# Programmazione e Calcolo Scientifico

Corso di Laurea
in Matematica per l'Ingegneria, Politecnico di Torino

Progetto di gruppo

Gaia Catalano, Federico Di Leo, Greta Di Vincenzo


Anno Accademico 2023-2024

# Indice

# Capitolo 1

# Prima parte

In questo capitolo ci occuperemo di spiegare le funzioni e le scelte che hanno caratterizzato la ricerca delle tracce del DFN. In particolare, come abbiamo trovato le tracce per ogni frattura, separando quelle non passanti da quelle passanti, ordinandole in seguito in base alla loro lunghezza.

## 1.1 Strutture dati scelte e importazione DFN

### 1.1.1 Struttura DFN

Abbiamo scelto di rappresentare il Discrete Fracture Network per mezzo della struct *DFN*, struttura versatile che ci permette di conservare i dati in modo personalizzato, rendendo il codice più leggibile e chiaro all'utente. A loro volta, anche le fratture e le tracce, che corrispondono ai poligoni planari e alle loro intersezioni, sono organizzate in struct, *Fracture* e *Trace*, che contengono i dati relativi ai due oggetti.

Più nel dettaglio, la struct *DFN* è caratterizzata da un intero che indica il numero di fratture, un vettore di fratture ed uno di tracce.
La struct *Fracture* è invece caratterizzata da un identificativo univoco, il numero di vertici della frattura ed le loro coordinate, salvate in una matrice, i vettori contenenti le coordinate del baricentro, della normale, il piano che contiene la frattura, identificato dai suoi quattro coefficienti rispetto agli assi coordinati, e i vettori degli identificativi delle tracce passanti e non passanti.
La struct *Trace*, infine, contiene un identificativo univoco, un array con le coordinate dei punti estremi, salvate in vettori tridimensionali, gli identificativi delle fratture coinvolte, in un array di interi, la lunghezza della traccia, la retta che contiene la traccia, memorizzata per mezzo di un suo punto e della sua direzione, e un booleano *tips*, che è impostato a True se la traccia è non passante.

### 1.1.2 La funzione *ImportFractures*

La funzione *ImportFractures* legge i files contenenti i dati sul DFN da analizzare contenuti nell'omonima cartella e inserisce ciascuna informazione nella struttura DFN di cui al punto 1.1.1. In particolare, essa prende come argomenti il percorso del file, la struttura dfn di tipo *DFN* e una tolleranza. Infatti, poiché i numeri in virgola mobile non possono essere confrontati direttamente a causa degli errori di arrotondamento e della rappresentazione finita, è necessario impostare una tolleranza per il loro confronto. La scelta della tolleranza è stata effettuata confrontando il numero 1 con il più piccolo numero in virgola mobile maggiore di 1 che può essere rappresentato in un *double*. Infatti, moltiplicando l'epsilon di macchina per 10, si ottiene una tolleranza pari a dieci volte la differenza minima rappresentabile.

Dopo aver eseguito i dovuti controlli sulla corretta lettura del file e sul suo adeguato formato, la funzione procede con la lettura riga per riga del testo.
Per prima cosa, il numero di fratture presenti viene estrapolato dalla prima riga utile e si procede con un ridimensionamento del vettore di oggetti di tipo *Fracture*, i.e. *Fractures* del *DFN*. Grazie a ciò, si può iterare lungo le righe del file senza eccedere il numero necessario di iterazioni.
Procedendo, per ogni frattura si salva l'id, il numero di vertici e si costruisce di conseguenza nella dimensione corretta la matrice che conterrà le coordinate dei vertici. In questo modo si utilizza esattamente lo spazio che è necessario.
Infine si individuano, per ogni frattura, due oggetti importanti: il baricentro e la normale, da cui si definisce naturalmente il piano su cui appoggia la frattura.

## 1.2 Individuazione e salvataggio tracce

### 1.2.1 Metodi per l'esclusione di intersezioni

Lo scopo di questa sezione di codice è quello di individuare, con un basso costo computazionale, alcune coppie di fratture di cui escludere senza dubbio l'intersezione.

In primo luogo, utilizziamo la funzione *Parallel* per verificare se due piani che contengono una frattura sono paralleli. Se lo sono, escludiamo un'intersezione utile al nostro scopo. Infatti, non consideriamo la possibilità che due poligoni siano coincidenti oppure che, solo se paralleli, condividano una porzione di area, un vertice, un lato o un qualsiasi punto.
In secondo luogo, per ciascuna frattura costruiamo la più piccola sfera che la contiene ed escludiamo l'intersezione fra due fratture le cui sfere non si intersecano. Tutto ciò è svolto dalla funzione *IntersectionSphere*, che assegna il centro della sfera al baricentro e il raggio alla massima distanza fra tale centro ed un vertice. La sfera così ottenuta contiene il poligono, perché ipotizziamo che i poligoni siano convessi.

