
Programmazione e Calcolo Scientifico

Laurea Triennale in Matematica per l'Ingegneria

# Relazione Progetto
# Discrete Fracture Network

*Sophie Vigè 295469*
*Alice Zurru 294176*

Anno Accademico 2023-2024

# Capitolo 1

# Determinare le tracce di un DFN

## 1.1 Scelta delle strutture dati

Per elaborare la prima parte del progetto si è scelto di definire due strutture, `Fracture` e `Trace`, entrambe racchiuse nel namespace `Geometry`, che rappresentassero rispettivamente le fratture e le tracce date dall'intersezione di due di esse (la relazione traccia-frattura è quindi "2 a molti").

Ogni frattura è caratterizzata da cinque attributitra cui i più rilevanti sono:

- id, memorizzato come `int` in modo da avere la possibilità di assegnare un valore "sentinella" (-1) alle fratture problematiche, in particolare quelle con lati di lunghezza inferiore alla tolleranza;
- un vettore (`std::vector`) contenente le coordinate dei vertici, memorizzate in tutto il programma come `Eigen::Vector3d` per potervi applicare delle operazioni (ad esempio farne la sottrazione o `.squaredNorm()`). Vengono memorizzate in un vettore perché se ne conosce già la dimensione (numero di vertici) quando viene creato e per l'efficienza delle operazioni di accesso;
- due `vector<unsigned int>` che contengono, rispettivamente, gli id delle tracce passanti e non passanti per la frattura considerata. La scelta di memorizzarli in vettori è dovuta alla necessità, in un secondo momento, di ordinarli.

Ogni traccia è caratterizzata da nove attributi, tra cui:

- un `array<Vector3d,2>` contenente le coordinate delle due estremità della traccia. In questo come in altri attributi si sceglie di usare `std::array` poiché la sua dimensione è nota a tempo di compilazione e non varia all'interno del programma;
- un `array<bool,2>` che segnala se e per quale frattura coinvolta si verifica il caso in cui essa è "appoggiata" sul piano contenente l'altra frattura;
- un `bool` e due vettori (`vector<Vector3d>` e `vector<unsigned int>`) che serviranno poi nella seconda parte (2.2.3) per indicare se la traccia in questione è intersecata da altre tracce e, in caso positivo, per memorizzarvi coordinate e id dei vertici generati dalla traccia corrente.

## 1.2 Descrizione degli algoritmi e funzioni

L'algoritmo che determina le traccia per ogni frattura è incentrato sull'idea di base che se due poligoni si intersecano (esclusi casi particolari trattati diversamente) allora accade che, preso il piano su cui giace

un poligono e considerando lo spazio 3D tagliato a metà dal piano, i vertici dell'altro poligono non si troveranno tutti dalla stessa parte del piano (e viceversa). In questo modo è possibile trovare esattamente quale sia il lato di un poligono che interseca il piano dell'altro poichè corrisponde al lato tra i due vertici tali che uno di essi si trova da una parte rispetto al piano dell'altro poligono e l'altro dall'altra parte (evitando, ad esempio, di dover calcolare l'eventuale intersezione di ogni lato con il piano e i costi computazionali che ne derivano). Di seguito sono elencati più precisamente i passaggi che sono stati implementati:

### 1.2.1 Controlli iniziali

Per ogni coppia di fratture è analizzata nella principale funzione `findTraces` l'eventuale intersezione. Prima di dare via all'algoritmo vero e proprio, tuttavia, vengono effettuati vari controlli per evitare di verificare l'esistenza di una traccia tra fratture in casi particolari e rendere il programma più efficiente. Il primo controllo che viene effettuato, nella funzione `passBoundingBox` esclude il caso che venga controllata l'intersezione tra due fratture che sono molto lontane nello spazio. In particolare si calcola un punto che sarà il centro di una sfera facendo la media delle coordinate di tutti i vertici. Successivamente si prende come raggio di essa il massimo tra le distanze dei vertici dal centro, per assicurarsi che tutto il poligono sia contenuto all'interno della sfera considerata. Il procedimento è ripetuto per entrambi i poligoni e infine si controlla se la distanza tra i centri delle due figure sia minore della somma dei raggi (per evitare il calcolo costoso della radice quadrata si lavora con la distanza al quadrato). In tal caso, si prosegue con il calcolo dell'eventuale frattura, altrimenti siamo certi che non c'è intersezione.

