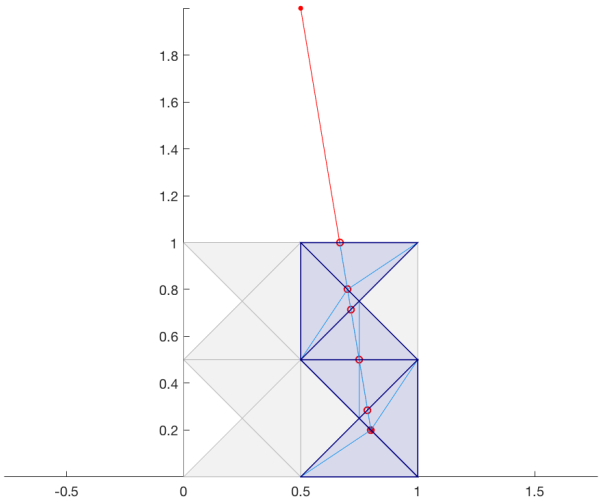
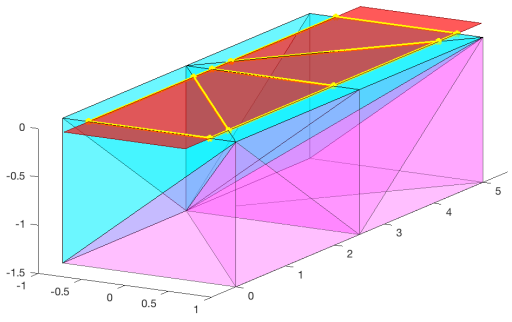
Programmazione e calcolo scientifico
Anno Accademico 2017/2018

Bessi Ludovico - Boscolo Cegion Claudio
Cataldo Luca - Licari Valentina



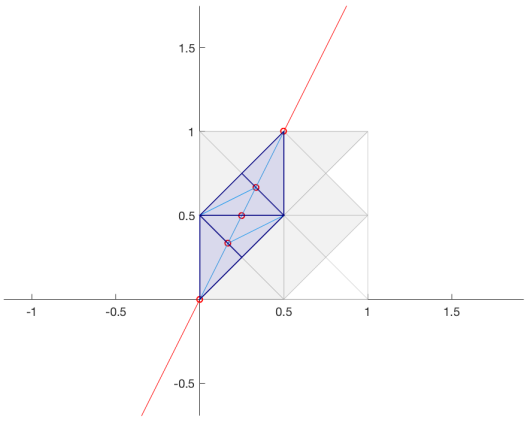


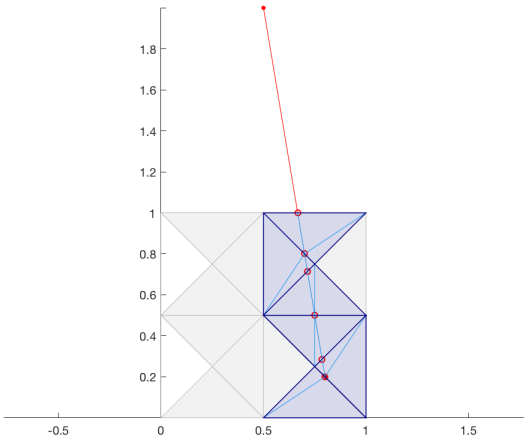
Indice

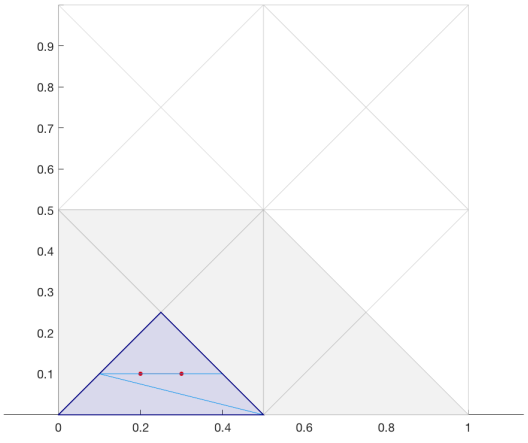
1 Progetto 2D **4**
- 1.1 Struttura del programma . . . 4
- 1.2 Lettura file e strutture dati . . . 4
- 1.3 Script di riempimento e di utilitá . . . 7
- 1.4 Le funzioni which_side e intersect . . . 7
- 1.5 Ricerca del primo triangolo tagliato . . . 9
  - 1.5.1 somma = 1 o -1 . . . 9
  - 1.5.2 somma = 0 o 4 . . . 10
  - 1.5.3 somma = 2 . . . 11
  - 1.5.4 somma = 3 o 5 . . . 12
- 1.6 Script di triangolazione . . . 13
  - 1.6.1 it_is_cut . . . 13
  - 1.6.2 it_is_near . . . 13
  - 1.6.3 enqueue_tri_to_check . . . 13
- 1.7 Ricerca di tutti gli altri triangoli tagliati . . . 13

2 Progetto 3D **16**
- 2.1 Struttura del programma . . . 16
- 2.2 Lettura file e strutture dati . . . 16
- 2.3 Script di riempimento e di utilitá . . . 20
  - 2.3.1 Computing fract plane . . . 20
  - 2.3.2 tets_for_node . . . 20
  - 2.3.3 edges_for_node . . . 20
  - 2.3.4 faces_for_tet . . . 20
  - 2.3.5 edges_for_tet . . . 20
  - 2.3.6 global_toll . . . 20
- 2.4 which_edge . . . 21
- 2.5 which_side_2D . . . 21
- 2.6 which_side_3D . . . 21
- 2.7 intersect_plane_edge . . . 21
- 2.8 Intersect_2D . . . 22
- 2.9 Intersect_2D_bis . . . 22
- 2.10 Intersect_3D . . . 23
- 2.11 enqueue_tet_to_check . . . 27
  - 2.11.1 it_is_near . . . 27
- 2.12 Ricerca primo tetraedro tagliato . . . 27
  - 2.12.1 somma = 2 o -2 . . . 28
  - 2.12.2 somma = 0 . . . 29
  - 2.12.3 somma = 3 o 5 . . . 30
  - 2.12.4 somma = 6 o 10 . . . 32
  - 2.12.5 somma = 8 . . . 33
  - 2.12.6 somma = 11 o 13 . . . 34

2.13 Script di tetraedrizzazione e poliedrizzazione 35
2.13.1 Slicing_tet 35
2.14 filling_queue_temp 36
2.15 Ricerca di tutti gli altri tetraedri tagliati 36


1.5
1
0.5
-0.5
-1
-0.5
0.5
1
1.5


1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
-0.5
0
0.5
1
1.5


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.2
0.4
0.6
0.8
1

1 Progetto 2D

Obiettivo della parte bidimensionale del progetto é, data una triangolazione e un segmento definito "traccia", portare a termine le seguenti richieste:

- Individuare quali triangoli di partenza vengono tagliati dalla traccia
- Individuare tutti i triangoli vicini ai triangoli tagliati, i.e. tutti quelli che condividono un vertice o un lato con un triangolo tagliato
- Costruire sottopoligoni a partire dalle informazioni ottenute analizzando la posizione della traccia rispetto alla triangolazione di partenza.
- Costruire sottotriangoli a partire dalle informazioni ottenute analizzando la posizione della traccia rispetto alla triangolazione di partenza.
- Salvare le informazioni relative alle intersezioni tra la traccia e la triangolazione, i.e. le ascisse curvilinee dei punti di intersezione.

1.1 Struttura del programma

Il nucleo del programma é diviso in due parti: la ricerca del primo triangolo tagliato dalla traccia e la ricerca di tutti gli altri.
La prima parte della ricerca consiste nel controllare i triangoli ad uno ad uno: trovato un triangolo tagliato sappiamo che tra i suoi vicini ci saranno altri triangoli tagliati. Di conseguenza tutti i vicini di tale triangolo vengono inseriti in una coda (vettore di strutture *queue*), quindi in tale coda ci possono essere o triangoli tagliati oppure triangoli vicini a triangoli tagliati. Successivamente, vengono controllati i triangoli partendo dalla testa della coda, eventualmente aggiungengo i relativi vicini.
Se il triangolo in testa alla coda non é tagliato, esso viene catalogato come triangolo vicino.

1.2 Lettura file e strutture dati

In questa sezione vengono brevemente presentate le scelte adottate per salvare le informazioni del file relativo alla triangolazione. Per generare le strutture dati di *node* e *info_trace* é stato utilizzato il comando *repmat* di matlab.

Node
Si è scelto di salvare nella struttura *node* le coordinate dei punti della triangolazione del piano cartesiano 2D, un "side" del punto rispetto alla traccia (un intero che puó assumere valore 0, 1, -1, 2) che esprime dove si trova il

punto rispetto alla traccia. Il campo side assume valore '0' se la posizione del punto non é nota, '1' o '-1' se il punto si trova a sinistra o a destra della traccia e '2' se si trova sulla retta della traccia. Il campo "toll" presenta una tolleranza relativa al problema, calcolata tenendo conto delle dimensioni dei lati della triangolazione e verrá meglio descritta in seguito. 'Edges' e 'Triangles' sono vettori che contengono rispettivamente gli indici dei segmenti e dei triangoli che hanno il nodo della struttura dati come estremo. 'tot_edges' e 'tot_triangles' contengono rispettivamente il numero di segmenti e il numero di triangoli condivisi da ciascun nodo. L'ultimo campo di *node* contiene l'ascissa curvilinea s del punto nel caso in cui questo si trovi sulla traccia.