### 1.2.2 Le funzioni *LineIntersection* e *InterFractureLine*

Le due funzioni in oggetto sono necessarie per la comprensione della funzione *FindTraces*.

La prima funzione, *LineIntersection*, prende in input due fratture e restituisce l'intersezione fra i piani che le contengono. In particolare, la direzione di tale retta d'intersezione è ottenuta come il prodotto vettoriale fra le normali ai poligoni; quindi, per definire in toto la retta, è sufficiente individuare un suo punto. Questo viene eseguito mettendo a sistema i due piani e l'equazione della retta eguagliata a 0 e risolvendo il sistema con la fattorizzazione della matrice dei coefficienti A = LU, quindi senza pivoting. La scelta di questa fattorizzazione è stata determinata dalla ricerca di una migliorare la stabilità numerica del processo di fattorizzazione. Inoltre, tale fattorizzazione è applicabile ad una vasta gamma di matrici non singolari, per cui risulta garantisce generalità. La funzione restituisce quindi la retta, individuata da punto di passaggio e direzione.

Invece, *InterFractureLine* valuta l'intersezione fra una frattura ed una retta $r$, restituendo un booleano che testimonia la presenza di almeno un punto d'intersezione ed, eventualmente, la/e ascissa/e curvilinea/e dei punti di intersezione lungo la retta $r$. Osserviamo che, data la convessità dei poligoni, tali punti possono essere al massimo 2, ed in tal caso le ascisse vengono fornite in ordine crescente.
La funzione itera sui vertici ed inizialmente separa i casi in cui il lato che va dal vertice corrente a quello successivo e la retta $r$ siano paralleli, che si verifica se il loro prodotto vettoriale è nullo. Se non sono paralleli, allora si intersecano. La posizione dell'intersezione lungo la retta contenente il lato è parametrizzata da un'ascissa $\alpha$. Se l'intersezione avviene all'interno del lato, quindi se $\alpha \in [0,1]$, allora si procede individuando un'altra ascissa, $\beta$, che rappresenta la posizione dell'intersezione lungo la retta $r$.
Se non sono paralleli, verifica se il vertice o il lato intero giacciano sulla retta.

### 1.2.3 La funzione *FindTraces*

La funzione *FindTraces* prende in input una lista di fratture, che nel corpo del main sarà quella del nostro DFN, la tolleranza precedentemente definita e la struttura *DFN* su cui salva i risultati ottenuti. Come suggerisce il nome, essa individua le intersezioni, cioè le tracce, presenti fra i vari poligoni e le salva nel DFN, corredate dei dati a loro associati.

La funzione, tramite due cicli for, si occupa di considerare ogni possibile coppia di fratture, provando inizialmente ad escludere l'intersezione a priori come al punto 1.2.1.
Se non è stato possibile escludere l'intersezione, prosegue individuando la retta di intersezione dei piani, richiamando il metodo *LineIntersection*, e le intersezioni eventuali delle due fratture con essa richiamando *InterFractureLine*.
In particolare, basta che uno dei due valori di ritorno booleani di *InterFractureLine* sia False per escludere un'intersezione; infatti, se le due fratture si intersecano, necessariamente lo fanno lungo la retta di intersezione dei piani che le contengono. Se entrambi i valori sono true, allora si verifica la presenza di un'intersezione guardando alla posizione reciproca delle ascisse dei punti di intersezione sulla retta.

In caso affermativo, si salvano gli estremi della traccia e si verifica che sia passante per una frattura o non passante, osservando se tali estremi coincidano con i punti di intersezione della frattura stessa con la retta.
Infine, si salvano gli altri attributi della frattura utilizzando i dati ottenuti fino a questo punto.

