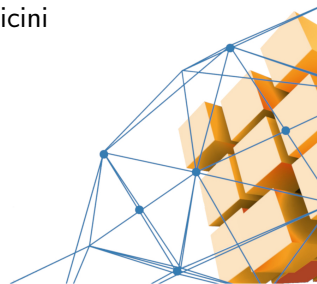
Politecnico di Torino
Dipartimento di Scienze Matematiche "G. L. Lagrange"
DISMA ECCELLENZA 2018•2022

Tracce e Fratture

Candidati:
Alessio Massenzana
Francesco Licitra
Alexis Laurient

Docenti:
Prof. S. Berrone
Prof. M. Ciccuttin
Prof. F. Vicini

Programmazione e Calcolo Scientifico



Parte 1: determinazione delle tracce

Il cuore della prima parte del progetto è definito dalle funzioni:

1. checkIntersection prende in input due fratture e cerca di capire se si può escludere a priori la possibilità che ci sia intersezione
2. ElaborateV ha il compito di trovare la traccia e definire per quali fratture è passante e non-passante

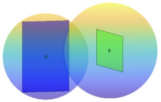
Queste funzioni lavorano in sinergia nella funzione computeDFN()

```
void DFNLibrary::DFN::computeDFN(){
    for(int i = 0; i < numberFractures; i++){
        for(int j = i+1; j < numberFractures; j++){
            vector<Eigen::Vector3d> v;
            if(checkIntersection(fractures[i], fractures[j], v)){
                elaborateV(fractures[i], fractures[j], v);
            }
        }
    }
}
```

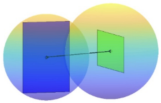
CheckIntersection

Inizialmente va ad effettuare dei controlli preliminari, per ottimizzare e per escludere a priori delle intersezioni in casi particolari.
Check 1:

(a) Fratture nello spazio



(b) Creazione sfere contenenti le fratture



(c) Calcolo la distanza tra i centri

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.

Check 2: basato sul fatto che i punti nello spazio possono soddisfare o meno l'equazione di un piano. Dato un punto (x_0, y_0, z_0) e dato un piano

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

$$\text{Se } (x_0, y_0, z_0) \notin \pi \Rightarrow \begin{cases} ax_0 + by_0 + cz_0 + d > 0 & \text{il pt si trova da una parte del piano} \\ ax_0 + by_0 + cz_0 + d < 0 & \text{il pt sta dall'altra parte del piano} \end{cases}$$

Il segno dipende dall'orientazione del piano e dalle coordinate del punto rispetto al piano. Per questo motivo eseguo due controlli:

- consideriamo il piano π_1, contenente la frattura F_1,e sostituiamo all'interno della sua equazione i vertici della frattura F_2.
- consideriamo il piano π_2, contenente la frattura F_2,e sostituiamo all'interno della sua equazione i vertici della frattura F_1.

Se almeno uno di questi restituiscono tutti lo stesso segno (o tutti positivi o tutti negativi) escludiamo l'intersezione $\Rightarrow$ non si generano tracce.

Calcolo della retta d'intersezione

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

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:

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

risoluzione del sistema

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

con la decomposizione QR implementata all'interno della libreria Eigen.

Calcolo punti d'intersezione

Vado a scontare la retta r con tutti i lati di F_1 e di F_2.
i lati delle fratture sono del tipo:

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

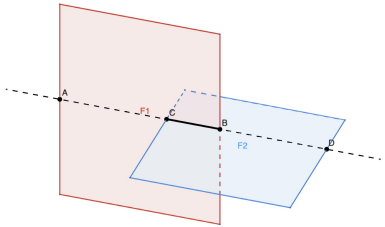
la retta r può essere vista come

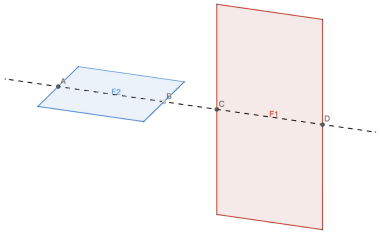
$$r = \left\{ \mathrm{x} \in \mathbb{R}^3 : \mathrm{x} = \mathrm{P} + v_1 \mathrm{t},\ v_1 \in \mathbb{R} \right\}$$

Risolviamo il sistema

$$\begin{cases} Q = \mathrm{B} + (\mathrm{A} - \mathrm{B})s \\ Q = \mathrm{P} + \mathrm{v}_1 t \end{cases}$$

e trovo s e v_1.
Controllo, meno della tolleranza, sul valore di s: se $Q \in AB$ allora $s \in [0 - \tau, 1 + \tau]$.





Reiterando questo processo per ogni lato di ciascuna frattura, F_1 ed F_2, alla fine otteniamo 4 punti di intersezione :

$$A, B \in F_1 \cap r \quad \text{e} \quad C, D \in F_2 \cap r$$

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 F_1 con quelli di F_2, quella che avevamo trovato non era effettivamente un'intersezione.