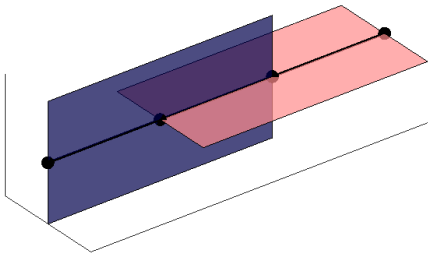
Un ulteriore controllo viene eseguito dopo che è stata trovata l'equazione di entrambi i piani (tramite la funzione `findPlaneEquation`) su cui giacciono le due fratture (in `findIntersectionPoints`) escludendo il caso in cui i due piani siano paralleli.

### 1.2.2 Determinare se due fratture generano una traccia

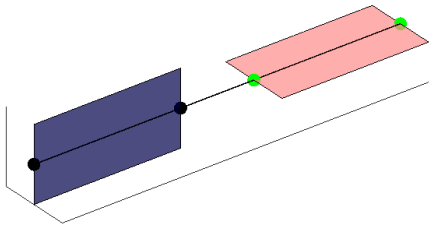
Nella funzione `findIntersectionPoints`, nel caso in cui una coppia di fratture abbia passato i controlli iniziali, si comincia a controllare da che parte siano i vertici di un poligono rispetto al piano su cui giace l'altro. In particolare si valuta l'equazione del piano nelle coordinate del vertice e si controlla se il risultato è positivo o negativo. Considerando lo spazio tridimensionale tagliato a metà dal piano su cui giace il poligono, si parte dalla zona in cui si trova il primo vertice (per i casi particolari legati ad esempio a vertici esattamente sul piano dell'altro poligono si rimanda alla sezione apposita), si scorrono i vertici del poligono finchè non si incontra un vertice che si trova nella parte opposta. A questo punto si conta il numero di vertici che stanno dall'altra parte per determinare esattamente quali saranno i lati del poligono che intersecano il piano dell'altro poligono. Una volta incontrato un vertice che ritorna ad essere nella prima zona dello spazio il controllo termina poiché tutti i successivi punti saranno ancora da quella parte, essendo i poligoni convessi. In particolare i lati che intersecano il piano sono quelli che hanno come estremità due vertici che si trovano nelle parti opposte dello spazio. Questa strategia permette di evitare il calcolo dell'intersezione di ogni lato con il piano dell'altro poligono, considerato l'elevato costo computazionale (la risoluzione di un sistema lineare a $n$ inognite ha un costo di $\mathcal{O}(n^3)$). Si è, infatti, considerato che i poligoni potessero avere anche un numero elevato di lati.

Se dal controllo risulta che almeno per uno dei due poligoni tutti i vertici si trovano dalla stessa parte dello spazio tridimensionale rispetto al piano dell'altro poligono, si conclude che non c'è intersezione e si può passare alla coppia di fratture successiva. Alternativamente vengono trovati per ognuno dei poligoni i due punti di intersezione con il piano dell'altro: in tutto si ottengono 4 punti che sono sulla retta di intersezione tra i due piani su cui giacciono i poligoni. Per un esempio si veda la Figura 1.1a.

Infine, per essere certi che ci sia intersezione viene effettuato un ultimo controllo nella funzione `findInternalPoints` in cui si vede l'ordine con cui i 4 punti sopraccitati sono disposti sulla retta: se i due appartenenti a un poligono sono da una parte e i due appartenenti all'altro poligono si trovano successivamente senza mischiarsi non c'è intersezione, altrimenti si. Per un esempio di questo caso si veda la Figura 1.1b.



