POLITECNICO DI TORINO

Progetto PCS: Tracce e Fratture

Alessio Massenzana 273845
Francesco Licitra 282697
Alexis Laurent 298576

Anno Accademico 2023-2024

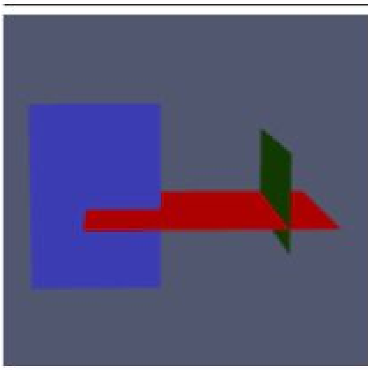
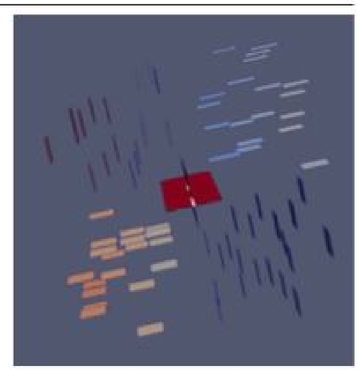
1 Discrete Fracture Network

Esibiamo una breve relazione riguardo al problema di Discrete Fracture Network (DFN). Si tratta di un sistema costituito da N fratture $F_n, n \in \{1, \ldots, N\}$, rappresentate da poligoni planari che si intersecano tra di loro nello spazio tridimensionale. Le M intersezioni (chiamate tracce) tra le fratture $F_m, m \in \{1, \ldots, N\}$, possono essere identificate da un segmento, supponendo di non considerare le tracce di misura nulla. Per ciascuna frattura, una traccia può essere passante o non-passante.
Una traccia passante per una frattura è un segmento con entrambi gli estremi che giacciono sul bordo della frattura stessa. Al contrario, una traccia non-passante per una frattura è un segmento che ha almeno un suo estremo all'interno della frattura stessa.
Faremo due assunsioni:

- la frattura F_n è rappresenta univocamente da poligoni convessi semplici definiti da V vertici non allineati ordinati in senso antiorario;
- la traccia T_m è condivisa da esattamente due fratture $T_m := F_i \cap F_j$. Esiste pertanto una relazione biunivoca tra l'identificatore della traccia m e la coppia degli indici delle fratture (i,j) che la definiscono.

Dato in input un DFN, il progetto è costituito da due parti:

1. Determinare le M tracce del DFN. Per ciascuna frattura, differenziare le tracce in passanti e non-passanti. Infine, per ciascuna frattura ordinare separatamente i due sottoinsiemi di tracce passanti e non-passanti per lunghezza in ordine decrescente;
2. Per ciascuna frattura, determinare i sotto-poligoni generati dal taglio della frattura con le sue tracce. Procedere al taglio della frattura seguendo prima l'ordine prestabilito dall'insieme di tracce passanti e successivamente quello delle tracce non-passanti.

DFN con 3 fratture	DFN con 10 fratture	DFN con 82 fratture
		

2 Strutture dati

Le strutture dati e le funzioni definite sono state suddivise nei 3 file DFNLibrary.hpp, PolygonalMesh.hpp, DFNLibrary.cpp. Nei file DFNLibrary si vanno a definire le strutture e le funzioni necessarie per la prima parte del progetto (per ogni frattura si ricercano le tracce, passanti e non passanti,e le si ordinano in ordine decrescente di lunghezza), mentre nel file PolygonalMesh.hpp si vanno a definire strutture dati che modellizzano oggetti (triangoli e poligoni) geometrici e funzioni che non hanno nulla a che fre con un DFN ma solo con la geometria.

2.1 Strutture e funzioni

Per affrontare la prima parte del progetto abbiamo definito tre strutture:

- **Frattura**: rappresenta una frattura composta dall'identificatore univoco, dal numero di vertici che la definiscono, dai vertici del poligono (come vettori di punti dello spazio ordinati in senso antiorario) e da un vettore di coppie in cui la prima componente è un puntatore ad una traccia da lei generata e un booleano che definisce se per la frattura in questione, la traccia generata è passante o non-passante. Include, inoltre, una serie di metodi per calcolare il centro, il raggio di una sfera che circoscrive la frattura e il piano della frattura (void computeCenter(), void computeRadius(), void computePlane()).

