FINDER TETRA

Giorgio Racca, Luca Rossi

Progetto 3

Sommario

**1 Introduzione alla struttura del progetto2**

* 1. La classe TestBox2
  2. La classe FinderTetra4

**2 Il metodo FindCell5**

2.1 Ricerca del primo tetraedro intersecato5

2.2 Visita della mesh a partire dal primo tetraedro7

**3 SignTest9**

**4 SphereTest11**

4.1 Coordinate estreme del poligono11

4.2 Coordinate estreme della cella e intersezione13

**5 FinalTest15**

5.1 Calcolo della sezione15

5.2 Check punto della sezione interno al poligono17

5.3 Check intersezione dei lati18

5.4 Check punto poligono interno alla sezione19

5.5 La funzione PointInPolygon21

5.6 Considerazioni sulla complessità computazionale24

**6 Test e risultati finali25**

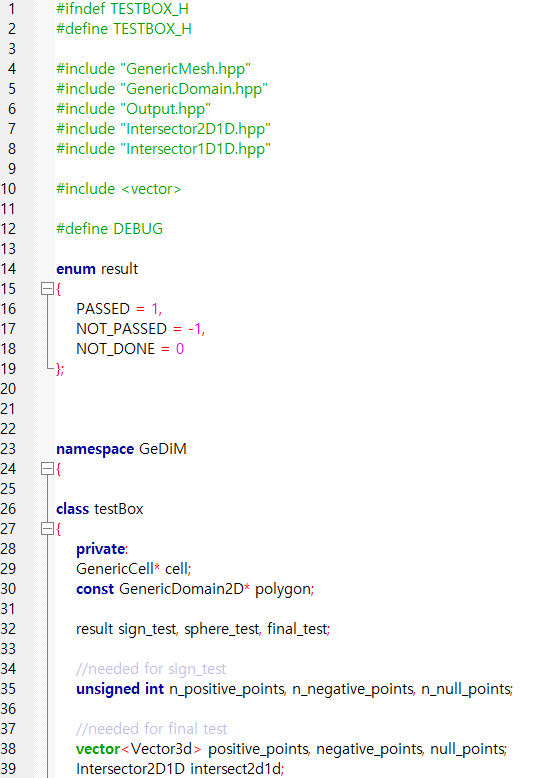
**1 INTRODUZIONE ALLA STRUTTURA DEL PROGETTO**

Lo scopo del progetto è quello di individuare i tetraedri di una mesh intersecati da un poligono convesso. Il nostro compito è stato quello di implementare il metodo *FindCell* della classe *FinderTetra*. A tal fine, abbiamo raccolto vari metodi in una classe ausiliaria denominata *TestBox*. Questi metodi, pur operando con accuratezza e complessità diverse, condividono lo stesso obiettivo: testare se c’è o meno intersezione fra il poligono e una data cella.

L’implementazione del metodo *FindCell* si trova nel file *FinderTetra.cpp*, mentre la classe ausiliaria è divisa in *TestBox.h* e *TestBox.cpp*.

**1.1 La classe TestBox**

Di seguito riportiamo il file TestBox.h:





La classe *TestBox* comprende tra i suoi **membri** privati:

* Un puntatore alla cella ed un puntatore ad un poligono costante sui quali effettuerà i test.
* I risultati dei test definiti dall’*enum* *result*.
* Il numero di punti positivi (dalla parte della normale del poligono), negativi (nell’altro semispazio) e nulli (che giacciono sul piano del poligono). Ci serviranno per il metodo *ComputeSignTest*.
* Tre vettori contenenti rispettivamente i punti positivi, negativi e nulli del tetraedro. Ci serviranno per il metodo *ComputeFinalTest*.
* Gli oggetti di tipo *intersector2D1D* e *intersector1D1D* che utilizzeremo rispettivamente per il calcolo della sezione generata dall’intersezione del piano del poligono con il tetraedro e per il calcolo dell’effettiva intersezione tra i segmenti del poligono e i segmenti di tale sezione. Abbiamo scelto di inserirli come membri della classe in modo tale da non doverli creare ogni volta che chiamiamo il metodo *ComputeFinalTest* per verificare l’effettiva intersezione del poligono con il tetraedro.
* I vettori *polyMin* e *polyMax* contenenti rispettivamente i minimi e i massimi di ogni coordinata del poligono. Ci serviranno per il metodo *ComputeSphereTest*.