Triangle
É una matrice di n_triangles righe e 10 colonne. Le colonne 1, 2, 3 contengono i vertici del triangolo (indici dei nodi). Le colonne 4, 5, 6 contengono degli status relativi ai lati dell' i-esimo triangolo. Gli status indicano le possibili posizioni del segmento rispetto alla traccia (interno, secante, non incidente). É importante osservare che la colonna 4 della matrice contiene lo status del segmento formato dai nodi 2-3; la colonna 5 della matrice contiene lo status del segmento formato dai nodi 1-3; la colonna 6 della matrice contiene lo status del segmento formato dai nodi 1-2. Le colonne 7, 8, 9 contengono le eventuali ascisse curvilinee dei segmenti del triangolo con la traccia, seguendo una numerazione analoga a quella poco prima descritta. La colonna 10, contiene un flag che monitora l'operazione di messa in coda del triangolo. Il flag puó assumere i seguenti valori:

- -1 Lo status é sconosciuto
- -2 Il triangolo in coda ha due nodi condivisi
- -3 Il triangolo in coda ha un nodo condiviso
- -4 Il triangolo é oppure é stato nella coda secondaria
- 0 Il triangolo é tagliato.
- > 0 Il triangolo é vicino e il flag indica la posizione in *near_tri*.

Edge
É una matrice contenente i segmenti indicizzati della triangolazione: le due colonne della matrice contengono gli indici degli estremi del segmento.

Neigh
É una matrice le cui colonne dell'i-esima riga contengono i vicini dell'i-esimo triangolo

Trace_vertex
É una matrice le cui colonne contengono le *coordinate* degli estremi della traccia.

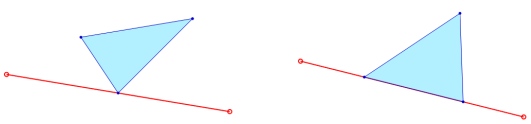
Trace
É una matrice le cui colonne contengono gli indici degli estremi della traccia riferiti alla matrice 'trace_vertex'.

Info_trace
É una struttura che contiene, per ogni traccia, le informazioni relative alle richieste del problema. Questa struttura é divisa in sottostrutture.

- **cut_tri** é una struttura che contiene l'indice dei triangoli tagliati. Le coordinate dei vertici del triangolo e i punti di intersezione con la traccia sono inseriti nel campo *points*. Nel campo "poly_1" e "poly_2" sono presenti i punti, riferiti a *points*, che costituiscono la poligonalizzazione. Analogamente, il campo "tri" contiene i punti della sottotriangolazione.
- **s** é un vettore contenente le ascisse curvilinee delle intersezioni con la traccia (riferite alla parametrizzazione della traccia).
- **near_tri** é una struttura che contiene le informazioni relative ai triangoli vicini, in particolare "id" é l'indice del triangolo, "nodes" e "edge" contengono rispettivamente i nodi e i lati condivisi con un triangolo tagliato.

Queue
Queue é un vettore di strutture con tre campi: id, points, edges. *Points* contiene i punti in comune a un triangolo tagliato, *edges* i lati.
In queue vengono messi i triangoli vicini a triangoli tagliati che devono essere controllati.

Queue temp
É una coda secondaria in cui vengono inseriti triangoli vicini a triangoli non tagliati, per i quali peró la traccia ricopre interamente un lato o passa per un vertice senza tagliare il triangolo (vedi figura). Molto probabilmente avranno come vicini dei triangoli tagliati, ed é questo il motivo per il quale vengono salvati.



1.3 Script di riempimento e di utilitá

In questa sezione vengono brevemente presentati alcuni script che calcolano informazioni utili per portare a termine le richieste del problema. I dati calcolati da triangles_for_node, edges_for_node e toll_for_node vengono salvati globalmente nelle strutture presentate in precedenza.

- **triangles_for_node**
  Calcola per ogni vertice i triangoli aventi quel nodo come estremo. La strategia é quella di scorrere i triangoli uno a uno e salvare in ognuno dei tre vertici l'id del triangolo in esame.

- **edges_for_node**
  Analoga allo script "triangles_for_node": calcola le stesse informazioni per i segmenti e inserisce gli indici degli edges nel campo 'edges' di node. Il numero totale dei segmenti condivisi é salvato in 'tot_edges'.

- **toll_for_node**
  L'idea utilizzata per avere una tolleranza con cui lavorare é quella di fare una media sulle lunghezze dei segmenti della triangolazione. Per ogni nodo é quindi stata calcolata una media delle lunghezze di tutti i segmenti aventi quel nodo come estremo. Indicati con $\mathbf{v_i}$ l'i-esimo edge e con n il numero totale di edge che condividono tale nodo, la tolleranza é calcolata su:
  $$\sum_{i=1}^{n} \frac{\|\mathbf{v_i}\|_\infty}{n}$$
  Il numero ottenuto verrá moltiplicato per l'*accuracy*. Al fine di *diminuire il costo computazionale* viene utilizzata, per il calcolo delle lunghezze, la norma infinito invece della norma euclidea.

- **which_edge**
  É una function a cui vengono passati due punti. L'output é l'edge che unisce i due punti. Viene utilizzata in *enqueue_tri_to_check*.

1.4 Le funzioni which_side e intersect

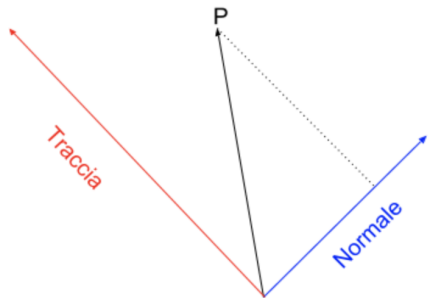
In questa sezione vengono descritte due funzioni importanti che servono per capire come i segmenti di un triangolo sono posizionati rispetto alla traccia, ossia se questi la intersecano e come. Grazie a queste informazioni sará possibile capire quali triangoli sono tagliati o meno.

- **which_side**
  É una function a cui vengono passati l'id della traccia e il punto del quale si vuole conoscere la posizione rispetto ad essa. La retta passante per i due nodi della traccia divide il piano in due semipiani:

which_side restituiscce 1 o -1 a seconda del semipiano a cui il punto appartiene. Se invece giace sulla retta restituisce 2 e la coordinata curvilinea s del punto. Questa coordinata viena calcolata in questo modo:

$$s = \frac{\mathbf{T} \cdot \mathbf{v}}{\|\mathbf{T}\|_2^2}$$

Dove **T** é il vettore della traccia. Definiamo con T1 T2 gli estremi della traccia e con p il punto da analizzare. Troviamo il vettore **N** perpendicolare a T2 - T1 e il vettore **v** = P - T1. Il segno del prodotto scalare $\mathbf{N} \cdot \mathbf{v}$ indica in quale semipiano si trova p.



- **intersect**
  É una funzione alla quale viene passato l'id della traccia in esame e i due punti p_1 e p_2 che formano il segmento del quale si vuole conoscere l'intersezione con essa. Restituisce lo status del segmento e l'ascissa curvilinea dell'intersezione riferita alla parametrizzazione della traccia. La logica secondo cui lavora la function é la seguente: prima viene chiamata "which_side" per ciascuno dei due punti passati: si ottengono i side dei rispettivi punti e in base al loro prodotto si imposta l'algoritmo. Se per esempio il prodotto é uguale a 1 é impossibile che vi sia intersezione tra il segmento e la traccia (poiché entrambi i side dei punti sono o 1 o -1). Se il prodotto é -1, l'intersezione é ottenuta risolvendo il seguente sistema lineare

$$\begin{bmatrix} x_{p_2} - x_{p_1} & -(x_B - x_A) \\ y_{p_2} - y_{p_1} & -(y_B - y_A) \end{bmatrix} \begin{bmatrix} t \\ s \end{bmatrix} = \begin{bmatrix} x_A - x_{p_1} \\ y_A - y_{p_1} \end{bmatrix}$$

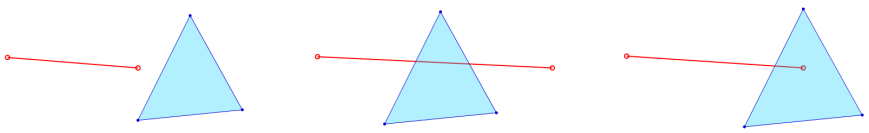
  Dove p_1 e p_2 sono i punti passati a "intersect" e i punti A e B sono i punti iniziale e finale della traccia. Il sistema é stato risolto utilizzando l'algoritmo di riduzione gaussiana implementato nativamente in Matlab. Se il prodotto dei side in partenza é uguale a 4 vengono fatte delle considerazioni per capire se segmento e traccia sono staccati, uno interno all'altro o si intersecano parzialmente. La funzione

intersect valuta per gli altri casi la s restituita dal sistema e assegna uno status all'edge preso in esame a seconda che questo non intersechi la traccia, la intersechi a croce, muoia dentro la traccia o sia tutto interno alla traccia. Questo status viene salvato nelle colonne 4, 5 e 6 della matrice triangle durante la ricerca dei triangoli tagliati.

1.5 Ricerca del primo triangolo tagliato

Lo script "finding_first_tri" ricerca il primo triangolo tagliato e mette in coda tutti i vicini di tale triangolo, in modo che la ricerca cessi di procedere casualmente. La ricerca prende in esame una riga alla volta della matrice triangle. Per ciascun punto del triangolo in esame chiama la funzione *"which_side"* a meno che non sia giá stata chiamata in precedenza.
Called_ which_side é un vettore di flag che serve per capire se la funzione *which_side* é stata chiamata nell'iterazione corrente oppure se il side del nodo é giá noto poiché calcolato analizzando altri triangoli. Nel primo caso, se il nodo si trova sulla traccia, la sua coordinata curvilinea viene inserita in info_trace e salvata in *node.s*.
Lo script ignora un triangolo quando i suoi tre vertici hanno lo stesso "side". In caso contrario l'algoritmo calcola la somma dei side e in base a questa segue procedimenti diversi.