- **Traccia**: rappresenta una traccia composta dall'identificatore univoco, due vettori tridimensionali di Eigen che rappresentano gli estremi del segmento, dall'id delle fratture che generano la traccia (int idF1, idF2) e dal metodo double length() usato per calcolare per calcolare la lunghezza della traccia.

- **DFN**: rappresenta una rete di fratture e tracce, e include metodi per calcolarne proprietà, generare visualizzazioni, elaborare interazioni tra fratture e gestire l'output dei dati.
  Al suo interno possiede il numero di fratture, il vettore delle fratture che lo compongono, con la lista delle traccie generate.
  Infine include 3 funzioni (void computeDFN(), void plotFracture(), void output())per l'utilizzo dell'oggetto stesso.

Per affrontare invece, la seconda parte del progetto abbiamo definito la seguente struttura:

- **PolygonalMesh**: rappresenta una mesh poligonale, suddivisa in celle di diverse dimensioni (0D, 1D, 2D).

  Le Cell0D, che rappresentano i punti, sono composte dall'informazione del numero di Cell0D presenti all'interno della mesh, da un vettore che contiene gli identificatori univoci per ogni Cell0D e da un vettore contenente le coordinate dei punti.

  Le Cell1D, che rappresentano i segmenti, sono composte dell'informazione del numero di Cell1D presenti all'interno della mesh, da un vettore che contiene gli identificatori univoci per ogni Cell1D e da un vettore che contiene gli indici (ossia i riferimenti agli identificatori e alle coordinate delle Cell0D) che rappresentano i vertici di ogni segmento.

  Le Cell2D, che rappresentano i poligoni, sono composte dall'informazione del numero di Cell2D presenti all'interno della mesh, da un vettore contenente, per ogni poligono, un vettore con gli identificatori univoci delle celle0d componenti la Cella2D. Infine un vettore che, per ogni poligono, contiene il vettore degli identificativi delle Celle1D che la compongono.

2.2 Funzioni

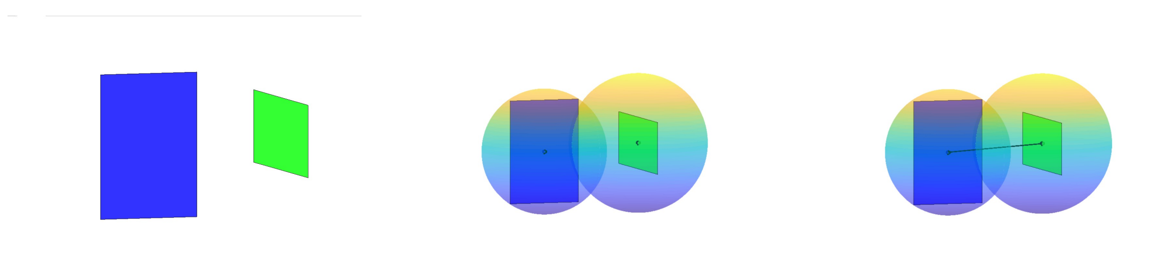
La funzione importDFN inizia a costruire l'oggetto dfn grazie ai dati presi da un file di testo denominato path. La funzione checkIntersection prende in input due fratture e cerca di capire se si può escludere a priori la possibilità che ci sia intersezione (i dettagli verranno discussi in seguito). La funzione elaborateV, se la checkIntersection non interrompe la ricerca della traccia va a calcolare effettivamente gli estremi del segmento di intersezione e a metterli all'interno dell'oggetto DFN. La funzione compareTrace prende due tracce e ne restituisce un booleano secondo una relazione d'ordine totale (sarà utile per l'ordinamento della lista tracce). La funzione CutDFN andrà ad innescare il processo di taglio delle fratture del dfn secondo le tracce trovate dalla computeDFN, inoltre andrà a chiamare ricorsivamente la funzione CutPolygon, quest'ultima data la traccia effettua il taglio del poligono e successivamente va ad eseguire delle operazioni per mantenere la coerenza dell'intera mesh. La funzione prolunge() calcola l'intersezione tra il prolungamento della traccia e un lato della frattura, se questo esiste ritornerà true altrimenti false. La funzione stadentro() prende in input una traccia e una frattura e ritorna un booleano che indica se la traccia va elaborata sul poligono in questione.