(a) Intersezione e rappresentazione dei 4 punti in nero.



(b) Non c'è intersezione perché i punti sono i primi due di un poligono e poi gli altri due dell'altro.

### 1.2.3 Identificazione delle estremità e tipologia di una traccia

Una volta trovati i 4 punti è necessario capire quali sono i punti che corrispondono alle estremità dell'intersezione: i due punti centrali tra i 4 considerati delimitano la traccia. Attraverso una serie di prodotti scalari, osservando se si ottiene un risultato positivo o negativo, si riesce a comprendere, in quale delle $\frac{4!}{2} = 12$ permutazioni (ignorando quelle speculari) si presentano i 4 punti. Infatti è sempre necessario considerare che si trovano tutti sulla stessa retta: la retta di intersezione dei due piani su cui giacciono le fratture. Varie situazioni sono possibili in base all'ordine dei punti:

1. se i primi due punti provengono da una frattura e gli ultimi due all'altra allora non c'è intersezione (ci sarebbe a meno di una traslazione lungo la retta di intersezione dei due piani). Questo caso è rappresentato in Figura 1.1b.

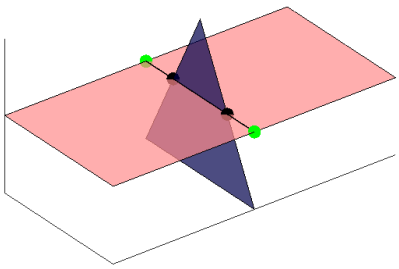
2. se i punti centrali provengono dalla stessa frattura e non c'è coincidenza con i punti dell'altra frattura allora la traccia sarà rispettivamente passante per la prima e non passante per la seconda (Figura 1.2a);

3. se i punti centrali provengono uno da una frattura e l'altro dall'altra frattura e non si è nel caso 1., allora la traccia è non passante per entrambi (Figura 1.2b).

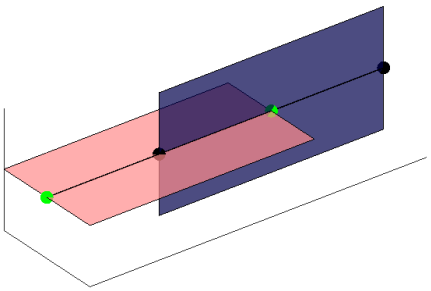
Vengono trattati separatamente i casi in cui 2 o più punti dei 4 coincidono a meno della tolleranza considerata:

- se i due punti del primo poligono coincidono con i due punti del secondo poligono, allora la traccia è passante per entrambi (si veda Figura 1.3a per un esempio);

- se solo uno dei punti del primo poligono coincide con un punto del secondo poligono vengono ancora analizzate le varie possibilità di tracce passanti e non passanti considerando lo stesso ragionamento spiegato in precedenza;

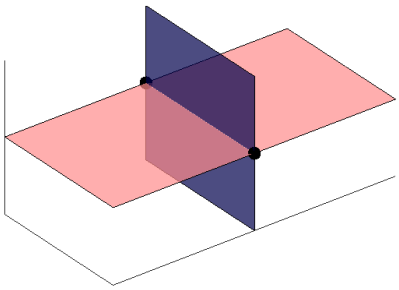


(a) I due punti centrali provengono dalla stessa frattura: per essa la traccia è passante.

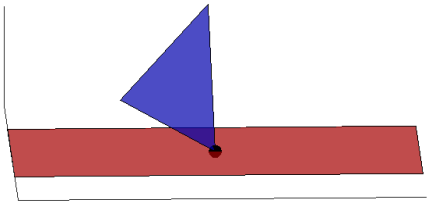


(b) I due punti centrali provengono da due fratture diverse: la traccia è non passante per entrambe.