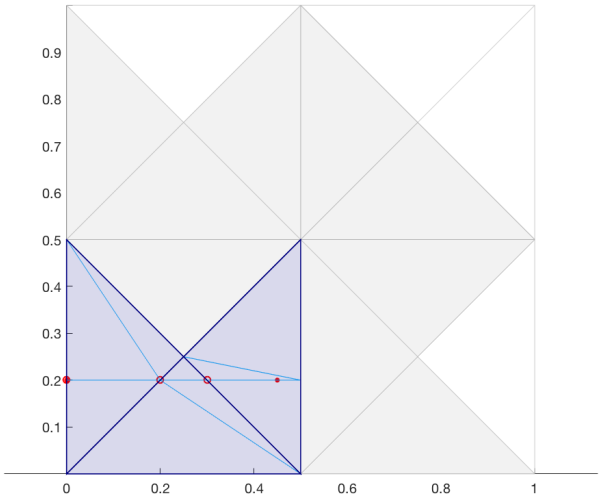
1.5.1 somma = 1 o -1



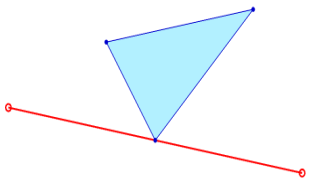
Significa che due nodi stanno in un semipiano individuato dalla traccia e il punto restante si trova nell'altro semipiano. In questo caso l'algoritmo trova i punti che stanno nello stesso semipiano (*points_together*) e quello nel semipiano opposto (*lonely_point*). Si valutano in seguito gli status dei "lati interessanti". I "lati interessanti" sono quelli formati dal nodo solitario e ciascuno dei due nodi nel semipiano opposto. Si salvano gli status di questi edge nella matrice triangle. Dopo aver considerato un edge e avergli assegnato uno status, quest'ultimo si salva anche per tutti i triangoli che condividono lo stesso edge, attraverso lo script *saving_in_neigh*.

L'algoritmo valuta a questo punto gli status degli edge e ogni volta che trova un triangolo tagliato chiama lo script *it_is_cut*. Inoltre viene chiamata la function *enqueue_tri_to_check* per mettere in coda i triangoli vicini a quello considerato. Tramite un controllo sulle ascisse curvilinee, si verifica se

la traccia é tutta interna al triangolo: in questo caso il triangolo é tagliato. Trovato il triangolo tagliato, l'algoritmo inserisce le ascisse curvilinee dei punti di intersezione in *info_trace*, trova le coordinate cartesiane dei punti sostituendo le ascisse curvilinee nella parametrizzazione della traccia e prosegue con la triangolazione e la poligonalizzazione.
Il triangolo tagliato puó essere sottotriangolato in due modi distinti: o in due triangoli o in tre triangoli: in quest'ultimo caso, se necessario, la traccia viene prolungata fino a toccare l'edge del triangolo. Per quanto riguarda la poligonalizzazione, nel primo caso vi é un unico poligono che coincide con il triangolo stesso, nel secondo caso ci sono due poligoni.



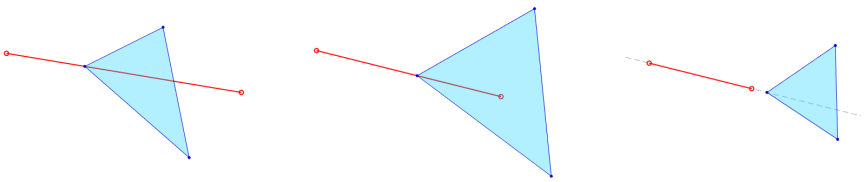
1.5.2 somma = 0 o 4



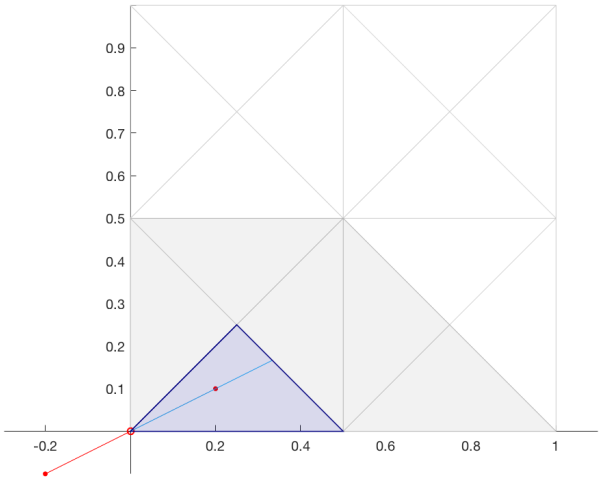
Significa che un nodo sta sulla traccia e gli altri si trovano nella stessa parte del semipiano. Trovato il nodo che si trova sulla traccia si analizza la sua ascissa curvilinea per capire la sua posizione: potrebbe infatti trovarsi sulla

retta della traccia, ma non sul segmento. In questo caso il triangolo non é tagliato, tuttavia molto probabilmente i triangoli vicini al triangolo considerato sono tagliati: vengono quindi aggiunti nella coda *queue_temp*.

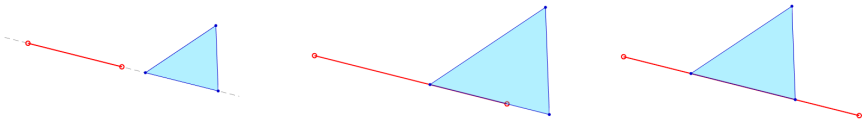
1.5.3 somma = 2



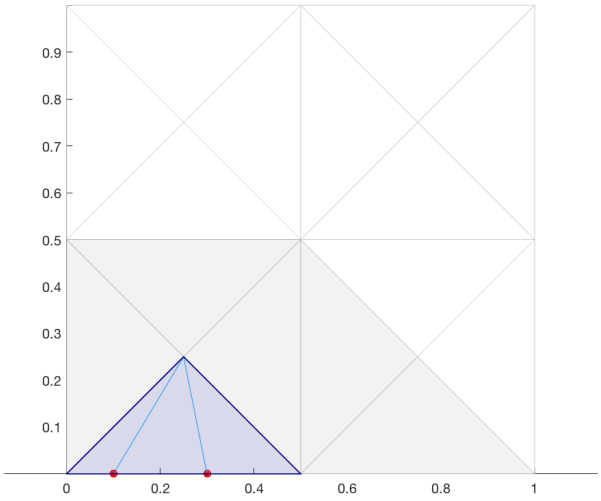
Significa che un nodo sta sulla traccia e gli altri due si trovano in due semipiani opposti. L'algoritmo trova a questo punto il nodo che sta sulla traccia (*node_on_trace*) e i due nodi "opposti" (*opposite_nodes*).
In seguito controlla il segmento formato dai nodi opposti chiamando la funzione intersect (nel caso in cui non sia giá stata chiamata). Dopo aver salvato lo status e la *s* dell'intersezione anche per i triangoli vicini, l'algoritmo controlla la *s* del nodo *node_on_trace* e la *s* dell'edge che taglia la retta della traccia. Se il triangolo risulta tagliato, viene chiamato lo script it_is_cut e viene trovata l'intersezione con la traccia e eventualmente il suo prolungamento.
Il triangolo viene infine sottotriangolato e sottopoligonalizzato. I triangoli della sottotriangolazione sono due e la sottopoligonalizzazione coincide con la sottotriangolazione.



1.5.4 somma = 3 o 5



Significa che la traccia é coincidente e parallela al segmento. Dopo aver salvato quale dei tre nodi non é contenuto nella traccia su *lonely_point*, viene chiamata la funzione intersect per l'edge formato dai due nodi sulla traccia. Vengono analizzate (e inserite in info_trace) le ascisse curvilinee e poi si controlla se il triangolo é tagliato oppure no. Se la traccia tutta interna al segmento,la sottotriangolazione consiste in tre triangoli unendo il vertice opposto (lonely_point) ai due estremi della traccia.
Il triangolo, quando é tagliato, se la traccia é parzialmente interna al lato, il triangolo é suddiviso in due triangoli, unendo l'estremo della traccia contenuto nel segmento con il vertice opposto.



1.6 Script di triangolazione

1.6.1 it_is_cut

É uno script che viene chiamato quando l'algoritmo trova un triangolo tagliato. Lo script mette in *info_trace*, nel campo *cut_tri*, l'id del triangolo tagliato e nel sottocampo *points* di *cut_tri* inserisce le coordinate dei punti del triangolo considerato. Queste serviranno per la triangolazione e la poligonalizzazione.

1.6.2 it_is_near

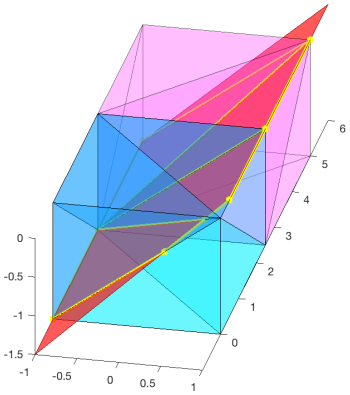
É un breve script che inserisce in *near_tri* l'id di un triangolo vicino a un triangolo tagliato. Se é la prima volta che lo si incontra, viene salvato nell'ultima cella di *near_tri*. Se é giá stato salvato, si aggiornano le relative informazioni, aggiungendo eventualmente nuovi nodi e/o lati in comune. Inoltre, viene salvata nella colonna 10 di *triangle* la posizione in cui é salvato in *near_tri*.