3 Calcolo delle tracce

La prima parte del progetto riguarda il calcolo delle tracce, svolto dalla funzione DFNLibrary::DFN::computeDFN con l'ausilio delle funzioni DFNLibrary::checkIntersection, DFNLibrary::DFN::elaborateV.
La checkIntersection effettua dei controlli preliminari, per ottimizzare e per escludere a priori delle intersezioni in casi particolari.

3.1 Ottimizzazione: Bolle e distanze

Il primo processo di ottimizzazione si innesca quando le fratture in questione sono molto distanti tra loro. Viene calcolata la sfera contenente la frattura in questione calcolando centro e raggio. Questi vengono calcolati dalle due funzioni Vector3d computeCenter(), double computeRadius() interne alla struct Frattura. Se la somma dei raggi delle due sfere contenente le fratture è minore della distanza dei centri escludiamo a priori l'intersezione $\Rightarrow$ non si generano tracce.
Nell'esempio mostrato in Figura 1, essendo che le due sfere si intersecano non riusciamo a escludere l'intersezione a priori.



(a) Fratture nello spazio

(b) Creazione sfere contenenti le fratture

(c) Calcolo la distanza tra i centri

Figura 1: rappresentazione grafica del controllo sulle bolle

3.2 Ottimizzazione: Separazione dei piani

Il secondo processo di ottimizzazione è basato sul fatto che i punti dello spazio sostituiti all'interno dell'equazione di un piano assumono un determinato segno.
Dato un punto (x_0, y_0, z_0) e data l'equazione di un piano π definita da:

$$\pi : ax + by + cz + d = 0 \quad \text{con} \quad a, b, c \in \mathbb{R}$$

Se il punto (x_0, y_0, z_0) appartiene al piano π, le sue coordinate soddisfano l'equazione del piano π, quindi questo significa che:

$$ax_0 + by_0 + cz_0 + d = 0$$

Se il punto (x_0, y_0, z_0) non appartiene al piano π, le sue coordinate non soddisfano l'equazione del piano π,quindi questo significa che:

$$ax_0 + by_0 + cz_0 + d \neq 0 \Rightarrow \begin{cases} ax_0 + by_0 + cz_0 + d > 0 & \text{se il punto si trova da una parte del piano} \\ ax_0 + by_0 + cz_0 + d < 0 & \text{se il punto si trova dall'altra parte del piano} \end{cases}$$

Il segno del risultato dipende dall'orientazione del piano e dalle coordinate del punto rispetto al piano. Il segno del risultato indica da quale lato del piano si trova il punto rispetto al piano stesso.

Per questo motivo, consideriamo il piano π_1, contenente la frattura F_1, e sostituiamo all'interno della sua equazione i vertici della frattura F_2. Se questi restituiscono tutti lo stesso segno (o tutti positivi o tutti negativi) escludiamo l'intersezione $\Rightarrow$ non si generano tracce.

Altrimenti provo ad eseguire lo stesso procedimento con il piano π_2, contenente la frattura F_2, contro i vertici della frattura F_1. Anche in questo caso se si ottengono valori tutti lo stesso segno (o tutti positivi o tutti negativi) escludiamo l'intersezione $\Rightarrow$ non si generano tracce.
Resta da escludere solo un ultimo caso di non intersezione quando questi controlli falliscono entrambi che verrà trattato in seguito.

3.3 Calcolo della retta di intersezione

Arrivato a questo punto la maggior parte dei casi di non intersezione è stata esclusa. Rimane da trattare il caso in cui [...]
Iniziamo ora la ricerca delle tracce. Cominciamo trovando la retta di intersezione r tra i piani su cui giacciono le due fratture, ovvero:

$$r : \begin{cases} \pi_1 : a_1x + b_1y + c_1z + d_1 = 0 & \text{piano contenente la frattura } F_1 \\ \pi_2 : a_2x + b_2y + c_2z + d_2 = 0 & \text{piano contenente la frattura } F_2 \end{cases}$$

Siano (a_1, b_1, c_1) e (a_2, b_2, c_2) le giaciture rispettivamente di π_1 e π_2 (ovvero le normale ai piani) e d_1 e d_2 i loro termini noti.
La direzione di r sarà perpendicolare ad entrambe le normali uscenti dai piani, per cui si ottiene nel modo seguente:

$$v_1 = [a_1, b_1, c_1] \times [a_2, b_2, c_2]$$

Il punto delicato è trovare un punto appartenente alla retta r che richiede la risoluzione del sistema

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

Per risolvere all'interno del calcolatore questo sistema abbiamo utilizzato la decomposizione QR implementata all'interno della libreria Eigen. Questa è la migliore dal punto di vista computazionale per sistemi non quadrati [Costo computazionale $\mathcal{O}(\frac{2n^3}{3})$].

