 0.103464; 0.15339; 0.679372; 0.118867; 0.233654
21; 0; 36; 0.752776; 0.560814; 0.462314; 0.744688; 0.0802158; 0.126462
22; 0; 38; 1.06815; 1.17923; 0.523793; 0.761481; 0.927876; 0.715754
23; 0; 40; 0.836653; 0.531549; 0.339313; 0.647892; 0.137495; 0.285317
24; 0; 41; 0.462909; 0.353275; 0.665369; 0.495023; 0.227957; 0.536195
25; 0; 42; 0.988429; 0.889281; 0.41214; 0.72579; 0.867562; 0.715754
26; 0; 45; 0.460738; 0.263319; 0.603304; 0.498587; 0.483616; 0.715754
27; 0; 46; 0.673213; 0.365491; 0.418532; 0.45921; 0.417075; 0.715754
28; 0; 47; 0.966896; 0.77037; 0.352782; 0.83124; 0.205691; 0.11151
29; 0; 49; 0.89116; 0.306948; 0.11151; 0.62009; 0.688943; 0.715754
30; 0; 51; 1.3057; 1.01404; 0.116233; 0.752434; 0.075632; 0.11375
31; 0; 52; 0.927507; 0.579727; 0.263534; 0.489713; 0.380474; 0.652352
32; 0; 53; 0.615668; 0.452527; 0.55108; 0.450255; 0.254449; 0.609665
33; 0; 55; 0.875734; 0.801106; 0.485697; 0.629436; 0.704735; 0.715754
34; 0; 56; 0.802309; 0.819296; 0.580028; 1.101; 0.661548; 0.11151
35; 0; 57; 0.566289; 0.504319; 0.648353; 0.471833; 0.438406; 0.715754
```

Figure 11. FR200

Figure 12. Immagini ricavate dai file output generati da printTraces.

```
# FractureId; NumTraces
0; 2
# TraceId; Tips; Length
0; 1; 1
1; 0; 0.316184
# FractureId; NumTraces
1; 1
# TraceId; Tips; Length
0; 1; 1
# FractureId; NumTraces
2; 1
# TraceId; Tips; Length
1; 0; 0.316184
```

Figure 13. FR3

```
# FractureId; NumTraces
0; 7
# TraceId; Tips; Length
0; 1; 0.674714
6; 0; 0.609552
5; 0; 0.48119
1; 0; 0.426801
4; 0; 0.421609
3; 0; 0.394399
2; 0; 0.146053
# FractureId; NumTraces
1; 29
# TraceId; Tips; Length
20; 1; 0.622641
35; 1; 0.332684
19; 1; 0.303184
8; 1; 0.256902
9; 0; 0.937414
29; 0; 0.933607
```

Figure 14. FR50

```
# FractureId; NumTraces
0; 112
# TraceId; Tips; Length
66; 1; 1.11292
49; 1; 0.80635
29; 1; 0.764532
2; 1; 0.752255
18; 1; 0.740402
89; 1; 0.736545
12; 1; 0.734931
14; 1; 0.547752
4; 1; 0.467026
60; 1; 0.444911
22; 1; 0.440539
62; 1; 0.434152
93; 1; 0.382108
43; 1; 0.346969
94; 1; 0.276378
9; 1; 0.273968
```

Figure 15. FR200

Figure 16. Immagini ricavate dai file output generati da printTraces.