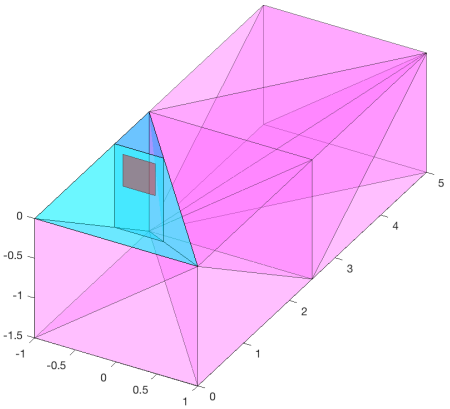
1.6.3 enqueue_tri_to_check

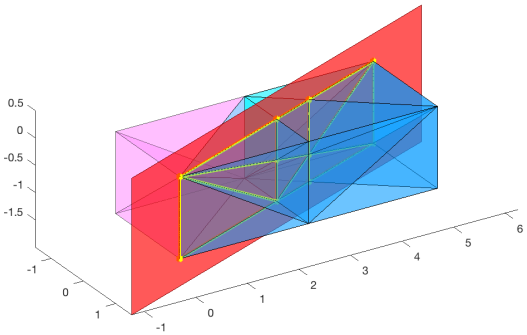
Funzione che valuta i triangoli da inserire nella coda vicini al triangolo passato come parametro. Viene chiamata quando un triangolo é tagliato. Prima di inserirli controlla che non siano giá tagliati osservando il loro status. Oltre a inserire l'id del triangolo inserisce anche nodi e edge condivisi con il triangolo tagliato.

1.7 Ricerca di tutti gli altri triangoli tagliati

Dopo aver trovato il primo triangolo tagliato, la ricerca continua finché *queue* o *queue_temp* contengono triangoli da analizzare. Prima di tutto viene chiamato lo script *checking_queue_temp* che svuota la coda secondaria e ottimizza la ricerca dei triangoli tagliati, andando a considerare i triangoli potenzialmente tagliati. Viene poi considerata la coda primaria, costituita dai triangoli vicini ai tagliati. Prima di procedere in modo analogo a *finding_first_tri*, si controlla che l'id del triangolo messo in queue non sia minore del numero di triangoli giá analizzati nella ricerca del primo triangolo. In tal caso *it_is_near* verifica se il triangolo é giá stato salvato nei vicini di un tagliato o meno. Se il triangolo é ancora da analizzare viene chiamata la funzione *which_side* per ciascuno dei nodi del triangolo per i quali il side é ancora sconosciuto. Se il side dei tre nodi risulta lo stesso, il triangolo viene messo in *near_tri* (é vicino a un tagliato in quanto si trova in *queue*, ma non é lui stesso tagliato per via delle considerazioni sui side). L'algoritmo procede considerando le somme dei side e adottando le stesse scelte della ricerca del primo triangolo tagliato.


0
-0.5
-1
-1.5
-1
-0.5
0
0.5
1
0
1
2
3
4
5
6


0
-0.5
-1
-1.5
-1
-0.5
0
0.5
1
0
1
2
3
4
5


0.5
0
-0.5
-1
-1.5
-1
0
1
-1
0
1
2
3
4
5
6

2 Progetto 3D

Obiettivo della parte tridimensionale del progetto scritto in Matlab é, data una frattura e una tetraedrizzazione di una porzione di spazio, portare a termine le seguenti richieste:

- Individuare quali tetraedri di partenza vengono tagliati dalla frattura
- Individuare l'elenco dei tetraedri che condividono almeno un vertice con un tetraedro tagliato e memorizzzare per questi anche quali vertici o lati sono condivisi con un tetraedro tagliato
- Sottopoliedrizzare ciascun tetraedro utilizzando i punti di intersezione tra tetraedro e frattura. Nel caso in cui la frattura terminasse all'interno del tetraedro la specifica richiede che questa sia prolungata fino ad una faccia o ad un vertice del tetraedro
- Suddividere la frattura in poligoni ottenuti dall'intersezione con i tetraedri

2.1 Struttura del programma

Per ogni frattura il programa si divide in due blocchi principali: la ricerca del primo tetraedro é a tappeto. Individuato il primo tetraedro tagliato, la ricerca cessa di essere casuale. Questo accade quando viene individuato un tetraedro che tocca la frattura. A questo punto tutti i tetraedri vicini al tetraedro tagliato vengono aggiunti in una coda (*queue*). La coda viene svuotata partendo dalla testa. Se un triangolo in coda non é tagliato, viene automaticamente classificato come vicino.
Per capire se un tetraedro é tagliato si trovano le sue intersezioni con il piano della frattura, quindi si calcola la sua "impronta". Studiando la posizione reciproca tra impronta e frattura é possibile dedurre se il tetraedro é tagliato oppure no. Per tale analisi, impronta e frattura sono proiettate sul piano Oxy, Oxz oppure Oyz, in base a quello che comporta una minor distorsione delle figure.

2.2 Lettura file e strutture dati

Per la lettura del file sono state adottate strategie analoghe al caso bidimensionale.

Node
É una struttura creata con il comando *repmat* che contiene le coordinate dei nodi della tetraedrizzazione nel campo *coord*, un "side" che indica in quale semispazio il nodo si trova rispetto al piano della frattura (side = 4 nel caso in cui giaccia sul piano). Il campo *edges* di contiene gli id dei segmenti

aventi quel nodo come estremo e, analogamente, il campo tet contiene gli id dei tetraedri aventi il quel nodo come vertice. I campi tot_edges e *tot_tets* tengono il conto di quanti segmenti e di quanti tetraedri il nodo fa parte. Il *where_on_plane* viene utilizzato nel caso in cui il punto giaccia sul piano della frattura: esso indica la posizione del nodo nel vettore di strutture *node_on_plane*.

Tet
É una struttura che ha 4 campi: il campo *P* contiene gli indici dei punti che costituiscono il tetraedro, *faces* contiene gli indici delle facce del tetraedro e *status_queue* contiene lo status del tetraedro. *status_queue* assume i seguenti valori:

- **0** indica che il tetraedro é tagliato
- **-1** indica che non si hanno ancora informazioni sul tetraedro
- **-2, -3, -4** indica che il tetraedro inserito nella coda ha rispettivamente 1, 2 o 3 nodi in comune con un tetraedro tagliato
- $>$**0** il tetraedro é stato analizzato e il numero in *status_queue* indica la sua posizione in *info_trace.near_tet*

Il campo *edges* contiene gli indici dei segmenti che formano il tetraedro.

Edge
É una struttura che ha come campi *P*, checked e tets. *P* contiene l'indice dei punti che compongono il segmento. *Checked* é un flag e assume valori:

- **-1** se non é ancora nota la posizione dell'edge rispetto al piano della frattura
- **0** l'edge non interseca il piano della frattura
- $>$**0** nel caso in cui l'edge sia stato giá considerato e intersechi il piano della frattura. L'intersezione con il piano é salvata su un vettore di strutture chiamato *node_plane*. Lo status indica la posizione in cui tale punto é stato salvato su *node_plane*

Neigh
Contiene gli indici dei vicini dei tetraedri.

Face
Contiene gli indici delle facce dei tetraedri. Nella quarta e quinta colonna sono salvati i tetraedri che condividono l'i-esima faccia.

Fract_vertex

Contiene le coordinate dei vertici della frattura.

Fract
É una struttura che ha come campi: *P, n_points, N, d, G, r, side_int,protocol*. *P* contiene gli indici dei punti della frattura, *n_points* il numero di punti della frattura. *N* é il vettore normale al piano della frattura, *d* é il termine noto dell'equazione del piano. *G* é il baricentro della frattura, *r* é la distanza massima in norma infinito tra il baricentro e i vertici della frattura; *side_int* serve come riferimento per capire se un punto é interno o no alla frattura e assume valori 1 o -1 (utilizzato in intersect_3D).
Il campo *protocol* viene usato per trattare casi particolari nella funzione *intersect_3D*.

Info_fract
É una struttura che serve per salvare tutte le informazioni richieste. Ha come campi principali: *cut_tet, near_tet,pol e points*.

- **cut_tet** É una struttura che contiene l'indice del tetraedro tagliato nel campo *id*. Nei campi *points* e faces sono contenuti rispettivamente i punti e le facce necessarie per descrivere la poliedrizzazione. Nei campi *poly_1* e *poly_2* sono descritti i due sottopoliedri, facendo riferimento a faces.

- **near_tet** É una struttura che contiene l'indice del tetraedro vicino nel campo *id*. Nei campi *points, edges faces* sono contenuti rispettivamente i punti, i lati e le facce condivise con un tetraedro tagliato.

- **points** Contiene i punti necessari alla poligonalizzazione della frattura.

- **pol** Contiene nel campo *v* il vettore degli indici dei vertici per ogni poligono.

Node_plane
É una struttura che viene popolata mano a mano che si trovano nuove intersezioni tra gli edges dei tetraedri e il *piano* della frattura. Ha 7 campi:

- **coord** contiene le coordinate dei punti sul piano giá proiettate in due dimensioni

- **third_coord** contiene la terza coordinata nello spazio.

- **sides** é un vettore che contiene tutti i side di un punto rispetto a ogni lato della frattura.

- **is_out** é un flag che vale 1 se il punto é esterno alla frattura, 0 altrimenti.
- **near_nodes** é una matrice $n \times 2$ contenente nella prima colonna gli indici dei nodi vicini e nella seconda colonna -1 nel caso l'edge formato dal nodo in esame e dal nodo vicino non tocchi la frattura. Contiene un numero > 0 nel caso in cui le informazioni si trovino su *info_node*: in tal caso il numero indica la posizione nella quale le informazioni sono state su salvate su *info_node*.
- **from_edge** é un flag che vale 1 se il punto é intersezione tra un edge e il piano della frattura, vale 0 se é un nodo della tetraedrizzazione.