- se i due punti centrali coincidono, la lunghezza della traccia sarebbe nulla e quindi non viene neanche salvata come traccia (controllo effettuato in `findTraces`). Esempio in Figura 1.3b.



(a) Traccia passante per entrambi.



(b) Non c'è traccia perché c'è un unico punto di intersezione.

### 1.2.4 Creazione delle tracce

Le tracce sono memorizzate all'interno di un vettore per facilitare l'accesso casuale, infatti in questo modo sarà possibile accedere ad ogni traccia con un costo $\mathcal{O}(1)$, contro l'accesso sequenziale ($\mathcal{O}(n)$) della lista. Infatti, nella parte 2 e per ordinarle, sarà necessario accedere alle tracce in modo semplice, considerando che nella `struct Fracture` sono memorizzate solo gli id delle tracce passanti e non.

Tuttavia, si vuole sfruttare la maggior efficienza delle liste nel salvare un numero non conosciuto di elementi perciò nella funzione `findTraces` le tracce trovate vengono di volta in volta aggiunte all'interno di una lista che contiene tutte le tracce. Solo successivamente, quando sarà noto il numero totale di tracce, vengono memorizzate all'interno di un vettore.

### 1.2.5 Caso particolare: poligoni che si intersecano ma non si attraversano

Il programma tiene conto di questo caso particolare attraverso l'`array<bool,2> onThePlane` per ogni traccia. Se, ad esempio, il suo primo valore è impostato su `vero`, significa che il primo poligono non passa

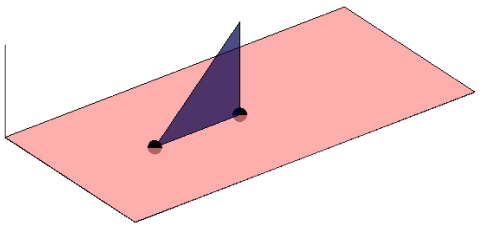


Figura 1.4

effettivamente attraverso l'altro, ma risulta solo "appoggiato" sul piano dell'altro poligono (si veda la Figura 1.4). In questo caso l'algoritmo usuale non è adatto ad evidenziare la presenza della traccia perciò si controlla se due punti adiacenti di un poligono si trovano sul piano dell'altro. In tal caso, quei punti diventano le estremità della traccia, perciò non è neanche necessario verificare l'ordine come spiegato nella sezione 1.2.3. Inoltre, la traccia sarà considerata passante per il primo poligono.

### 1.2.6 Ordinamento tramite mergesort

Si è scelto di ordinare le tracce tramite l'algoritmo di `mergesort`, opportunamente modificato affinchè tenga conto delle lunghezze delle tracce. Infatti è noto che per un vettore senza proprietà particolari, questo tipo di algoritmo ha un costo computazionale di $\mathcal{O}(n\log(n))$, che non può essere migliorato.

### 1.2.7 Considerazioni sulla tolleranza

In tutto il codice si è scelto di utilizzare la tolleranza assoluta data dal massimo di quella fornita dall'utente e `10*numeric_limits<double>::epsilon()`.

Talvolta si è utilizzata una tolleranza al quadrato data dal massimo tra il quadrato della tolleranza precedentemente descritta e ancora `10*numeric_limits<double>::epsilon()`. In particolare, ciò è stato necessario nel momento in cui si sono calcolate delle distanze: per questioni di efficienza computazionale non si calcola la radice al quadrato ma si confronta il quadrato dei membri.