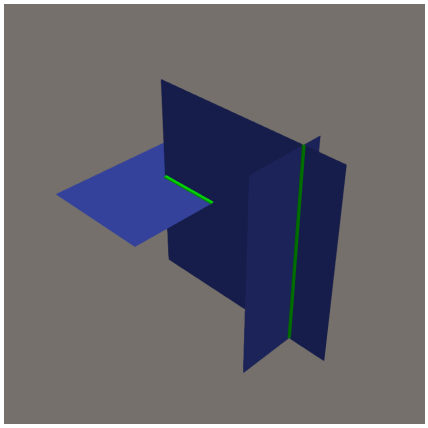


Figure 17. FR3

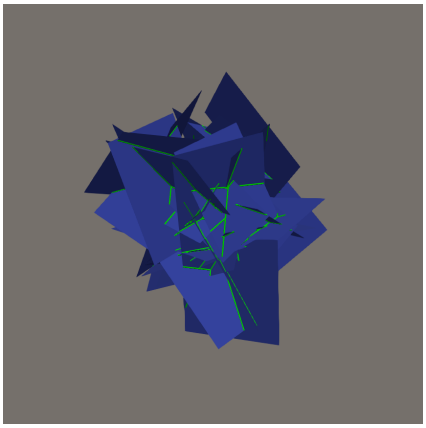


Figure 18. FR50

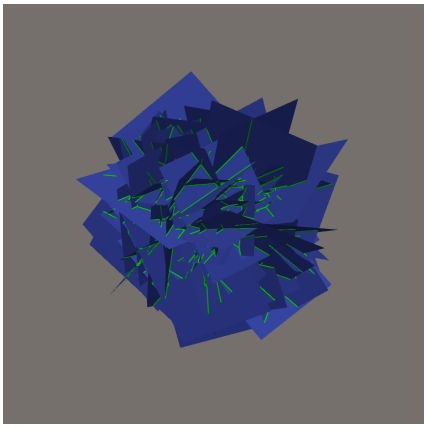


Figure 19. FR200

Figure 20. Immagini ricavate usando *Paraview* di tre dataset forniti: in blu sono raffigurate le fratture, in verde le tracce.

Parte 2 - DETERMINARE I SOTTO-POLIGONI GENERATI PER OGNI FRATTURA

INTRODUZIONE

In questa seconda parte del progetto per ogni frattura, si sono determinate le mesh poligonali generati dal taglio della frattura con le sue tracce generando infine una mesh. Per il taglio della frattura si procede prima dalle tracce passanti e poi da quelle non passanti, entrambi in ordine decrescente di lunghezza.

STRUTTURA DATI

Anche in questo caso la definizione di una struttura dati efficiente è stata di primaria importanza per rendere il codice il più veloce possibile. Per la seconda parte del progetto sono state necessarie due strutture dati: quella della DFN e quella per la costruzione dei sotto-poligoni.

AGGIORNAMENTO DELLA STRUTTURA DFN

Per quanto riguarda la struttura della DFN già spiegata nel dettaglio nella prima parte, è stato necessario aggiungere un contenitore relativo alle mesh di ogni frattura.

```
std::vector<PolygonalMesh> meshPoligonali; //vettore con le mesh poligonali relative ad ogni frattura
```

Code 2. Stringa di codice aggiunta alla struttura DFN precedente.

◆ STRUTTURA MESH

La struttura dati definita nel namespace PolygonalLibrary descrive una mesh poligonale, che può rappresentare una varietà di geometrie tramite celle 0D (punti), celle 1D (segmenti) e celle 2D (poligoni). La struttura PolygonalMesh è organizzata in tre sezioni principali, ciascuna delle quali rappresenta un diverso tipo di cella: 0D, 1D e 2D. Ogni sezione contiene attributi specifici per gestire gli elementi e le loro proprietà. Questa struttura è stata progettata per fornire una rappresentazione flessibile e dettagliata della mesh in modo tale che le informazione all'interno dei vettori permettano di accedere rapidamente agli elementi geometrici tramite i loro ID. In particolare, nella definizione della struttura abbiamo utilizzato `unsigned int` per rappresentare i numeri degli elementi che non possono essere negativi. L'uso di `std::vector<unsigned int>` è stato utile per avere un contenitore che più aumentare o diminuire la sua dimensione automaticamente consentendo un accesso rapido e diretto. Abbiamo invece sceleto di utilizzare `std::vector<Eigen::Vector3d>` per rappresentare vettori a 3 dimensioni x,y,z creando un array dinamico. Abbiamo scelto `std::vector<std::array<unsigned int, 2»` perchè permette di avere un array di dimensioni fisse che facilitasse l'accesso. Infine abbiamo optato a `std::vector<std::vector<unsigned int»` per creare un array dinamico di altri vettori.