Ogni volta che nel codice viene inserito un nuovo punto in node_plane, vengono inizializzati i campi is_out a -1, from_edge (a 1 o 0 a seconda che il punto derivi dall'intersezione di un segmento col piano della frattura o dal nodo di un tetraedro) e in_info a -1.

Info_node
É un archivio di informazioni su coppie di punti vicini sul piano. Tali informazioni riguardano la posizione delle coppie rispetto alla fratura e verranno descritte meglio in intersect_3D.

Queue
É una coda in cui vengono inseriti i tetraedri da analizzare, ossia i vicini ad un tetraedro tagliato.
L'indice del tetraedro é salvato nel campo *id*. Nei campi *points, edges e faces* sono salvati rispettivamente i punti, lati e facce condivisi con un tetraedro con un tetraedro tagliato.

Queue_temp
Vengono inseriti in *queue_temp* i tetraedri che condividono un edge che giace completamente interno alla frattura. Tale edge é condiviso con un tetraedro avente una faccia completamente interna alla frattura. Come nel caso 2D, una coda secondaria che contiene tetraedri non tagliati ma che molto probabilmente hanno dei vicini che lo sono.
É una matrice con due colonne: la prima contiene l'indice del tetraedro, la seconda l'indice dell'edge condiviso.

2.3 Script di riempimento e di utilitá

2.3.1 Computing fract plane

É uno script che serve a calcolare il piano della frattura. Si calcola e si salva in *fract.N* il vettore normale al piano della frattura normalizzato (le sue coordinate corrispondono ai coefficienti di x, y e z nell'equazione del piano) e si individua la componente massima per ottenere una proiezione ottimale. Si calcola inoltre il termine noto dell'equazione del piano.
Successivamente si calcola il baricentro e la distanza massima tra baricentro e vertici, informazioni che serviranno poi in *intersect_3D*. Per capire se un punto é interno alla frattura si sfrutta il baricentro come punto campione (il baricentro é sempre interno). In particolare si calcola il *side_int* della frattura. Maggiori dettagli descritti in *intersect_3D*.
Infine si memorizzano i vertici della frattura in *info_fract.points*.

2.3.2 tets_for_node

Calcola per ogni nodo i tetraedri aventi quel nodo come estremo. La strategia é quella di scorrere i tetraedri uno a uno e salvare in ognuno dei vertici l'id del tetraedro in esame.

2.3.3 edges_for_node

Analogo allo script "tets_for_node". Calcola le stesse informazioni per i segmenti e inserisce gli indici degli edges al campo "edges" di node. Il numero totale di segmenti condivisi é salvato in "tot_edges".

2.3.4 faces_for_tet

Salva le facce nel campo faces di tet.

2.3.5 edges_for_tet

Calcola per ogni tetraedro gli edges che lo costituiscono.

2.3.6 global_toll

Si vuole ottenere una tollerenza globale, calcolata con la media aritmetica delle lunghezze dei segmenti in norma 1 moltiplicata per un valore predefinito di accuratteza pari a 10^{-14}. Il calcolo é il seguente:

$$\sum_{i=1}^{n} \frac{\|\mathbf{v_i}\|_1}{n}$$

Dove v_i é l'i-esimo edge, e n é il numero totale di edge.

2.4 which_edge

Dati gli indici di due punti, which_edge é una funzione che calcola l'indice dell'edge che li unisce.

2.5 which_side_2D

É una funzione a cui vengono passati gli estremi A, B di un segmento e un punto P del quale si vuole conoscere la posizione rispetto al segmento. Essa restituisce -1, 1 o 2, come nel caso 2D.

2.6 which_side_3D

É una function alla quale vengono passate l'id della frattura e il punto P del quale si vuole conoscere il side rispetto al piano della stessa. La funzione restituisce -1 o 1 a seconda che il punto si trovi da una parte o dall'altra rispetto al piano e 4 se il punto giace su di esso. Ricordando che $\mathbf{N}$ é il vettore normale alla traccia e definendo con $\mathbf{v}$ il vettore che congiunge P e il primo punto della frattura si ha che il prodotto scalare $\mathbf{N} \cdot \mathbf{v}$ é uguale a 0 se il punto é sul piano. Altrimenti il side del punto é 1 o -1 in base al segno del prodotto scalare.

2.7 intersect_plane_edge

É una funzione che riceve in ingresso l'id della frattura e l'id dell'edge del quale si vuole calcolare l'intersezione e restituisce il punto di intersezione con il piano della frattura, da memorizzare su node_plane. Restituisce inoltre una terza coordinata, ovvero quella eliminata dalla proiezione in due dimensioni. Il piano é scritto in forma cartesiana $ax + by + cz + d = 0$. I coefficienti a, b, c sono le tre componenti di $\mathbf{N}$ e la d che viene salvata dallo script *computing_fract_plane* in *fract.d*. Data la parametrizzazione del segmento formato da P_1 e P_2:

$$\begin{aligned} x &= P_{1x} + (P_{2x} - P_{1x}) \cdot s \\ y &= P_{1y} + (P_{2y} - P_{1y}) \cdot s \\ z &= P_{1z} + (P_{2z} - P_{1z}) \cdot s \end{aligned}$$

per calcolare l'intersezione, si risolve l'equazione:

$$s = -\frac{a \cdot P_{1x} + b \cdot P_{1y} + c \cdot P_{1z} + d}{a \cdot (P_{2x} - P_{1z}) + b \cdot (P_{2y} - P_{1y}) + c \cdot (P_{2z} - P_{1z})}$$

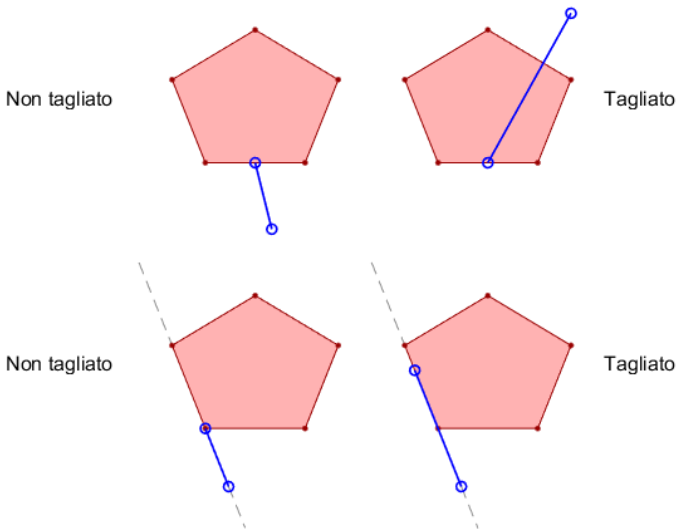
dove s é l'ascissa curvilinea della retta parametrizzata.

2.8 Intersect_2D

La function *intersect_2D* prende come input gli indici di due punti salvati su node_plane e l'id della frattura. Restituisce una matrice 2×2 contente le intersezioni tra segmento e frattura, il numero di intersezioni proprie (ossia non comprendenti vertici della frattura) e due interi che indicano l'indice del lato della frattura in cui il segmento eventualmente entra e quello in cui eventualmente esce.
Per ogni lato della frattura, cerca se ci sono delle intersezioni con il segmento (controllo analogo tra traccia e segmento del progetto 2D). Se i punti stanno in semipiani differenti é necessario fare un sistema, controllando che l'intersezione non avvenga proprio nell'estremo. Se segmento e retta sono paralleli, é necessario trattare tre casi distinti: nessuna intersezione, intersezione in un estremo o sovrapposizione dei due segmenti.
Se l'uscita avviene attraverso il vertice n della frattura, allora l'indice di uscita é n-1; se invece avviene l'entrata attraverso il vertice n, l'indice di entrata é n stesso. Nel caso in cui gli estremi del segmento siano entrambi esterni e ci sia solo un'intersezione con un vertice n della frattura, l'indice di entrata e di uscita é n.

2.9 Intersect_2D_bis

É stato necessario creare una questa funzione per gestire un caso molto particolare di *intersect_3D*: quando si hanno solo due punti nell'impronta con uno esterno e uno sulla frontiera della frattura.
Si presentano quattro alternative:



2.10 Intersect_3D

É una function alla quale vengono passati l'id della frattura, il vettore *id_node_plane* e il vettore third_cord. Il vettore *id_node_plane* contiene le posizioni dei punti in esame che giacciono sul piano della frattura. Tali posizioni si riferiscono alla struttura *node_plane*: é infatti tale struttura a contenere tutte le informazioni sui punti che popolano il piano.
Questa funzione é per certi aspetti la piú importante poiché ha in primo luogo l'obiettivo di stabilire se un tetraedro é effettivamente tagliato o meno, in secondo luogo aggiunge l'eventuale poligono che si forma nell'intersezione. La funzione restituisce 0 se il tetraedro sicuramente non é tagliato, 1 se il tetraedro é sicuramente tagliato e restituisce 2 nel caso in cui l'impronta sia completamente interna alla frattura.
Procede attraverso controlli sempre piú raffinati, articolati in tre step.

- **Step 1**

Nello step 1, si calcola il "baricentro" e distanza massima dell'impronta del tetraedro, facendo la media aritmetica tra le coordinate dei vertici. Il baricentro della frattura e la sua distanza massima sono giá stati calcolati precendentemente.
Se la distanza tra i due baricentri é maggiore della somma delle due distanze massime, allora sicuramente i due poligoni non si intersecano. Nonostante questo sia un controllo piuttosto grossolano, il suo costo computazionale é quasi nullo.
Se la condizione sulle distanze non dovesse essere soddisfatta e se l'impronta ha almeno tre vertici(il caso con due vertici viene trattato a parte), si passa allo step 2.