La classe *TestBox* comprende tra i suoi **metodi**:

* Costruttore e distruttore di default.
* Costruttore con argomenti che copia i puntatori al poligono e alla cella e inizializza i risultati dei test a *NOT\_DONE*. Setta, inoltre, i parametri dell’*Intersector2D1D* e calcola i vettori *polyMin* e *polyMax* del poligono.
* *SetCell* che permette di modificare la cella che stiamo testando. Oltre ad aggiornare il puntatore alla cella, il metodo azzera i risultati dei vari test. Il metodo, infatti, pone a *NOT\_DONE* i *result*, azzera i contatori dei punti (positivi, negativi e nulli) e svuota i rispettivi vettori.
* I metodi *result* dei vari test che restituisce i *result* dei rispettivi test.
* I metodi *compute* di ogni test che restituiscono un *Output success* se il test è stato eseguito correttamente, altrimenti un *generic error*.

**1.2 La classe FinderTetra**

Di seguito riportiamo il file FinderTetra.hpp:



La classe *FinderTetra* comprende tra i suoi **membri** privati:

* *meshpointer*: puntatore alla mesh.
* *cellFound*: lista di puntatori a celle. Sarà riempito con le celle intersecate tramite il metodo *FindCell*.
* *skipTest2*: booleano che prescrive o meno l’esecuzione di *SphereTest.* Non presente nel progetto originale*.*

La classe *FinderTetra* comprende tra i suoi **metodi**:

* costruttore e distruttore.
* Un setter per il membro *meshpointer*.
* Un setter per il membro *skipTest2.* Non presente nel progetto originale*.*
* *FindCell*: il metodo principale. Riempie il membro *cellFound* con le celle intersecate dal poligono fornito come parametro di ingresso.
* Un getter per il membro *cellFound.*

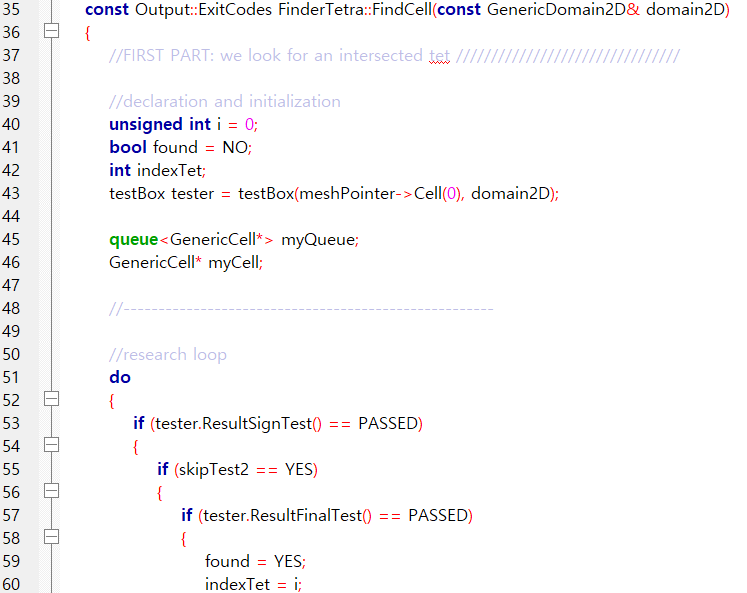
**2 FIND CELL**

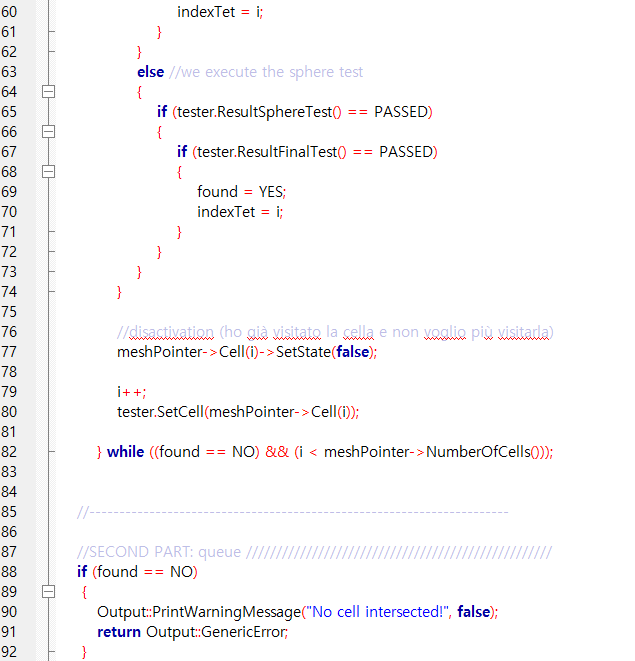
Il metodo *FindCell* della classe *FinderTetra* viene invocato nel *main* ricevendo come parametro di ingresso il poligono intersecante. Oltre a restituire un *ExitCodes*, il compito di *FindCell* è quello di riempire la lista *CellFound* (membro di *FinderTetra*) con i puntatori alle celle della mesh intersecate dal poligono.