### 1.2.4 L'ordinamento delle tracce e la stampa

Come ultime funzioni di questa prima parte, abbiamo innanzitutto confrontato la lunghezza delle tracce i cui id sono passati come input tramite la funzione *compareTraceLength*. A questo punto, con la funzione *SortTracesByLength*, che prende in input il vettore con gli id delle tracce e il vettore delle tracce, abbiamo ordinato quest'ultime in ordine di lunghezza decrescente. Per farlo, abbiamo sfruttavo la funzione *sort* della libreria Standard, che ordina gli elementi di un range specifico, nel nostro caso dal primo elemento all'ultimo. Inoltre, abbiamo definito al suo interno una lambda function, che si occupa di accedere al vettore delle tracce per riferimento, quindi leggendolo e modificandolo senza farne una copia. Essa prende in input i due id delle tracce da confrontare e richiama nel suo corpo *CompareTraceLength* per dargli l'ordinamento da noi richiesto.
Inoltre, la funzione *SortTracesByLength* viene richiamata nella funzione *PrintLocalResults*, prima per l'insieme di tracce passanti, poi per quello delle tracce non passanti, che si occupa di stampare nei file di output *lresults.txt*, per ogni frattura, il numero di tracce trovate con i loro id, il valore booleano che indica se la traccia è passante o non passante e la rispettiva lunghezza.
Infine, ci siamo serviti della funzione *printGlobalResults*, che stampa sul file di output *results.txt* il numero complessivo di tracce per il *DFN*, l'id di ogni traccia e l'id delle due fratture che si intersecano, con le rispettive coordinate, per avere una panoramica complessiva sul *DFN*.

## 1.3 Test prima parte

Al fine di testare la correttezza e la coerenza dei metodi utilizzati finora abbiamo costruito in GeoGebra un insieme di fratture ad hoc, riportanti tutti i casi particolari interessanti ai fini dei test. Abbiamo poi riportato i dati di tale set sul file di testo *DFN8_test.txt*, presente nella cartella *DFN*, nel formato adatto alla lettura da parte del nostro codice.
Si puo' visualizzare tale insieme di fratture di seguito:
I test sono suddivisi in *IMPORTTEST*, *DISTANCETEST* e *TRACETEST*.

Il gruppo *IMPORTTEST* contiene solamente il test *TestImportFractures*, che verifica il corretto funzionamento della *ImportFractures*, controllando che i vertici, i baricentri, le normali delle fratture siano salvati correttamente. Inoltre, il test verifica che un file scritto in un formato scorretto non venga letto e produca un errore.

I test di *DISTANCETEST* sono quelli che verificano che le funzioni che escludono le intersezioni a priori non commettano errori. In particolare, si controlla che casi semplici

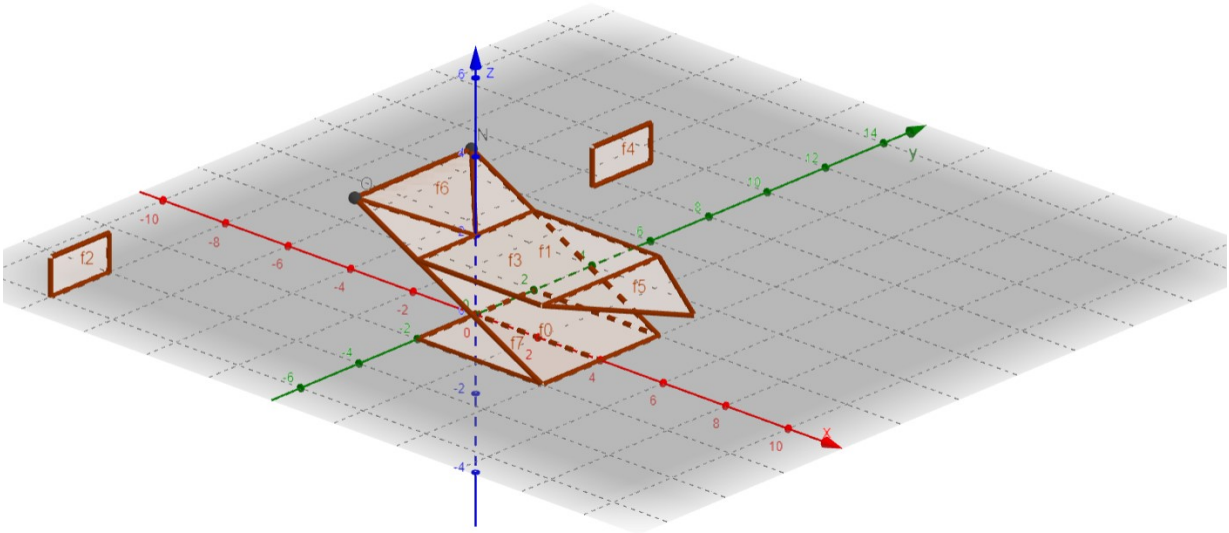


Figura 1.1. Rappresentazione in GeoGebra delle fratture utilizzate per i test

dell'esempio da noi ideati siano verificati: ad esempio, che $F_0$ ed $F_1$ siano paralleli o che $F_2$ ed $F_4$ siano molto distanti (vd. Figura 1.1).