- **Step 2**

Qui si controlla se l'impronta sia tutta interna alla frattura oppure no, considerando un vertice dell'impronta alla volta.
A tale scopo, sono calcolati i side relativi a ciascun lato della frattura. Se tutti i side sono uguali al side campione (side calcolato per un punto sicuramente interno, per esempio il baricentro), o eventualmente uguali a 2 nel caso in cui il vertice stia sulla frontiera, allora tale vertice é interno. Tutte queste informazioni vengono salvate globalmente in un campo di *node_plane*. Nel caso in cui un vertice sia interno, esso viene salvato su *info_fract.points*.
Se tutti i vertici sono interni, la funzione restituisce 2. Se almeno un punto dell'impronta é esterno alla frattura, si passa allo step 3.

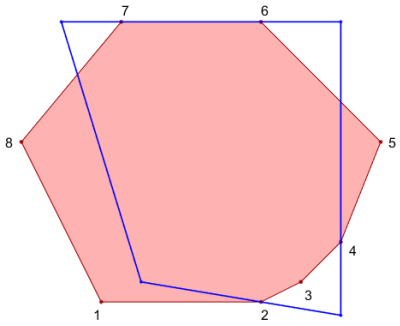
■ **Step 3**

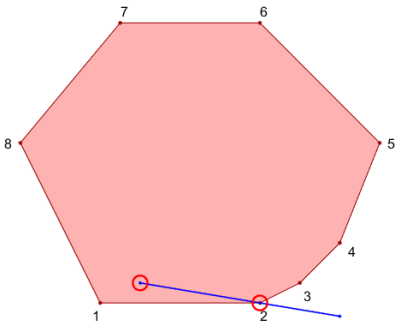
Si analizza un lato dell'impronta alla volta facendo una sorta di "girotondo", aggiungendo ad un vettore (*pol_temp*) i vertici del poligono che verrá a formarsi. Per ogni lato viene chiamata la funzione *intersect_2D*. Affinché l'algoritmo funzioni, il verso del girotondo deve essere uguale al verso di percorrenza della frattura. Nel caso in cui il verso dell'impronta e il verso della frattura non fossero coerenti, é necessario scambiare l'ordine dei punti dell'impronta. Capiamo il verso facendo il prodotto vettoriale tra il primo lato e il secondo lato dell'impronta e successivamente della frattura. Se il verso del prodotto é concorde allora la percorrenza é la stessa, altrimenti é necessario lo scambio.
Nel percorrere il girotondo possono avvenire delle transizioni, ossia un lato puó entrare e/o uscire dalla frattura. In ogni iterazione, é salvato in memoria il lato della frattura da cui é avvenuta l'ultima uscita e quello da cui é avvenuta l'ultima entrata.
Per ogni lato dell'impronta se il primo estremo non é esterno alla frattura, viene aggiunto a *pol_temp*. Se gli estremi del lato non sono entrambi interni, viene chiamata la funzione *intersect_2D*, la quale restituisce dove il lato é eventualmente entrato o uscito. Vengono quindi aggiunti a *pol_temp* i vertici della frattura compresi tra l'uscita e l'entrata (entrata esclusa, uscita inclusa).
Vengono infine aggiunte le intersezioni tra il lato e frattura, che sono restituite da *intersect_2D*. Si noti che se il lato interseca la frattura in un vertice, questo non viene restituito dalla funzione *intersect_2D* per evitare di inserirlo due volte in *pol_temp*. Se *pol_temp* contiene almeno tre punti, allora il tetraedro é sicuramente tagliato e il poligono viene salvato globalmente in *info_fract*.
Se *pol_temp* contiene due elementi o meno, allora é importante distinguere il caso in cui la frattura sia completamente interna oppure esterna all'impronta. Ci sono casi in cui si sa che la frattura non puó essere completamente interna, per questo introduciamo il campo *protocol* di fract. In questo caso, ossia quando si trova un'impronta completamente interna alla frattura, oppure quando il *pol_temp* ha almeno tre elementi, il protocollo é uguale a 0.
Nel caso in cui non si hanno ancora informazioni, il protocollo é posto uguale a -1.
Se il protocollo vale -1, allora si controlla se il baricentro della frattura é interno all'impronta. In tal caso, *protocol* vale 1 e il tetraedro é considerato tagliato. Altrimenti, si distinguono i casi in base alla lunghezza di *pol_temp*.
Se *pol_temp* ha due elementi, allora un lato dell'impronta é parzialmente o totalmente interno ad un lato della frattura. Per distinguerli, si controlla quanti e quali sono i vertici dell'impronta interni alla frattura.
Nel primo caso il tetraedro é sicuramente tagliato. Nel secondo caso, é ta-

gliato a meno che i due vertici interni non siano entrambi dei nodi effettivi del tetraedro (campo *from_edge* = 0 per entrambi).
Se *pol_temp* non contiene elementi, allora o frattura e impronta non si toccano affatto, oppure un vertice della frattura é interno ad un lato dell'impronta.
Per distinguerli si controlla l'ultimo output della funzione *intersect_2D*.
Se *pol_temp* ha un solo elemento, un vertice dell'impronta giace sul bordo della frattura. Il tetraedro é tagliato solo se tale punto non é un nodo del tetraedro.
Se l'impronta ha **due vertici** si controlla se sono entrambi interni. In tal caso il tetraedro non é tagliato.
Se almeno uno é esterno, si controlla se l'altro é propriamente interno, ossia quando non giace sulla frontiera. In questo caso il tetraedro é tagliato. Nel caso in cui entrambi i punti siano esterni viene chiamata la funzione *intersect_2D* e osservando il suo output si puó capire se l'impronta interseca o no la frattura.
Se invece un vertice dell'impronta giace sul bordo viene chiamata la funzione *intersect_2D_bis*.

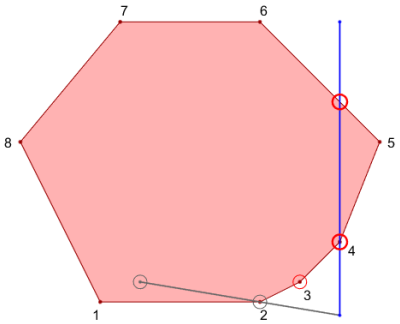
▪ Esempio

Trattiamo il caso mostrato in figura un lato alla volta. La figura é la frattura e il poligono blu é l'impronta del tetraedro.

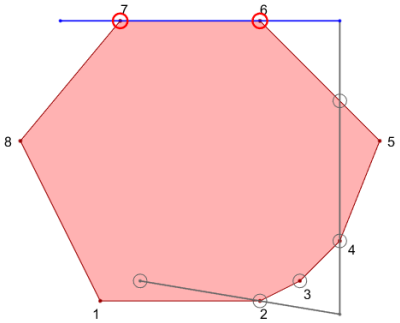




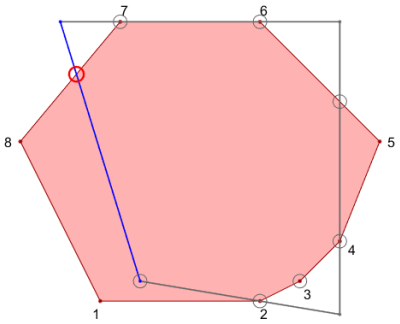
Il primo estremo del primo lato é interno alla frattura, quindi viene inserito in *pol_temp*. Dato che il segmento interseca la frattura in un vertice, la funzione *intersect_2D* non restituisce punti di intersezione. Restituisce 1 come indice di uscita e 0 come indice di entrata.



Il primo estremo del secondo lato é esterno alla frattura, quindi non viene inserito in *pol_temp*. La funzione *intersect_2D* restituisce solo un punto di intersezione (il secondo), perché il primo coincide con un vertice della frattura. L'indice di entrata é 4. L'indice di uscita é 5. Dato che precedentemente é avvenuta un'uscita, in *pol_temp* sono inseriti i vertici della frattura dall'1 al 4 (1 escluso). É inserito infine il punto di intersezione.

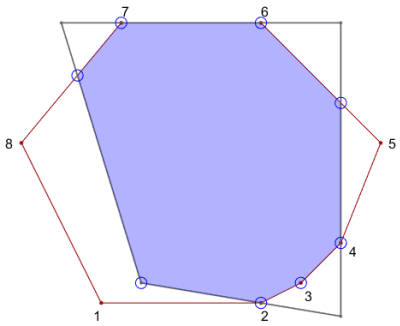


Il primo estremo del terzo lato non viene inserito in *pol_temp*. In questo caso, considera l'entrata attraverso il vertice 6 della frattura(indice di entrata = 6) e l'uscita attraverso il vertice 7 (indice di uscita = 6), quindi non restituisce punti di intersezione. Precedentemente é avvenuta un'uscita dal lato 5, quindi vengono inseriti tutti i punti dal 5 escluso al 6 incluso, ossia solo il vertice 6.



Il primo estremo del quarto lato non viene inserito in *pol_temp*. La funzione *intersect_2D* restituisce 7 come indice di entrata, 0 come indice di uscita e un punto di intersezione. Precedentemente, é avvenuta un'uscita dal lato 6, quindi il vertice 7 della frattura é inserito. Infine viene inserita l'intersezione tra segmento e frattura.