■ ALGORITMO MESH

Sempre in *Utils.cpp* abbiamo implementato l'algoritmo che per ogni frattura memorizza i vertici, i lati e i poligoni generati dal taglio della frattura con le sue tracce.
L'algoritmo è stato strutturato in 5 funzioni.

interseca_segmenti: Questa funzione, dati 2 segmenti S_1 e S_2 di estremi A_1,A_2 e B_1,B_2, trova il punto di intersezione tra le rette che contengono i segmenti quando esse sono perfettamente complanari e, quando non lo sono per errori di arrotondamento dovui al compilatore, trova il candidato migliore ad approssimare l'ipotetico punto di intersezione:

$$(\underline{P},\underline{Q}) = \arg\min\{\|\underline{P}-\underline{Q}\|_2 : \underline{P}\in S_1 \wedge \underline{Q}\in S_2\} \tag{4}$$

$$\underline{X} = \frac{\underline{P}+\underline{Q}}{2} \tag{5}$$

dopo vari calcoli si ottiene:

$$\underline{P} = \underline{A_1} + \beta(\underline{A_2}-\underline{A_1}) \tag{6}$$

$$\begin{cases} \beta = \frac{\langle \underline{V},\langle \underline{W},\hat{n}\rangle\hat{n}-\underline{W}\rangle}{\|\underline{V}\|^2-\langle \underline{V},\hat{n}\rangle^2} \\ \underline{V} = \underline{B_2}-\underline{B_1} \\ \underline{W} = \underline{B_1}-\underline{A_1} \\ \underline{M} = \frac{\underline{A_2}-\underline{A_1}}{\|\underline{A_2}-\underline{A_1}\|_2} \end{cases} \tag{7}$$

Una relazione analoga si ottiene anche per $\underline{Q}$.
Evidentemente quando le rette sono perfettamente complanari, $\underline{P},\underline{Q}$ e $\underline{X}$ coincidono con il punto di intersezione tra le due rette.
Data la ridotta dimensione di questa funzione e il frequente uso di essa nel programma, abbiamo scelto di implementare questa funzione come funzione *inline*, in modo da espandere il corpo della funzione dove essa viene chiamata per ridurre il numero di chiamate a funzione.

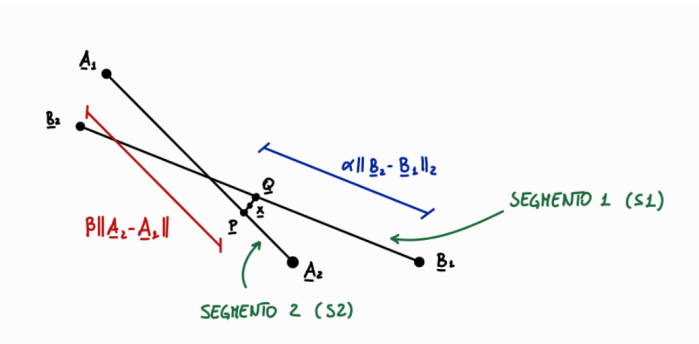


Figure 21. Illustrazione grafica del funzionamento della funzione

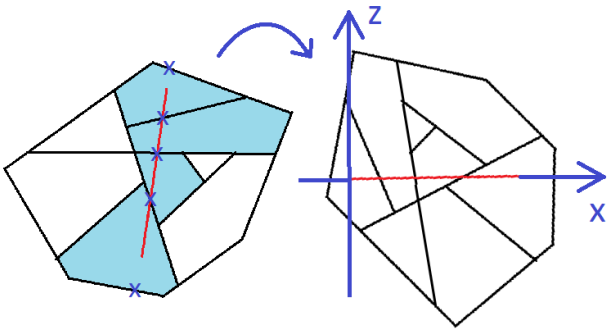


Figure 22. Roto-traslazione della mesh per la semplificazione dei calcoli, nella mesh a sinistra le celle 2D in azzurro sono quelle che intersecano significativamente il nuovo segmento, i lati con una "x" sopra solo i lati che vengono dati in output dalla funzione e che servono poi per il taglio della mesh

contatto_poligoni_segmento: Data una mesh poligonale e un segmento giacente su di essa, questa funzione trova e restituisce gli ID delle celle 2D che hanno una intersezione significativa con il segmento, per tali celle 2D vengono anche identificati relativi lati che intersecano l'asse che contiene il segmento, i quali vengono restituiti in output, in quanto servono poi per il taglio della mesh.