ElaborateV

Trovati i quattro punti di interesse, ha il compito di trovare la traccia e definire per quali fratture è passante e per quali è non-passante. Per adempiere a questo compito Possiamo analizzare tre casistiche possibili:

1. $A \equiv C \wedge B \equiv D \Rightarrow$ la traccia è passante per entrambe le fratture (Figura 1).
2. $A \equiv C \vee B \equiv D \Rightarrow$ per una frattura la traccia sarà passante, mentre per l'altra no (Figura 2).
3. Il caso di zero coppie comprende due diverse situazioni:
   - Una frattura attraversa l'altra: $A < C < D < B$
     La traccia è passante per F2 (Figura 3)
   - Traccia semplice: $A < C < B < D$
     Traccia non passante per entrambe le fratture (Figura 4)

Definizione della traccia: casistiche possibili

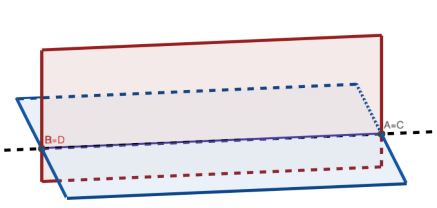


Figure: 1. Traccia passante per entrambi

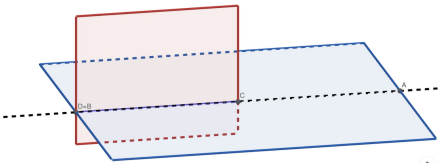


Figure: 2. Traccia passante per F1

Definizione della traccia: casistiche possibili

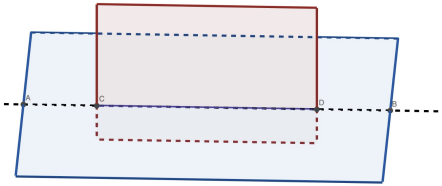


Figure: 3. Traccia passante per F2

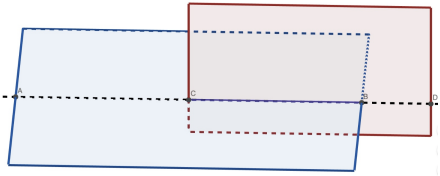


Figure: 4. Non passante per entrambe

Sort e Compare Trace

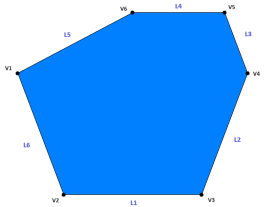
Per completare l'ordinamento abbiamo utilizzato la sort della libreria std

$$sort((f.tracce).begin(), (f.tracce).end(), compareTrace);$$

```
bool DFNLibrary::compareTrace(std::pair<Traccia*, bool>& T1, std::pair<Traccia*, bool>& T2){
    if(T1.second != T2.second){
        if(T1.second) return true;
        else return false;
    }

    return (T1.first->length() > T2.first->length());
}
```

Calcolo della mesh poligonale

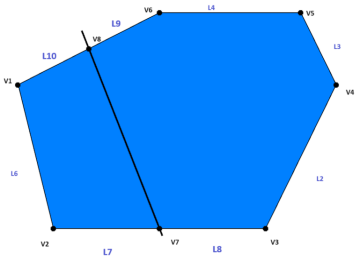


void cutPolygon (
. *list⟨pair⟨Vector3d, Vector3d⟩⟩&listaTagli*,
. *PolygonalMesh&mesh*, *unsigned int idP*)

Genera due sottopoligoni e ci richiama sopra se stessa.

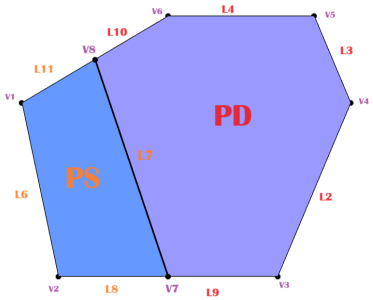
Variabili di supporto:
Vector<unsigned int> PDVert, PSVert;
Vector<unsigned int> PDEdges, PSEdges;
list<pair<Vector3d,Vector3d> > listaTagliSx, listaTagliDx;
bool inizioFine[2], lati[0];

Calcolo della mesh poligonali



Aggiungo alla mesh i due nuovi punti e i cinque nuovi lati. Ciclando il poligono in senso antiorario aggiungiamo i lati su PS, finchè non troviamo il lato da tagliare (lati[0]), quindi taglio;
Ora aggiungo su PD finchè non raggiungo l'altro lato da tagliare $\Rightarrow$ Taglio $\Rightarrow$ Ritorno su PS

Ridistribuzione delle tracce



Prelevo una traccia T da listaTagli;
Se T interseca L7 aggiungo T ad entrambe le liste.
Altrimenti, calcolo il piano perpendicolare a P, contenente il nuovo lato (L7).
Come nel Check 2 della checkIntersection, verifico il segno, che l'equazione del piano assume sostituendo un vertice di PS. Se l'origine di T ha lo stesso segno allora aggiungo a listaTagliSx.
In caso contrario aggiungo a listaTagliDx.

Aggiornamento & Ricorsione

Al termine del taglio verrà riaggiornata la mesh, sovrascrivendo mesh.Cell2DVertices[idPD] con PDVert, mesh.Cell2DEdges[idPD] con PDEdges, e facendo push_back() degli oggetti del nuovo poligono PS.

Infine, se ci sono ancora tagli da elaborare, viene richiamata la cutPolygon sui due sottopoligoni innescando la ricorsione.

```
if(!listaTagliDx.empty())
    cutPolygon(listaTagliDx, mesh, idP);

if(!listaTagliSx.empty())
    cutPolygon(listaTagliSx, mesh, idS);
```

Grazie per l'attenzione

Politecnico di Torino