Il poligono finale é rappresentato dalla seguente figura:



2.11 enqueue_tet_to_check

Enqueue_tet_to_check é uno script che mette nella coda principale i tetraedri vicini a tetraedri tagliati. Bisogna fare attenzione: potrebbero essere giá stati messi in coda. Si controlla che non sia questo il caso guardando la flag *status_queue*. Si inizia aggiungendo i vicini di faccia. Successivamente si aggiungono i vicini di edge facendo attenzione che non siano anche vicini di faccia. Infine, si aggiungono i vicini di vertice, facendo attenzione che non siano né vicini di faccia né vicini di edge.

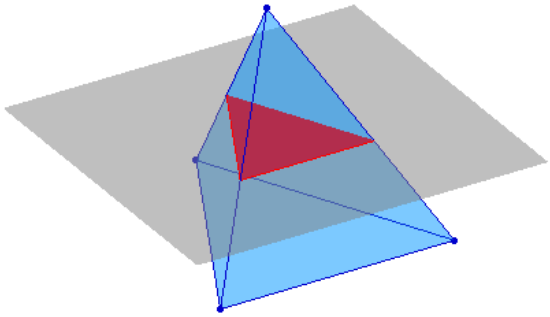
2.11.1 it_is_near

É uno script che viene chiamato quando il tetraedro in esame é vicino di un tetraedro tagliato. Viene aggiunto alla fine di *near_tet* se non era giá stato salvato. In caso contrario, si aggiungono eventuali nuovi punti, lati o facce in comune.

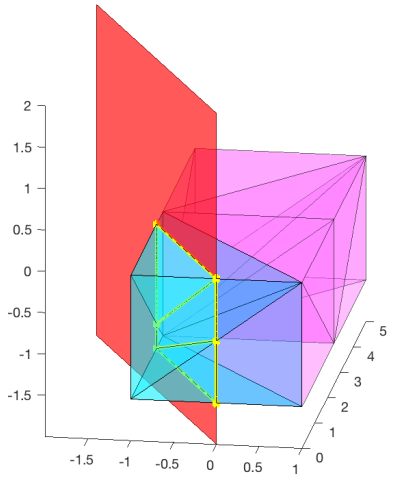
2.12 Ricerca primo tetraedro tagliato

Nello script *finding_first_tet* si comincia la ricerca del primo tetraedro tagliato. Si procede fino a quando il flag found resta uguale a 0 oppure si esauriscono i tetraedri della tetraedrizzazione. Per ciascuno dei 4 nodi del tetraedro viene chiamata la function *which_side_3D* nel caso in cui il side del nodo rispetto al piano della frattura non sia conosciuto. Nel caso in cui i 4 punti non si trovino nello stesso semispazio, il tetraedro potrebbe essere tagliato e l'algoritmo prosegue considerando la somma dei side.

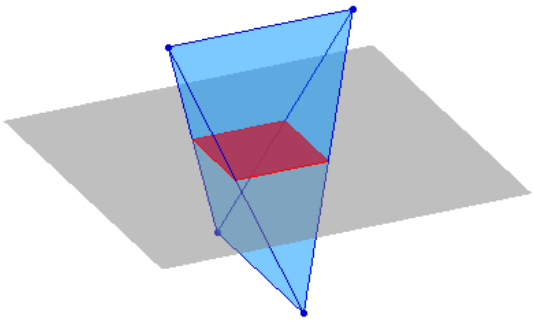
2.12.1 somma = 2 o -2



Significa che un nodo sta da una parte rispetto al piano della frattura e gli altri tre dalla parte opposta. In questo caso si ricercano il *lonely_point* e i *nodes_together*. Nel vettore *e_temp* vengono salvati gli indici degli edge che tagliano il piano (formati dal *lonely_point* e da ciascuno dei *nodes_together*). Se gli edge non sono giá stati analizzati si procede chiamando la function *intersect_plane_edge* alla quale vengono passati gli edge ancora da esaminare. Le coordinate del punto di intersezione col piano della frattura vengono salvate in *node_plane* e in checked (in edge) si salva la posizione su *node_plane* del punto appena trovato.
Viene in seguito chiamata la funzione *intersect_3D*, che rivela se il tetradro in esame é effettivamente tagliato. In tal caso, l'algoritmo di *finding_first_tet* aggiorna lo status del tetraedro a 0 nella struttura *tet* e riempie il campo *cut_tet.points* nella struttura *info_fract*. Viene in seguito eseguito lo script *slicing_tet*, al quale vengono passati tre vettori di nodi (*up,middle e down*) necessari per la sottopoliedrizzazione del tetraedro.
Si chiama *enqueue_tet_to_check* per mettere in coda i tetraedri vicini.

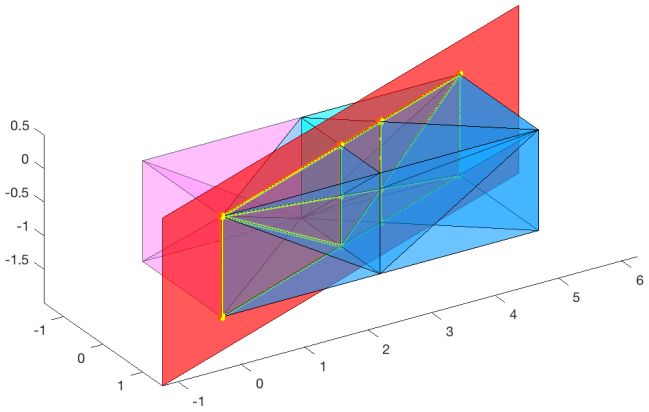


2.12.2 somma = 0

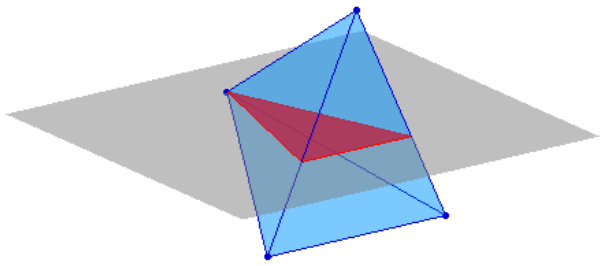


Significa che i nodi stanno in coppie opposte rispetto al piano della frattura. In questo caso si ricercano i nodi che stanno dalla stessa parte rispetto al piano e, grazie alla funzione *which_edge* si individuano i lati che intersecano il piano della frattura. Se le intersezioni di questi edge col piano della frattura sono sconosciute si esegue *intersect_plane_edge* e si salvano le informazioni sul nodo appena trovato in *node_plane*. Viene poi chiamata *intersect_3D* che controlla se il tetraedro é tagliato o meno e nel caso in cui questo sia tagliato vengono inseriti: lo status del tetraedro appena consi-

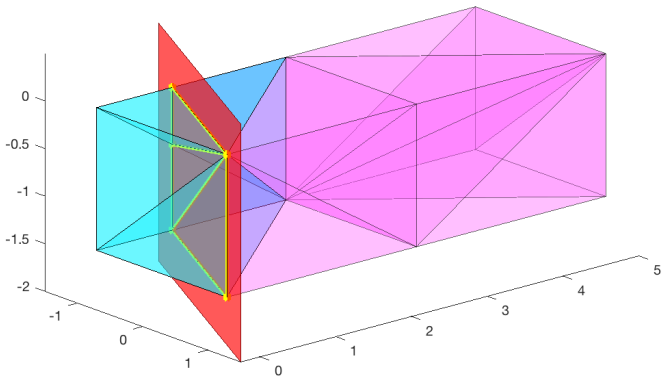
derato nella struttura *tet* ele coordinate dei punti del taglio in *info_fract* al campo *cut_tet.points*. Viene poi eseguito lo script *slicing_tet*, al quale vengono passati tre vettori di nodi (*up,middle e down*) necessari per la sottopoliedrizzazione del tetraedro. Si chiama *enqueue_tet_to_check* per mettere in coda i tetraedri vicini.



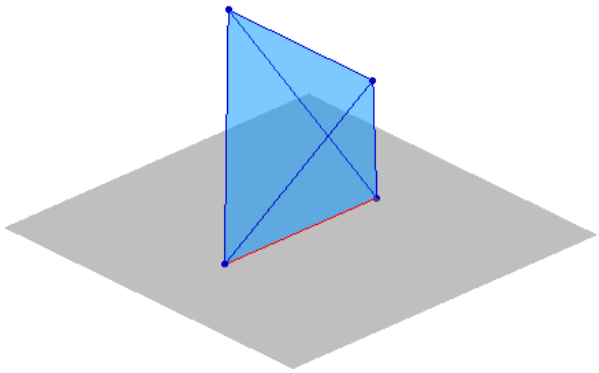
2.12.3 somma = 3 o 5



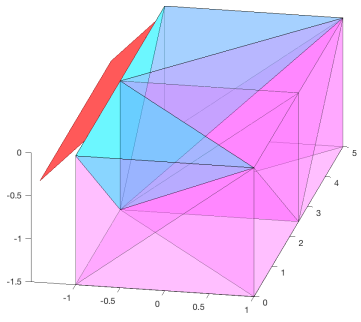
Significa che un nodo giace sul piano della frattura, due da una parte rispetto al piano e l'ultimo dalla parte opposta. Si ricercano, analogamente a prima, i nodi dalla stessa parte rispetto al piano (*nodes_together*) e il nodo solitario (*lonely_point*). Vengono salvati in *e_temp* i lati che intersecano il piano, si trovano i punti di intersezione se sconosciuti grazie a *intersect_plane_edge* e il punto avente side 4 (cioé quello che giace sul piano) viene salvato su *node_plane* (a meno che ció non sia giá stato fatto) non come derivante da un edge, bensí dal nodo di un tetraedro (campo *from_edge* = 0). Viene chiamata *intersect_3D*. Se restituisce che il tetraedro ́tagliato si inseriscono i punti del taglio in *info_fract* al campo *cut_tet.points* e viene poi chiamato lo script *slicing_tet*, al quale vengono passati tre vettori di nodi (*up,middle e down*) necessari per la sottopoliedrizzazione del tetraedro. Si chiama *enqueue_tet_to_check* per mettere in coda i tetraedri vicini.