I passaggi fondamentali sono:

1. La frattura viene rototraslata in modo da essere contenuta nel piano XZ, il segmento è contenuto nell'asse X (in particolare in $\mathbb{R}^+$ e con un estremo del segmento coincidente con l'origine degli assi cartesiani).
2. Per ogni cella 2D si stabilisce la lunghezza d dell'insieme cella 2D $\cap$ segmento: con d=0 non c'è intersezione, con d>0 invece c'è intersezione. Nota: nel terzo caso in basso a destra della figura 23, anche se l'intersezione è non nulla, comunque essa viene considerata nulla, in quanto non va a modificare la cella 2D.
3. Vengono salvati in vettore gli ID delle celle 2D che intersecano significativamente il segmento e in un altro vettore vengono salvate le coppie di celle 1D che intersecano l'asse che contiene il segmento (una coppia per ogni cella 2D che interseca significativamente il segmento).

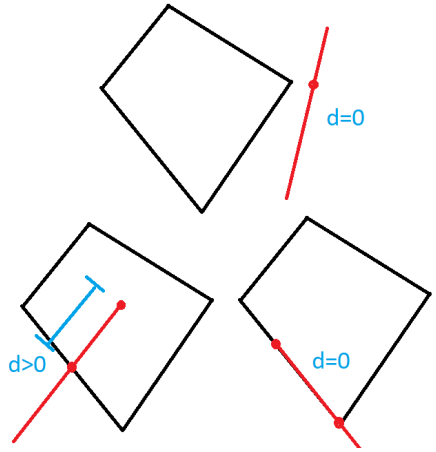


Figure 23. Esempi di intersezioni tra una cella 2D e il segmento: se d>0 allora l'intersezione è significativa, se d=0 invece non c'è intersezione o l'intersezione è non significativa

aggiorna_mesh: Data una mesh e una traccia che va a tagliare la mesh, questa funzione aggiorna i dati relativi ad essa in modo da includere il nuovo taglio, in particolare vengono aggiornate:

1. Celle 0D: per ogni cella 2D, utilizzando *interseca_segmenti*, si trovano le coordinate degli estremi del prolungamento della traccia fino ai bordi della cella 2D, si aggiornano così gli ID delle nuove celle 0D e i marker;
2. Celle 1D: il taglio genera una nuova cella 1D per ogni cella 2D coinvolta nel taglio e va eventualmente a dividere i 2 lati coinvolti anche loro nel taglio, vengono aggiornati così gli ID delle celle 1D, gli estremi e i marker;
3. Celle 2D: per ogni cella 2D che interseca significativamente la traccia, vengono generate 2 nuove celle 2D (utilizzando la funzione *nuovo_poligono*) a seguito del taglio.

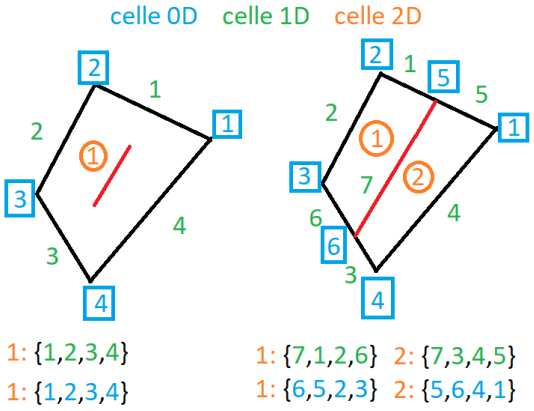


Figure 24. Esempio di aggiornamento dei dati relativi ad una cella 2D: a sinistra la situazione prima del taglio, a destra la situazione dopo il taglio

nuovo_poligono: Data una cella 2D che viene aggiornata dal taglio dovuto ad una traccia, crea uno dei due nuovi poligoni generati, ossia costruisce i vettori di ID dei lati e dei vertici che costituiscono la nuova cella: partendo da un estremo del segmento tagliante, specificato via input, e procedendo in senso antiorario, si visitano tutti i vertici che identificano il nuovo poligono. Questi vengono inseriti in un vettore (e lo stesso viene fatto per i lati) che viene dato in output.