Infine, i test raggruppati sotto *TRACETEST* testano le funzioni che compongono il processo di ricerca delle tracce. Il metodo è sempre quello di individuare casi semplici che comprendano tutte le possibili situazioni e verificare che le funzioni producano il risultato atteso. Ad esempio, per testare la funzione *FindTraces*, (test *TestFindTraces*), si verifica che $F_4$ non produca tracce, che l'intersezione fra $F_6$ ed $F_1$, in quanto puntiforme, non costituisca una traccia. Inoltre, verifica che le altre tracce siano passanti o non passanti (vd. Figura 1.1). Infine, *TestSort* verifica il corretto ordinamento delle tracce.

# Capitolo 2

# Seconda Parte

In questo secondo capitolo verranno illustrati i ragionamenti che ci hanno portato a costruire la nostra *Polygonal Mesh*.
Dopo aver trovato le tracce per ogni frattura, abbiamo salvato nella struct *PolygonalMesh* i vertici, i lati e i poligoni generati dai tagli delle fratture di partenza. Infine, abbiamo inserito tutte le mesh poligonali in un vettore della struct *Meshes*.

## 2.1 Struttura PolygonalMesh

La struct *PolygonalMesh* è caratterizzata dalle celle 0D, 1D e 2D, che rappresentano i vertici, i lati e i poligoni della mesh.

Per ogni tipo di cella abbiamo memorizzato il numero di celle presenti nella mesh.
Per ogni cella 0D, corrispondente ai vertici generati tagliato le fratture di partenza, abbiamo memorizzato l'id della cella in un vettore e abbiamo salvato le coordinate dei vertici.
Per le celle 1D, oltre a memorizzare l'identificativo di ogni lato, abbiamo salvato gli identificati dei vertici che caratterizzano il lato, un vettore di booleani che ci permette di capire quali solo i lati attivi della nostra mesh, un vettore di id corrispondenti ai poligoni adiacenti ad ogni lato e un vettori di id contente gli id di ogni lato nuovo aggiunto a seguito del taglio.
Inoltre, per le celle 2D abbiamo memorizzato un vettore di interi rappresentati gli id dei poligoni, un vettore contenente gli id dei lati e uno gli id dei vertici che caratterizzano il poligono, per ogni poligono. Infine, abbiamo salvato in un vettore di booleani l'informazione sui poligoni finali, permette di capire quali solo i poligoni attivi della nostra mesh.

Per ultimo, nella struct *Meshes* abbiamo un vettore di Polygonal Mesh in cui salviamo tutte le mesh generate.

### 2.1.1 La funzione *CreateMesh*

La funzione *CreateMesh* si occupa di generare, per ogni frattura del DFN, una mesh poligonale. Innanzitutto, raggruppa le tracce della frattura, prima quelle passanti e poi quelle non passanti; in seguito, crea la mesh poligonale di partenza, inizializzandola con tutti i suoi oggetti. A questo punto viene chiamata la funzione *CutFracture* che andrà a tagliare le fratture iniziali, basandosi sul lavoro delle funzioni *CutAndSave* e *PositionVert*.

## 2.2 Taglio delle fratture

### 2.2.1 La funzione *CutAndSave*

La funzion *CutAndSave* esegue il taglio di un poligono, il cui id viene passato alla funzione come input. Inoltre, il metodo prende in input anche la mesh su cui si sta lavorando, la retta passante per la traccia, gli indici dei vertici in cui iniziano i lati coinvolti nel taglio, gli id dei punti d'intersezione e una tolleranza.
Per ogni lato del poligono, viene calcolata la direzione del lato e il prodotto vettoriale con la direzione della retta per cui passa la traccia. Se il prodotto vettoriale non è nullo, confrontando con una tolleranza quadratica, vengono calcolati i parametri $\alpha$ e $\beta$, dal significato analogo ai parametri trovati in *InterFractureLine*; in questo modo si trovano le intersezioni della traccia con la frattura, salvando contestualmente i lati interessati, che poi verranno spenti, quindi non saranno più considerati, e quelli che abbiamo chiamato vertici d'aiuto, ovvero i vertici in cui iniziano i lati su cui ricade la traccia.