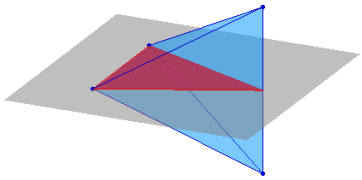
2.12.4 somma = 6 o 10



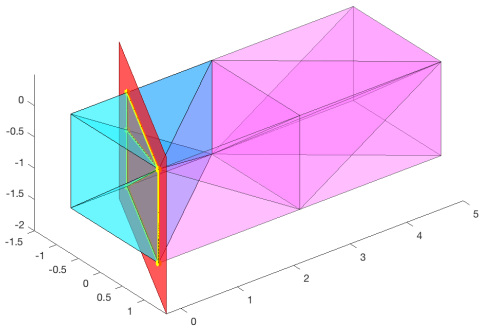
Significa che due nodi giacciono sul piano della frattura e i restanti due si trovano dalla stessa parte. Si ricercano i due nodi sul piano (*nodes_on_plane*) e i *nodes_together*. Si salvano in *node_plane* le informazioni, se sconosciute, dei nodi situati sul piano e viene chiamata *intersect_3D* che stabilirá se il tetraedro é tagliato (cioé se il poligono della frattura non ricopre lo spigolo del tetraedro per intero). Se il tetraedro é effettivamente tagliato vengono inserite le info dei punti del taglio al campo *cut_tet.points* di *info_fract* e vengono inserite le facce della poliedrizzazione senza chiamare lo script *slicing_tet*. Viene invece chiamato lo script *enqueue_tet_to_check* per continuare la ricerca.



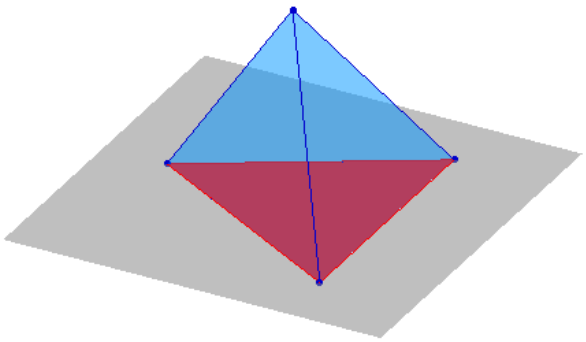
2.12.5 somma = 8



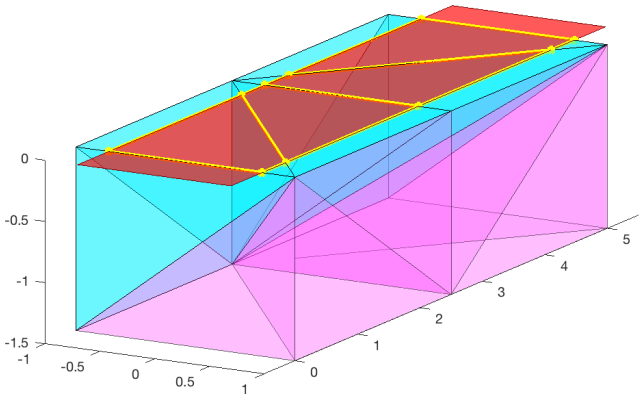
Significa che due nodi giacciono sul piano della frattura e i restanti 2 si trovano in parti opposte rispetto ad esso. Si ricercano i due nodi sul piano (*nodes_on_plane*) e i due punti solitari (*lonely_point*). Grazie a which_edge si prende in esame il segmento costituito dai due nodi opposti rispetto al piano. Si calcolano tutte le informazioni sconosciute e si inseriscono eventualmente in *node_plane* i due nodi sul piano e il punto di intersezione. Viene chiamata la funzione *intersect_3D* e, nel caso in cui questa confermi il taglio, si aggiornano lo status del tetraedro, si inseriscono i punti del taglio in *info_fract* nel campo *cut_tet.points* e viene poi chiamato lo script *slicing_tet*, al quale vengono passati tre vettori di nodi (*up,middle e down*) necessari per la sottopoliedrizzazione del tetraedro. Si chiama *enqueue_tet_to_check* per mettere in coda i tetraedri vicini.



2.12.6 somma = 11 o 13



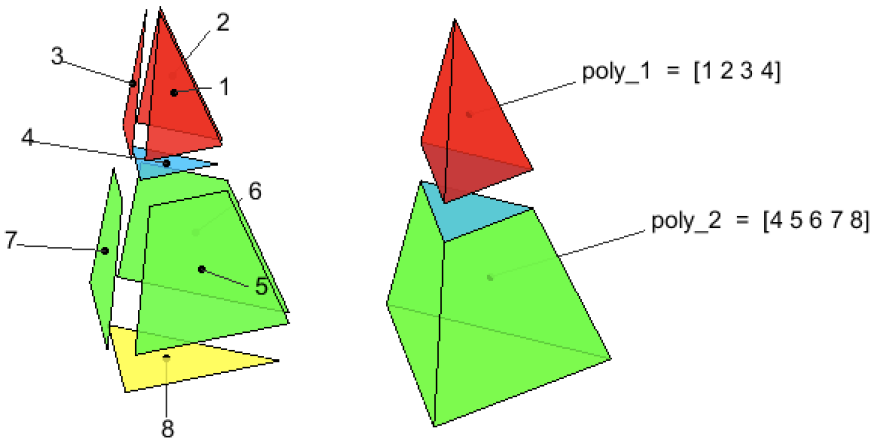
Significa che 3 punti giacciono sul piano della frattura. Si ricercano i tre nodi sul piano *nodes_on_plane* e il punto solitario (*lonely_point*). Non c'é bisogno di chiamare la function *intersecting_plane_edge* in quanto i tre punti derivano dalla tetraedrizzazione iniziale; questi vengono al massimo inseriti in *node_plane*. La funzione *intersect_3D* completa il lavoro e stabilisce se il tetraedro é effettivamente tagliato (e quindi se il poligono della frattura non ricopre per intero la faccia del tetraedro). In questo caso lo script *slicing_tet* non viene chiamato: le facce della sottopoliedrizzazione coincidono con quelle del tetraedro di partenza. Se é tagliato si chiama *enqueue_tet_to_check* per mettere in coda i tetraedri vicini. Se invece il tetraedro non viene tagliato e la faccia che giace sul piano é completamente interna alla frattura é probabile che i suoi vicini siano tagliati, viene quindi chiamato lo script *filling_queue_temp* che riempie la coda secondaria.
L'*edge_to_avoid* serve per ottimizzare l'analisi dei tetraedri vicini; quando questi vengono inseriti nella coda secondaria *queue_temp*, é infatti possibile che si appoggino sulla frattura interamente con una faccia e che anche i loro tetraedri vicini debbano essere inseriti nella coda secondaria. Vengono quindi inseriti tutti i vicini, tranne quelli che condividono l'eventuale *edge_to_avoid*, poiché quei tetraedri sono giá stati considerati.



2.13 Script di tetraedrizzazione e poliedrizzazione

2.13.1 Slicing_tet

É uno script che crea la poliedrizzazione di un tetraero tagliato nel caso in cui questo sia sottopoliedrizzato in due poliedri distinti.
A seconda delle somme dei quattro side riempie il campo *cut_tet.faces* di *info_fract*. Le facce della poliedrizzazione sono individuate da punti, indicizzate e inserite nei campi *cut_tet.poly_1* e *cut_tet.poly_2* di *info_fract*. Mostriamo un esempio nell'immagine che segue:



2.14 filling_queue_temp

Filling_queue_temp é uno script che riempie la coda secondaria. Vengono aggiunti tetraedri in questa coda quando una faccia é completamente interna alla frattura, un caso particolare della somma 11 o 13. Dopo aver trovato l'indice della faccia che sta sul piano, si cambia lo status al tetraedro simmetrico ad essa. Successivamente, si aggiungono in *queue_temp* tutti i vicini di edge del tetraedro in esame, a patto che il loro status sia sconosciuto e che il vicino di edge non sia da ignorare (*edge_to_avoid*).

2.15 Ricerca di tutti gli altri tetraedri tagliati

Dopo aver trovato il primo tetraedro tagliato, la ricerca continua finché *queue* e *queue_temp* contengono tetraedri da analizzare. Prima di tutto viene chiamato lo script *checking_queue_temp* che svuota la coda secondaria e ottimizza la ricerca dei triangoli tagliati, andando a considerare i tetraedri potenzialmente tagliati.
Successivamente viene considerata la coda primaria *queue*, costituita dai tetraedri vicini ai tetraedri tagliati. Prima di procedere in modo analogo a *finding_first_tet*, si controlla che l'id del tetraedro messo in queue non sia minore del numero di tetraedri giá analizzati nella ricerca del primo tetraedro. In tal caso *it_is_near* verifica se il tetraedro é giá stato salvato nei vicini di un tagliato o meno. Se il tetraedro é ancora da analizzare, viene chiamata la funzione *which_side_3D* per ciascuno dei nodi del tetraedro per i quali il side é ancora sconosciuto. Se il side dei quattro nodi risulta lo stesso, il tetraedro viene messo in *near_tet*, perché vicino a un tagliato in quanto si trova in *queue*, ma non é lui stesso un tagliato per via delle considerazioni fatte sui side. L'algoritmo procede considerando le somme dei side e adottando le stesse scelte di ricerca del primo tetraedro tagliato.