# Capitolo 2

# Determinare i sotto-poligoni generati per ogni frattura

## 2.1 Scelta delle strutture dati

Per la seconda parte del progetto si è scelto di utilizzare una struttura, chiamata PolygonalMesh, che ha una corrispondenza 1 a 1 con la classe Frattura e che è dotata di nove attributi:

- numero totale di vertici, di lati e poligoni salvati come `unsigned int`;
- due `vector<unsigned int>` per salvare tutti gli id di vertici e lati con i corrispondenti `vector<Vector3d>` per salvare le coordinate dei vertici e `vector<array<unsigned int, 2>>` per salvare gli id degli estremi per ogni lato. Nel caso delle coordinate dei vertici si è scelto di usare un `Eigen::Vector3d` per facilitare le operazioni matematiche sulle componenti dei vettori che rappresentano i punti;
- due `vector<vector<unsigned int>>` per salvare vertici e lati di ogni sotto-poligono; in particolare, si è scelto l'uso di un vettore piuttosto che di una lista per avere accesso casuale diretto. Tuttavia, per sfruttare la maggiore efficienza di inserimento di elementi della lista, spesso si è scelto di riempire inizialmente questo tipo di struttura e solo successivamente copiare i dati in un vettore, nel momento in cui la dimensione diventa nota.

## 2.2 Descrizione degli algoritmi e funzioni

L'obiettivo è costruire il `vector<PolygonalMesh>` dove in ogni posizione $i$ è memorizzata la mesh corrispondente alla frattura in posizione $i$-esima nel vettore già presente nella parte 1. Per fare ciò si usa una prima funzione `cutFractures` con lo scopo di gestirne una seconda (`makeCuts`) di tipo ricorsivo.

### 2.2.1 Inizializzazione della mesh

Per ogni frattura, all'interno della funzione `cutFractures`, innanzitutto si inizializza la mesh corrispondente e si iniziano ad aggiungere gli unici vertici già noti, ovvero quelli della frattura stessa. Poiché non si conoscono a priori i numeri di vertici e di lati della mesh, vengono inizializzate due liste dove sono salvati man mano che vengono eseguiti i tagli. In questo modo si ha una maggiore efficienza nell'inserimento (con un costo costante) e si inizializzano i vettori solo alla fine dei tagli ricorsivi in modo da evitare

i costi dovuti al ridimensionamento. Per non avere lati ripetuti, si utilizza una `map<array<unsigned int,2>,unsigned int>` di supporto in modo da poter verificare ad un costo limitato ($\mathcal{O}(logn)$) se un lato (caratterizzato dagli id dei due estremi) è già stato inserito o meno tra quelli della mesh. Si crea poi una lista di riferimenti a tutte le tracce che devono tagliare la frattura corrente, già ordinate nell'ordine in cui verranno "tagliate". È necessario usare i riferimenti per poter modificare le tracce originali e non solo una loro copia.

Si dà poi il via alla funzione ricorsiva che prende in input due code `std::queue` contenenti rispettivamente le coordinate e gli id dei vertici che costituiscono il sottopoligono da "tagliare". La scelta di utilizzare una coda è dovuta al fatto che sarà poi unicamente necessario aggiungere in coda, estrarre dalla testa ed eliminare l'elemento in testa. Tutte queste operazioni hanno un costo $\mathcal{O}(1)$ e rispetto a una lista si risparmia in memoria perché sono presenti meno puntatori per ogni nodo.

Una volta eseguite tutte le ricorsioni (descritte in 2.2.2). La funzione `cutFractures` trasforma in vettore le liste provvisorie e termina così la creazione della mesh.

### 2.2.2 Tagli ricorsivi

La funzione ricorsiva che si occupa del taglio dei poligoni lungo le tracce è `makeCuts`. Se quando essa viene chiamata la dimensione della lista di puntatori alle tracce che riceve in input (`traces`) ha dimensione non nulla, significa che sono ancora presenti tracce che devono tagliare il sottopoligono corrente e viene eseguito il taglio lungo la traccia in testa alla lista. Si distinguono il caso generale da quello particolare in cui la traccia cade sul lato del sottopoligono (questo caso particolare era già stato segnalato nella parte 1 dal booleano `onThePlane` come attributo delle tracce) e i due vengono trattati separatamente. Le tracce passanti e non passanti invece vengono trattate allo stesso modo.