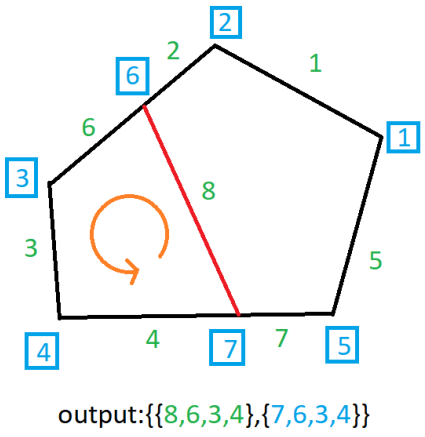


Figure 25. Esempio creazione di del nuovo poligono dopo il taglio generato dalla traccia

definisci_mesh: È la funzione principale che sfrutta tutte le funzioni definite in precedenza per creare una mesh su una frattura a partire dalle tracce giacenti su di essa.

1. Viene creata la mesh corrispondente alla frattura stessa come unica cella 2D.
2. Per ogni traccia passante (prese in ordine descrescente di lunghezza), utilizzando *contatto_poligoni_segmento* e *aggiorna_mesh*, si aggiorna la mesh in base al nuovo taglio.
3. Si ripete lo stesso procedimento con tutte le tracce non passanti prese in ordine decrescente di lunghezza.
4. La mesh ottenuta viene salvata in un vettore di mesh poligonali all'interno della struttura DFN.

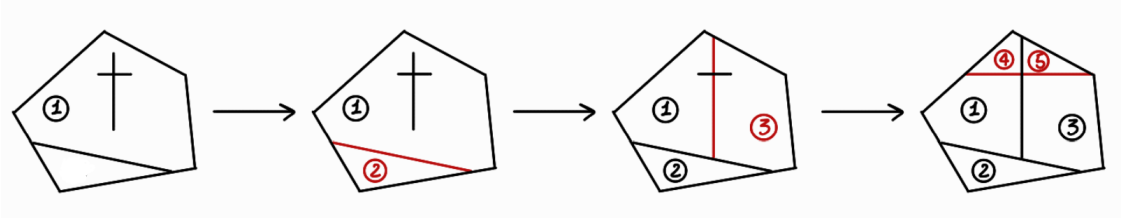


Figure 26. Illustrazione sull'ordine dei tagli e l'ordine di numerazione dei sotto-poligoni

GOOGLE TEST

In questa sezione vengono analizzati una serie di test unitari implementati utilizzando la libreria Google Test per verificare la funzionalità della libreria DFN che abbiamo realizzato nel progetto. I test coprono la lettura di file, la stampa di tracce, la triangolazione di fratture, la normalizzazione di poligoni, l'intersezione di segmenti, la gestione e le modifiche delle fratture. I test coprono una vasta gamma di funzionalità della libreria DFN, assicurando che le operazioni critiche come la lettura e scrittura di dati, il calcolo delle intersezioni e l'aggiornamento delle fratture siano implementate correttamente. Di seguito forniamo una descrizione dettagliata di ogni test.

Test per la funzione ReadDFNFromFile

Questo test verifica che la funzione *readDFNFromFile* legga correttamente i dati di fratture da un file. In particolare abbiamo eseguyito il test con il file noto *DFN/FR3_data.txt*. Abbiamo verificato che vengano letti correttamente i seguenti dati:

- Numero di vertici: 3;
- Numero di ID delle fratture: 3;
- Numero di vertici per fratture: 3;
- Coordinate dei vertici per ogni frattura.