A questo punto, abbiamo tenuto in considerazioni quattro casi possibili. Se gli estremi della traccia non coincidono con nessun vertice del poligono, allora salviamo le informazioni sui quattro nuovi lati e i due nuovi vertici generati dal taglio della frattura; se, invece, un estremo della frattura coincide con un vertice, solo un lato verrà tagliato, generando altre 2 celle 1D e troverà una sola cella 0D da aggiungere; infine, se entrambi gli estremi della traccia coincidono con i vertici, aggiorno soltanto le celle 0D inserendo i vertici del poligono corrispondenti. In tutti e quattro i casi, inserisco intanto i nuovi lati nella posizione corretta e salvo i poligoni adiacenti ad ogni lato; infine, inserisco il nuovo lato corrispondente alla traccia che taglia il poligono, inserendo anche per quest'ultimo i poligoni adiacenti.
Tutte queste informazioni vengono salvate nella mesh su cui si sta lavorando, garantendone la consistenza.

### 2.2.2 La funzione *PositionVert*

Abbiamo utilizzato la funzione *PositionVert* all'interno di *CutFracture* per dividere le tracce che tagliano le celle 2D. La funzione in questione determina la posizione di un punto rispetto ad una retta del piano, utilizzando il prodotto scalare e vettoriale per determinare, entro una certa tolleranza, se il punto sta sopra la retta, sotto o esattamente sulla retta.
Passando come input il punto estremo della traccia, la retta che contiene la traccia e

la tolleranza, *PositionVert* trova il vettore che va da un punto della retta all'estremo, calcolando il prodotto vettoriale trova un vettore perpendicolare al piano, infine con il prodotto scalare tra il vettore perpendicolare e quello unitario lungo l'asse z, misura la componente z del vettore ricavato dal prodotto vettoriale.
Se la componente z è maggiore di 0, entro una certa tolleranza quadratica, allora il punto è sopra la retta, viceversa sta sotto.

### 2.2.3 La funzione *CutFracture*

La funzione *CutFracture*, che richiama al suo interno entrambi i metodi appena citati, si occupa di creare le celle 2D originate dal taglio.
Il primo controllo che facciamo è nel verificare che vi siano delle tracce. Infatti, richiamando il metodo in modo ricorsivo, man mano ci occupiamo di eliminare la traccia che ha effettuato il taglio, finché non ci troviamo a non avere più tracce e a poter salvare il poligono finale. A questo punto, se vi sono ancora tracce nel poligono che stiamo considerando, lo disattiviamo perché esso sarà tagliato e aggiornato con due nuovi poligoni generati dal suo taglio.
Successivamente, fornendo alla funzione *CutAndSave* i dati di cui ha bisogno, ricaviamo gli id dei vertici e li inseriamo in due vettori distinti per le sottofratture che si generano. In questo modo, con l'aiuto di una variabile booleana che cambia in base alla posizione del vertice in esame, dividiamo i vertici appartenenti al primo poligoni da quelli appartenenti al secondo poligono attivo. In maniera analoga, ci occupiamo di salvare le celle 1D. Durante questi processi, si va sempre a verificare che il lato che si sta aggiungendo non sia già stato aggiunto precedentemente.
Infine, dividiamo le tracce rimanenti, sfruttando *PositionVert* per capire a che sottopoligono appartengono.
A questo punto, il metodo *CutFracture* viene richiamato per ogni sottoinsieme di poligoni in modo ricorsivo, fino ad esaurimento dei tagli.

## 2.3 Correzione dell'oggetto *PolygonalMesh*

A questo punto, all'interno della funzione *CreateMesh*, dopo l'esecuzione dei tagli, viene chiamata la funzione *CorrectMesh*, che permette di correggere la mesh, mantenendo la sua consistenza. Come ultimo step, inseriamo la mesh corretta nel vettore di mesh poligonali, contenente tutte le mesh generate.

### 2.3.1 La funzione *CorrectMesh*

La seguente funzione si occupa di contare il numero delle celle 0D, 1D e 2D da inserire nella mesh. In questo modo la mesh viene corretta, inserendo, oltre alle celle 0D che non hanno bisogno di controlli, le celle 1D e quelle 2D che effettivamente caratterizzano la nostra mesh poligonale finale.

### 2.3.2 La funzione *PrintMeshes*

Quest'ultima funzione, che prende in input l'oggetto di tipo *Meshes*, quindi il vettore di mesh poligonali, è stata implementata per verificare la consistenza delle mesh generate e la loro correttezza. Infatti, stampando nel file di output *meshes.txt* tutte le ifnromazioni relative alle mesh create abbiamo potuto verificare l'esattezza del codice.