- Caso generale: l'obiettivo è quello di popolare due nuovi insiemi di strutture dati in modo che contengano i nuovi vertici dei due sottopoligoni generati dal taglio. Si utilizza `findSideOfTheLine` per determinare, per ogni vertice del poligono corrente, da quale lato della retta contenente la traccia si trova (o se si trova su di essa). Questo permette di inserire i vertici già presenti nel primo sottopoligono, oppure nel secondo o in entrambi (nel caso in cui esso si trovi sulla retta). Ogni volta che accade che il vertice precedentemente analizzato si "trova" dal lato opposto di quello corrente, significa che quel lato viene tagliato dalla traccia. Si trova dunque l'intersezione tra rette (con `intersectionLines`) e si aggiunge questo punto ai vertici di entrambi i sottopoligoni. Si vuole anche aggiungere tale vertice alla mesh, ma solo se esso non è già presente. Per questo controllo si rimanda alla sezione 2.2.3. Si controlla allo stesso modo anche se c'è intersezione nell'ultimo lato. A questo punto si è finito di tagliare lungo la traccia corrente, dunque si verifica da che parte della retta si trovano le tracce rimanenti (possono anche trovarsi da entrambi i lati e in questo caso è necessaria un'accortezza aggiuntiva, spiegata in 2.2.3) e le si aggiunge alle sottoliste corrispondenti. Si rimuove infine la traccia appena tagliata e si chiama la funzione ricorsiva sui due sottopoligoni.

- Caso generale in cui il primo vertice cade sulla retta: poiché in questo caso il primo vertice è salvato in entrambi i sottopoligoni (e quindi non ha senso confrontare i vertici successivi con esso per capire in quale sottopoligono salvarli), si introduce una variabile booleana (`previousSide1`). In questo modo si decide da quale parte salvare un vertice confrontandola con quella dove si è salvato il vertice precedente (se "si è cambiato lato" della retta, lo si salva dalla parte opposta, altrimenti dalla stessa).

- Caso *onThePlane*: in questo caso il taglio della traccia corrente genera un unico sottopoligono uguale a quello precedente o con al più due vertici aggiuntivi sul lato interessato dalla traccia. I vertici del poligono sono aggiunti normalmente all'unico sottopoligono, ma quando ci si trova sul lato tagliato dalla traccia (ovvero quando sia il vertice corrente sia quello precedente cadono sulla retta) viene chiamata la funzione apposita `addVerticesOnThePlane` per aggiungere nell'ordine corretto anche i nuovi eventuali vertici. Questa prende in input i due estremi della traccia e quelli del lato di interesse e, dal momento che questi quattro punti cadono sulla stessa retta, con una serie di condizioni successive con prodotti scalari si determina se uno o entrambi gli estremi della traccia cadono all'interno del lato e in questo caso li si aggiunge nell'ordine corretto ai vertici del sottopoligono e alla mesh.

Se invece la dimensione di `traces` è nulla, significa che per quel sottopoligono non ci sono più tagli da effettuare. Si devono ora definire i lati del sottopoligono (dato che sono ora definitivi) e, se non già presenti, vanno aggiunti anche a quelli complessivi di tutta la mesh. Per fare ciò si scorrono tutti i vertici consecutivi del poligono e si verifica se la coppia dei loro id (che identifica dunque un lato) è già presente nella mappa dei lati (l'ordine degli estremi non conta) e in caso contrario si aggiunge il lato sia ad essa sia alla mesh.

### 2.2.3 Evitare di aggiungere duplicati alla mesh

Per non rischiare di inserire più volte lati coincidenti alla mesh si utilizza, come precedentemente spiegato, una mappa avete come chiave la coppia di id dei suoi estremi e come valore l'id del lato. Questo permette di verificare la presenza di un elemento ad un costo limitato.