3.4 Calcolo dei punti di intersezione

A questo punto vogliamo trovare i punti di intersezione tra la retta r e le due fratture F_1 ed F_2, per far ciò vado a scontare la retta r con tutti i lati di F_1 ed di F_2.
Ogni frattura per ipotesi rappresenta un poligono convesso, se considero un lato avente estremi A e B, allora i punti che giacciono sul segmento AB si possono esprimere come combinazione convessa degli estremi, ossia:

$$AB = \left\{ \mathbf{x} \in \mathbb{R}^3 : \mathbf{x} = s\mathbf{A} + (1 - s)\mathbf{B},\ s \in [0, 1] \right\}$$

Inoltre, la retta r può essere vista come

$$r = \left\{ \mathbf{x} \in \mathbb{R}^3 : \mathbf{x} = \mathbf{P} + v_1\mathbf{t},\ v_1 \in \mathbb{R} \right\}$$

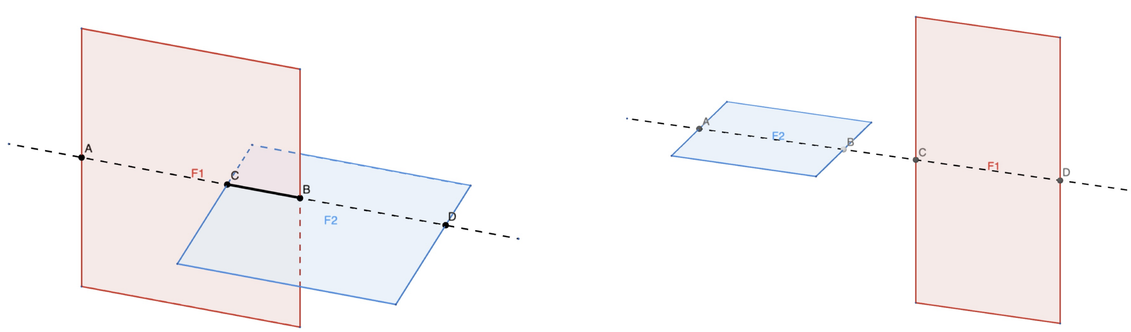
[controllo che forse sul codice non c'è.... non mi piacciono s $->$ alpha, v1 $->$ beta]
A questo punto dobbiamo trovare il punto di intersezione Q tra la retta r ($r : \pi_1 \cap \pi_2$) e il segmento

passante per i vertici ossia il lato della frattura.
Chiamando $v_2 = A - B$. Risolviamo il sistema

$$\begin{cases} Q = \mathbf{B} + \mathbf{v_2} s \\ Q = \mathbf{P} + \mathbf{v_1} t \end{cases}$$

Risolto il sistema dovremo effettuare un controllo sul valore di s: se l'intersezione è interna al segmento AB, il suo valore dovrà essere all'interno dell'intervallo [0,1] ($s \in [0, 1]$).
Per motivi computazionali dobbiamo adattare questo controllo a meno della tolleranza $\tau \Rightarrow$ a meno della tolleranza τ scelta: se $s \notin [0, 1]$, allora $Q \notin AB \Rightarrow s \in [0 - \tau, 1 + \tau]$.
Mentre il valore di v_1 costituisce l'ascissa curvilinea del punto Q sulla retta r rispetto al punto P.
Quando troviamo un punto di intersezione lo aggiungiamo al vettore d'appoggio v che verrà elaborato in seguito.
Reiterando questo processo per ogni lato di ciascuna frattura, F_1 ed F_2, sotto le ipotesi di convessità, alla fine di ogni ciclo otteniamo due punti di intersezione della retta r con ciascuna frattura.
Quindi, al termine di questi due cicli avremo il vettore d'appoggio v con quattro punti: due provenienti dall'intersezione della retta con F_1 e due provenienti dall'intersezione con F_2.