Test per la funzione printTraces

Questo test verifica che la funzione *printTraces* scriva correttamente le tracce in un file. Utilizzando dei dati di test inseriti manualmente nel codice, abbiamo verificato che nel file output *temp_output.txt* generato da *printTraces* sia:

- Buona apertura;
- Contenuto corretto del file.

◆ Test per la funzione sortTracesAndPrintByFracture

Questo test verifica che la funzione *sortTracesAndPrintByFracture* scriva correttamente le tracce per ogni frattura in un file. In modo analogo alla sezione precedente abbiamo verificato la buona apertura del file e che il contenuto sia corretto utilizzando un data set inserito manualmente.

◆ Test per la funzione triangola_frattura

Questo test verifica che la funzione *triangola_frattura* crei correttamente triangoli da una frattura. Abbiamo dato in input una frattura con 5 vertici e abbiamo verificato che venga restituita una lista di triangoli rappresentati da triple di indici dei vertici.

◆ Test per la funzione versore_normale

Questo test verifica che la funzione *versore_normale* calcoli correttamente il versore normale di un poligono. Dando in input un poligono di 4 vertici abbiamo verificato che fosse restituito un vettore `Eigen::Vector3d(0, 0, 1)`.

◆ Test per la funzione scarta_fratture

Questo test verifica che la funzione *scarta_fratture* identifichi correttamente le fratture che si intersecano e che non si intersecano. Per questa parte dell'algoritmo abbiamo testato:

- *NoIntersection*: Verifica che non ci siano intersezioni tra le fratture date.
- *Intersection*: Verifica che le fratture si intersechino correttamente.
- *Touching*: Verifica che le fratture si tocchino ma non si intersechino.

◆ Test per la funzione memorizza_tracce

Questo test verifica che la funzione *memorizza_tracce* memorizzi correttamente le tracce nel DFN. In particolare abbiamo verificato la correttezza dei vertici delle fratture dopo la memorizzazione delle tracce.

◆ Test per la funzione interseca_segmenti

Questo test verifica che la funzione *interseca_segmenti* calcoli correttamente l'intersezione tra due segmenti. Abbiamo verificato che:

- *NoIntersection*: non c'è intersezione tra i segmenti;
- *Intersection*:Verifica che i segmenti si intersechino e che il punto di intersezione sia corretto.

◆ Test per la funzione aggiorna_mesh

Questo test verifica che vengano aggiornati correttamente:

- il numero di celle 0D,1D e 2D;
- i vertici delle celle 1D e 2D;
- i markers delle nuove celle 1D.

◆ Test per la funzione nuovo_poligono

Questo test verifica che la funzione *nuovo_poligono* crei correttamente i nuovi poligoni dopo un taglio su un poligono esistente in una mesh poligonale. In particolare controlla che i nuovi lati e vertici del poligono risultante siano correttamente aggiornati sia nel caso di un poligno nomale che nel caso di un triangolo.

```
[----------] Global test environment tear-down
[==========] 14 tests from 10 test suites ran. (13 ms total)
[  PASSED  ] 14 tests.