Dal punto di vista computazionale sarebbe stato ottimo utilizzare una mappa anche per i vertici, ma risulta più complicato dal momento che si vogliono confrontare le coordinate a meno della tolleranza. Si è usata la seconda scelta (che, ogni volta che si vuole fare il controllo, costa $\mathcal{O}(n)$ contro $\mathcal{O}(log(n))$ della mappa), ovvero dei vettori dove salvare i vertici già inseriti della mesh e che potrebbero venire ripetuti. Per ammortizzare il costo nel caso peggiore $\mathcal{O}(n)$ dell'accesso ad ognuno di essi, vengono identificate le tracce che potrebbero generare vertici ripetuti, ovvero quelle che intersecano le tracce tagliate in precedenza e solo i vertici generati da queste tracce vengono salvati in questi vettori. Questa scelta risulta dunque più efficiente rispetto a salvare tutti i vertici già inseriti nella mesh perché permette di ridurne il numero e dunque i controlli che potenzialmente vanno fatti a ogni inserimento di un nuovo vertice. Tali informazioni vengono salvate direttamente nella traccia in questione, tramite tre attributi dedicati:

- un `bool` che indica se la traccia è una di quelle che potrebbe generare problemi nell'univocità dei vertici;
- un `vector<Vector3d>` dove si memorizzano le coordinate dei vertici generati dalla traccia (ovvero le "chiavi" da controllare che siano univoche a meno della tolleranza);
- un `vector<unsigned int>` dove, a ogni indice, corrisponde l'id del vertice le cui coordinate sono memorizzate nella posizione corrispondente del vettore precedente.

### 2.2.4 Nota sulla tolleranza

Si noti che, poiché non viene richiesto di produrre una mesh "di buona qualità", non viene effettuato il controllo per verificare se le aree dei poligoni sono maggiori di una certa tolleranza 2D. Tuttavia il

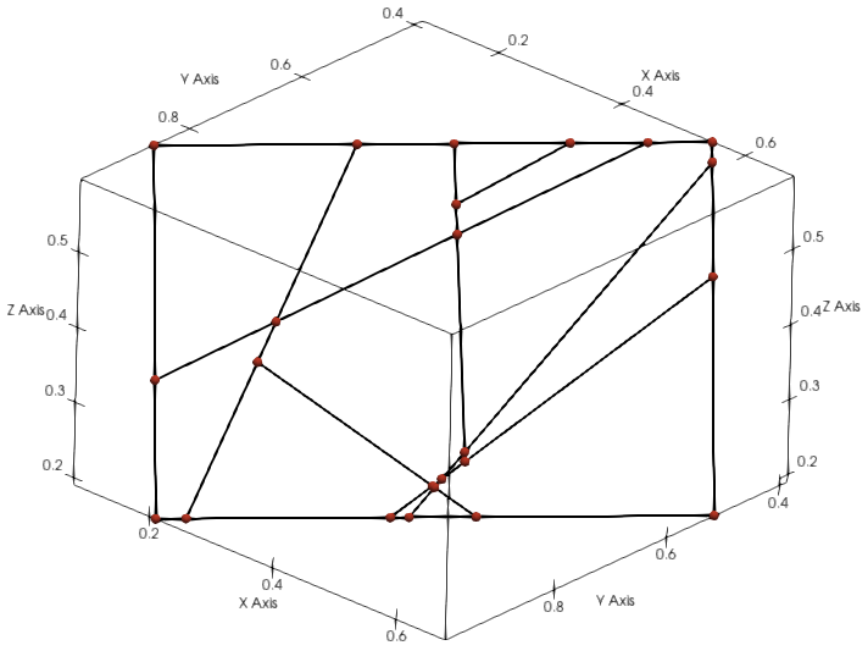



Figura 2.1: Esempio di mesh generata dal taglio di una frattura lungo le sue tracce.

controllo analogo sulla lunghezza dei lati si ottiene immediatamente dal fatto di aver incluso nei casi trattati quello in cui un vertice cade (a meno della tolleranza) sulla retta su cui giace la traccia.