Indichiamo ora con A e B i punti di intersezione di r con la prima frattura F_1, mentre C e D quelli con la seconda F_2. Con l'implementazione di un BubbleSort, abbiamo ordinato tali punti sulla retta r in base all'ascissa curvilinea, in modo che siano tutti susseguenti.
Se l'ordinamento ha lasciato il vettore invariato, oppure se ha invertito i punti di F1 con quelli di F2, quella che avevamo trovato non era effettivamente un'intersezione.
Questo ordinamento ci permette di escludere l'ultimo caso di non intersezione che rimaneva da trattare, che sfuggiva dal secondo controllo preliminare della checkIntersection, ovvero quando sia quando sostituiamo all'interno dell'equazione del piano contenete F_1 i vertici della frattura F_2, sia quando sostituiamo all'interno dell'equazione del piano contenete F_2 i vertici della frattura F_1, questi restituiscono segni diversi.



(a) ordinamento punti: intersezione efettiva

(b) ordinamento punti: non intersezione

3.5 Definizione della traccia: casistiche possibili

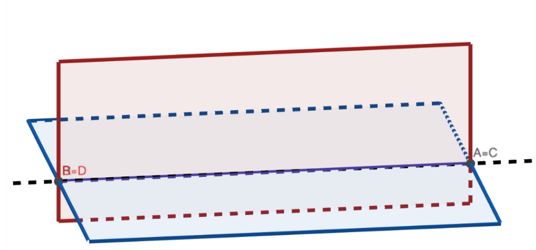
Trovati i quattro punti di interesse ed esclusa l'ultima situazione di non intersezione, la checkIntersection ritornerà valore positivo alla computeDFN() e in questo caso verrà chiamata la funzione elaborateV. Questa, presi in input gli id delle due fratture e il vettore v coi quattro punti di intersezione, ha il compito di trovare la traccia e definire per quali fratture è passante e per quali è non-passante.
Per far ciò conta preliminariamente il numero di punti coincidenti presenti nel vettore v.
Possiamo analizzare tre casistiche possibili, rispetto al numero di punti coincidenti trovati (in seguito indico col temine 'coppia' quando tra i quattro punti di interesse un punto generato da $F_1 \cap r$ coincide con un punto generato da $F_1 \cap r$) :

1. Se ci sono due coppie, vuol dire che i due punti di F1 coincidono con quelli di F2. In questo caso la traccia è passante per entrambe le fratture.

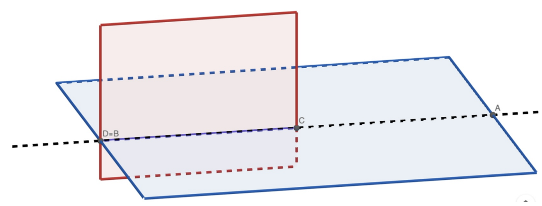
2. Nel caso ci sia una sola coppia, significa che per una frattura la traccia sarà passante, mentre per l'altra no. Per determinare la frattura per cui è passante andiamo a calcolare la lunghezza del segmento generato da entrambe le fratture, quella più corta sarà la frattura per cui la traccia è passante, e inoltre la traccia è data proprio dai suoi punti di intersezione con r.

3. Il caso di zero coppie comprende due diverse situazioni:

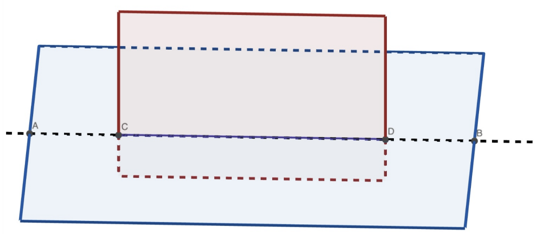
   - **Una frattura attraversa l'altra** Riordinando il vettore v, i punti della traccia si troveranno in posizione 1 e 2 (Interni). Se questi appartengono alla stessa frattura, allora la traccia è passante per la frattura in questione e non passante per l'altra.
   - **Traccia semplice** Nel caso in cui non si verifichi la situazione sopra, la traccia è non passante per entrambi.