```

Figure 27. Output del terminale relativo ai GoogleTest

DOCUMENTAZIONE UML

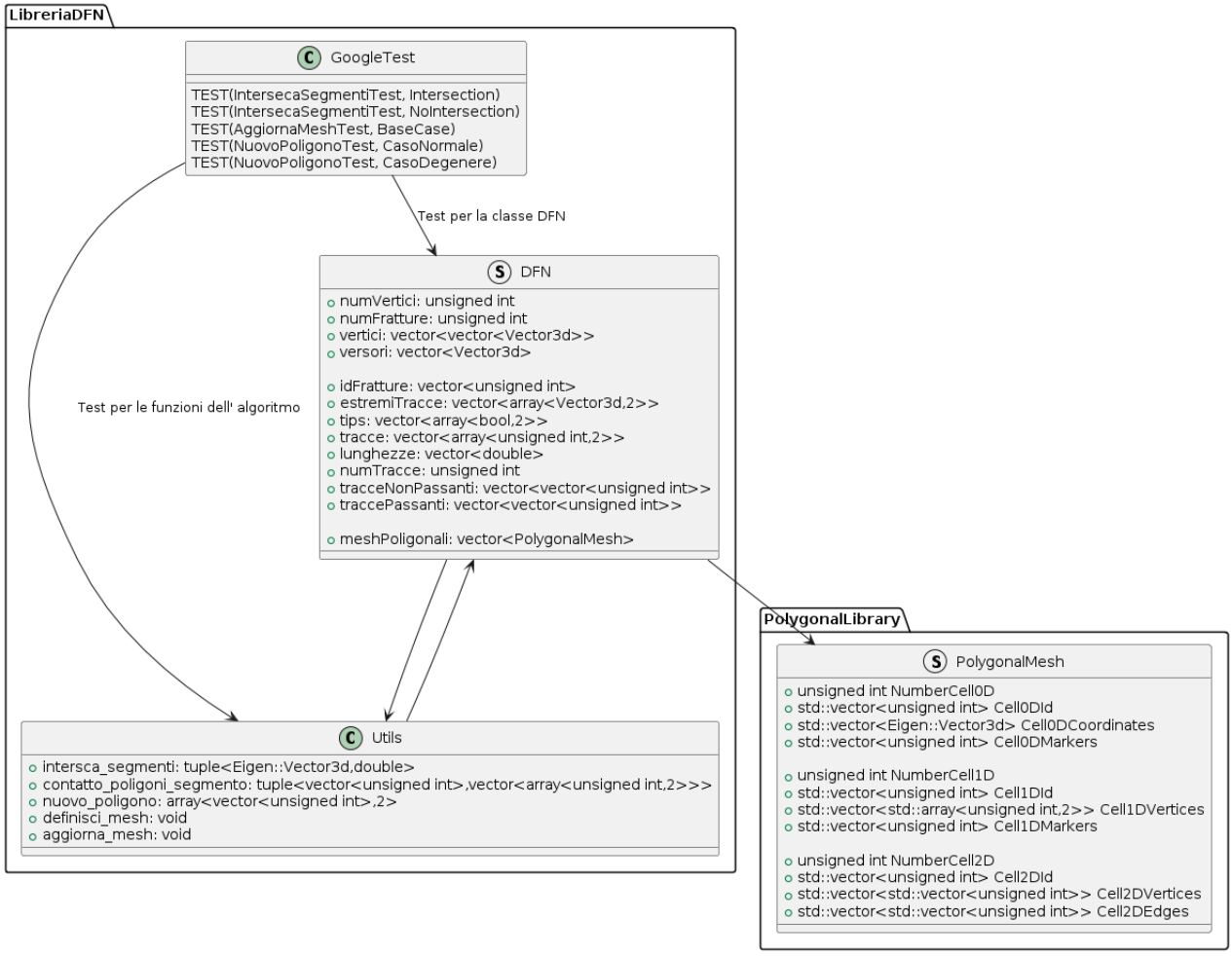


Figure 28. Diagramma delle classi della parte 2 del progetto in cui specifichiamo le relazioni tra le varie classi e strutture.