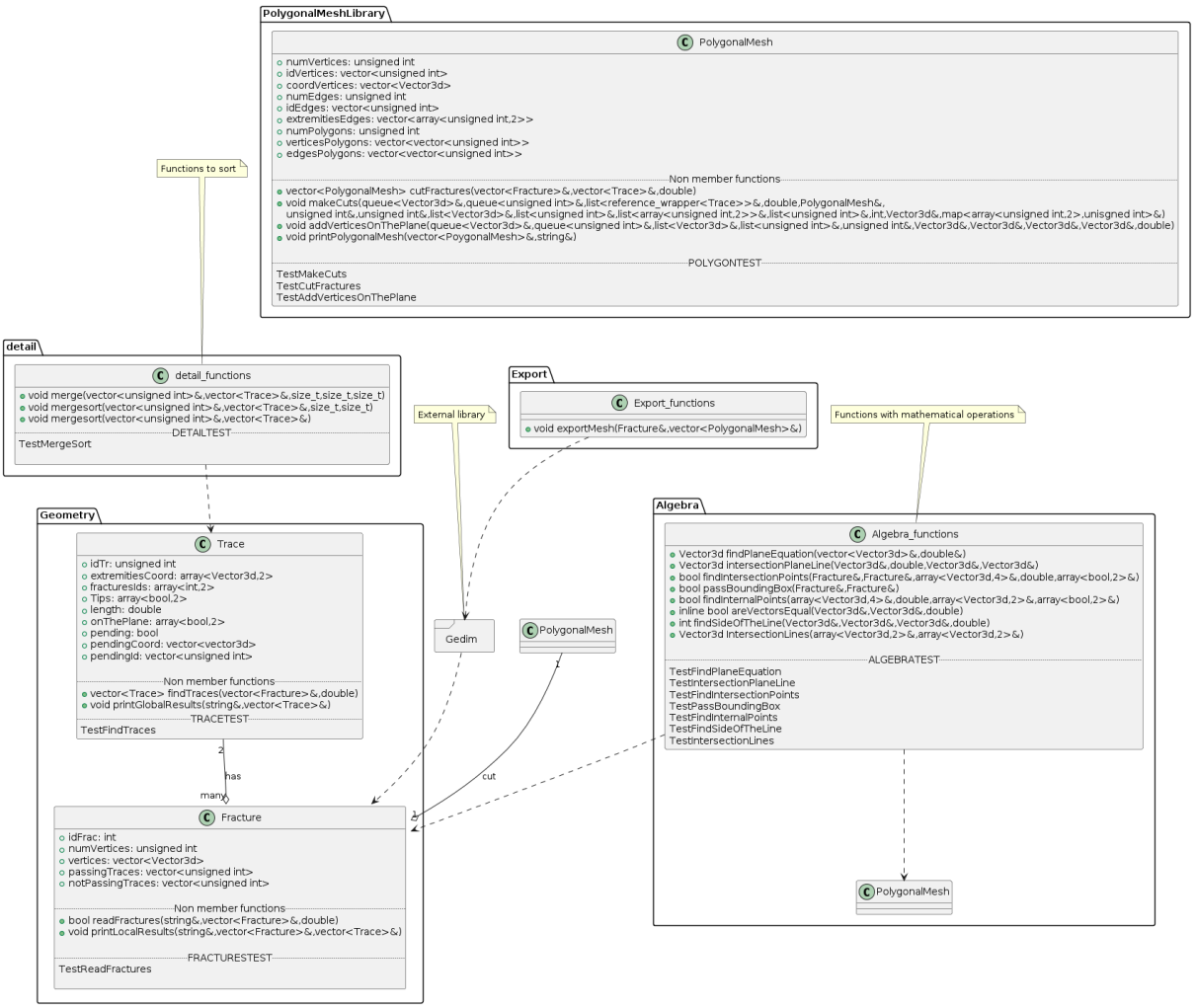



Figura 2.2: Documentazione