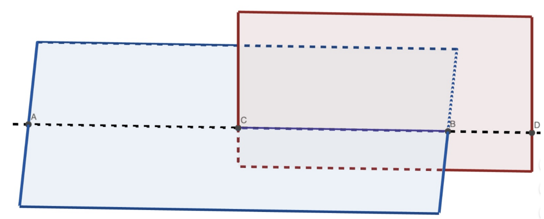
(c) prima casisitica descritta



(d) seconda casisitica descritta



(e) terza casistica descritta: una traccia attraversa l'altra



(f) terza casistica descritta: traccia semplice

3.6 Sort e Compare Trace

Una volta conclusa la funzione computeDFN() avremo all'interno del nostro oggetto dfn, tutte le fratture e le tracce generate. Viene ora chiamata la funzione output() che ha il compito di stampare su due file i risultati, secondo il formato richiesto.
Nel primo file, nominato "Output.txt", verranno stampate le tracce, seguendo lo schema

$$\{idTraccia\}\ \{idF1\}\ \{idF2\}\ \{Coordinate\ origine\}\ \{Coordinate\ fine\}.$$

Nel secondo file, "OutputFractures.txt", verranno invece stampate tutte le tracce di ogni frattura. Quest'ultime devono essere ordinate, mettendo prima le passanti e poi le non passanti, infine per ordine di lunghezza (decrescente). Questo compito viene svolto dalla std::sort((F.tracce).begin(), (F.tracce).end(), compareTrace) che presi in input l'inizio e la fine di una struttura dati (in questo caso la lista delle tracce di F), va a riordinarla secondo la relazione data dalla compareTrace. Dove la funzione compareTrace va a definire una relazione d'ordine, questa prende in input due tracce e ritorna un booleano, di risposta alla domanda " $(F1 < F2)$? ".

Infine, per ogni frattura verrà quindi stampato il risultato dell'ordinamento col seguente schema

$$\{idTraccia\}\ \{passante\ /\ non\}\ \{lunghezzaTraccia\}.$$

4 Calcolo della mesh poligonale

La seconda parte del progetto riguarda il taglio delle tracce, svolto dalla funzione cutPolygon con l'ausilio delle funzioni extendTrace(), cutIntersection() e isInside(). Il processo per il taglio è innescato dalla funzione cutDFN(), che prende in input l'oggetto DFN& dfn precedentemente calcolato e per ogni frattura, definisce un oggetto PolygonalMesh, nel vettore globalMesh. Quindi, inizializzato l'oggetto PolygonalMesh coi dati della frattura viene richiamato l'algoritmo ricorsivo cutPolygon.

4.1 CutPolygon

La funzione CutPolygon è definita ricorsivamente. Prende in input l'oggetto $list < pair < Vector3d, Vector3d >>$ $ListaTagli$ contenente i tagli, ordinati, ancora da effettuare, PolygonalMesh mesh è la mesh da aggiornare coi tagli effettuati e unsigned int idP, l'identificativo del poligono da elaborare.

Quindi per cominciare viene prelevato dalla lista il taglio da elaborare. All'interno di un ciclo vengono ricercati i due lati interessati dal taglio, memorizzati nel vettore lati[2], e i punti di intersezione nel vettore inters[2].

Adesso inseriamo i nuovi punti e i nuovi lati all'interno della mesh e creiamo quattro nuovi oggetti: PDEdges, PSEdges, PDVertices, PSVertices. Rispettivamente andranno a contenere i lati del poligono destro e sinistro, generati dal taglio, e i loro vertici.

Per il taglio andremo quindi a processare il poligono P iniziale andando a creare passo-passo i due sotto poligoni PS e PD.

Infine per tagliare lati e punti sarà utile l'introduzione di un vettore *bool* $inizioFine[2]$ per aiutarci a capire se al momento ci troviamo sul poligono sinistro o su quello destro.

4.2 Taglio dei lati del Poligono

Ciclando sul poligono P, iniziamo ad aggiungere i suoi lati al poligono sinistro. Quando verrà elaborato il primo lato interessato dal taglio (lati[0]), verranno aggiunti sul Poligono sinistro la corrispondente porzione del lato interessato, e il nuovo lato.

A questo punto ci spostiamo sul poligono destro (inizioFine[0] = 1): aggiungiamo la restante porzione del lato tagliato, e continuiamo aggiungendo i successivi lati di P. Quando dovremo elaborare lati[1], ovvero il secondo lato da tagliare, chiudiamo il poligono destro aggiungendo la sua porzione di tale lato, e il lato nuovo (inizioFine[1] = 1).

Rispostandoci su PS aggiungiamo la restante porzione di lati[1] e aggiungiamo i rimanenti lati di P.