QUALCHE OUTPUT OTTENUTO

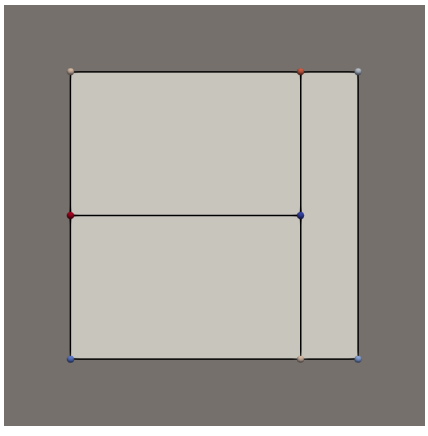


Figure 29. FR3

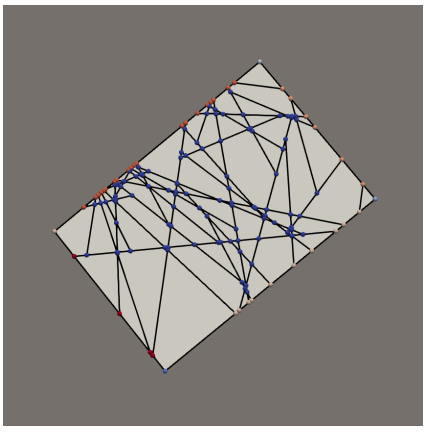


Figure 30. FR50

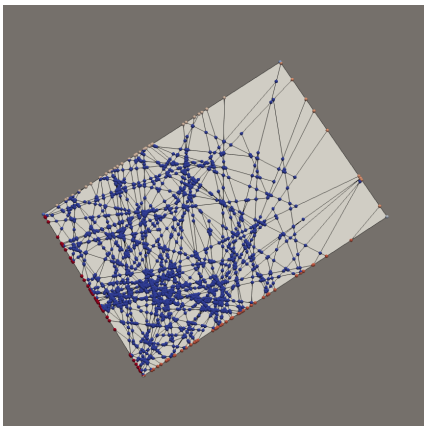


Figure 31. FR200

Figure 32. Immagini ricavate usando *Paraview* di tre dataset forniti.In queste immagini vengono evidenziati i marker delle Celle 0D. Quindi in nero sono rappresentati i lati dei sotto-poligoni, in bianco i poligoni.

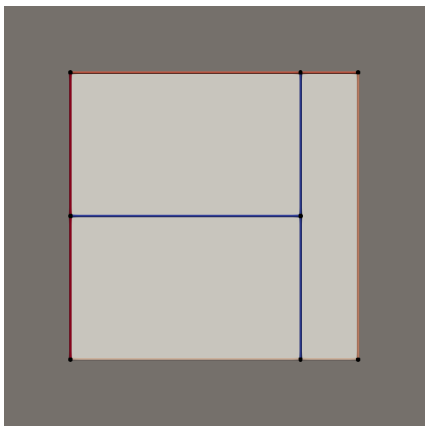


Figure 33. FR3

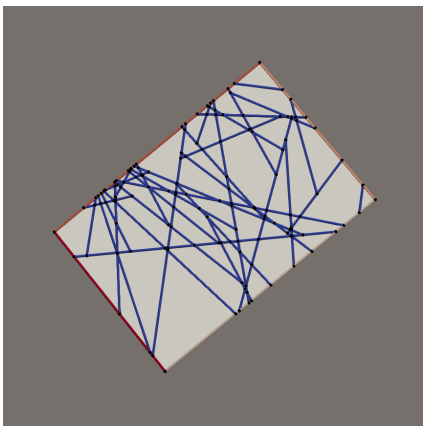


Figure 34. FR50

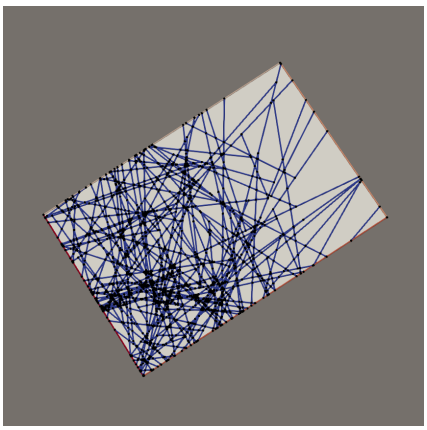


Figure 35. FR200

Figure 36. Immagini ricavate usando *Paraview* di tre dataset forniti. In queste immagini vengono evidenziati i marker delle Celle 1D. Quindi in nero sono rappresentati i vertici dei sotto-poligoni, mentre in bianco i poligoni.