4.3 Inserimento nuovi vertici

Per semplicità trattiamo i vertici di P in un secondo ciclo: Ciclando sui punti di P, iniziamo ad aggiungerli su PS. Quando troveremo che inters[0] appartiene al lato generato dal punto i-esimo col successivo, aggiungeremo il punto i a PS e inters[0] ad entrambi i poligoni. Impostiamo inizioFine[0] = 1. Ora, continuinamo sul poligono destro, aggiungendo i punti di P finchè il punto i-esimo e il successivo non contengano inters[1]. Aggiungiamo quindi a PD il punto i, ad entrambi i sotto-poligoni inters[1] e infine aggiorniamo inizioFine[1] = 1. Aggiungiamo i rimanenti vertici a PS.

N.B. Prima di innescare il processo di ricorsione dobbiamo prima prestare attenzione ad un dettaglio: punti e lati di un Poligono sono oggetti indipendenti e non è quindi detto che arrivati a questo punto PSVert e PSEdges siano riferiti allo stesso sotto-poligono. Andiamo a controllare che un vertice di PSVert sia appartenente a un lato ($PSEdges[i][j]$). Se questo non avviene invertiamo i riferimenti PSVert e PDVert.

4.4 Aggiornamento mesh e Ricorsione

Per dare il via alla ricorsione ci manca ora da definire gli ultimi due oggetti: $listaTagliDx$ e $listaTagliSx$. Grazie alle funzioni isInside() e cutIntersection() riusciamo a definire se una traccia sta all'interno del poligono destro, in quello sinistro oppure dentro entrambi. Andiamo quindi a svuotare l'oggetto listaTagli distribuendo i suoi oggetti sulle due nuove liste.

Andiamo ora ad aggiornare la mesh, sovrascrivendo il poligono P con il poligono PD e facendo una pushback degli oggetti di PS (PSVert e PSEdges).

Richiamiamo infine l'algoritmo cutPolygon(listaTagliDx, mesh, idPD) sul poligono PD e cutPolygon(listaTagliSx, mesh, idPS) sul poligono PS, quando le loro liste dei tagli sono non-vuote.

5 Test e documentazione UML

5.1 Test di verifica del codice

Durante la fase di scrittura del codice, in seguito all'implementazione di un metodo di una classe, abbiamo provveduto a creare dei test che ci hanno permesso di verificarne il corretto funzionamento. Abbiamo costruito tali test in modo che ci fornissero in output ciò che ci aspettavamo dalle funzioni o metodi.

- **TEST(DFN_UTILITIES, TestComputeCenter):** Va a verificare che i centri di due fratture, calcolati con la Frattura.computeCenter(), coincidano con i valori calcolati sulla carta, a meno di una tolleranza fissata *testtol*.

- **TEST(DFN_UTILITIES, TestComputeRadius):** Verifica che il raggio della bolla che inscrive una frattura sia uguale al valore teorico, a meno della tolleranza.

- **TEST(DFN_UTILITIES, TestComputePlane):** Testa che il piano contenente la frattura sia corretto, a meno di *testtol*.
- **TEST(DFN_UTILITIES, TestTraceLength):** Testa il funzionamento della TraceLength(), a meno della tolleranza.
- **TEST(DFN_UTILITIES, TestSign):** Verifica il corretto funzionamento della funzione sign(d).
- **TEST(DFN_UTILITIES, TestCompareTrace):** Testa il funzionamento di CompareTrace();
- **TEST(DFN_UTILITIES, TestLiesOnSegment):** Verifica il corretto funzionamento della funzione lies(A, B, P) coi casi limite P = A, P = B, P = A-tol, P = B-tol.
- **TEST(DFN_TESTS, TestImport):** Verifica che la funzione importDFN() restituisca valore positivo, nei dataset di base.
- **TEST(DFN_TESTS, TestLiesOn):** Sul dataset DFN3, verifica che le tracce calcolate interessino i lati corretti.
- **TEST(DFN_TESTS, TestDFN3):** Sul dataset DFN3, verifica che le tracce calcolate siano corrette, a meno della tolleranza.
- **TEST(DFN_TESTS, TestDFN10):** Verifica sul dataset DFN10 che il numero di tracce trovate sia uguale al numero teorico (25).
- **TEST(DFN_TESTS, TestDFN82):** Verifica sul dataset DFN82 che ci sia solo una traccia.

5.2 Visualizzazione UML del codice