Possiamo suddividere il codice del metodo *FindCell* in due sezioni concettualmente distinte. Grazie al primo blocco di istruzioni, la mesh viene percorsa alla ricerca di un tetraedro intersecato. Una volta trovato, nel secondo blocco viene implementata una sorta di visita in ampiezza della mesh a partire dal primo tetraedro. Al termine di tale visita, la lista *CellFound* risulta riempita correttamente.

Nei paragrafi successivi si farà più volte riferimento ai metodi della classe *TestBox*. Per una descrizione più dettagliata (nonché il codice) rimandiamo alle sezioni a essi dedicate.

**2.1 Ricerca del primo tetraedro intersecato**

****

****

Il primo blocco di istruzioni è costituito essenzialmente da un unico while-loop. **Si scorre la mesh** fino a quando non viene individuato un tetraedro intersecato (ovviamente il ciclo termina anche nel caso in cui l’intera mesh sia stata percorsa). Per ogni cella, si effettuano **fino a un massimo di tre test**, la cui accuratezza cresce progressivamente. A tal fine, viene creata un’istanza della classe *TestBox*, denominata *tester*. Si utilizzerà quest’unica istanza per tutto il metodo *FindCell*. Se una data cella supera tutti i test, allora siamo certi che sia una cella intersecata dal poligono. Per tenere traccia delle celle analizzate, le disattiviamo una volta terminate le operazioni su di esse.

In particolare, i test eseguiti sono i seguenti:

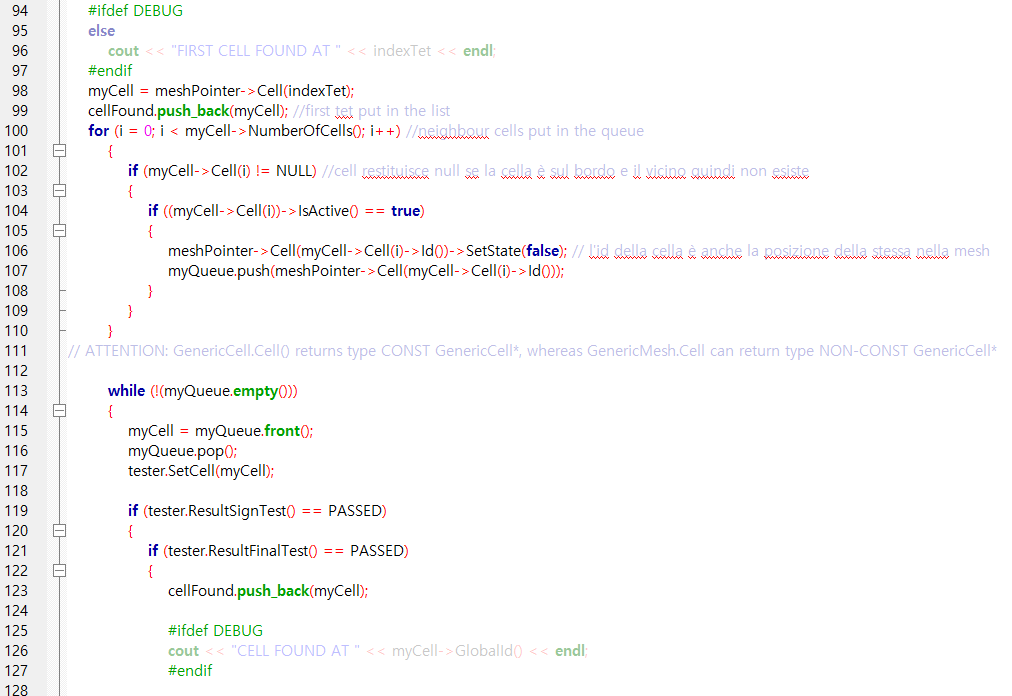
1. *SignTest*: questo test risulta superato se i vertici della cella considerata stanno in parte sopra e in parte sotto al piano del poligono intersecante, oppure almeno uno giace su di esso. In questo modo, vengono scartate tutte le celle che non sono tagliate almeno dal piano del poligono (condizione necessaria per essere intersecate anche dal poligono).
2. *SphereTest*: lo scopo di questo test è scartare i tetraedri che, pur giacendo sul piano del poligono, non sono tagliati poiché si trovano a “grande distanza” dal poligono. Questa nozione di “grande distanza” è definita come segue: *una cella tetraedrica e un poligono si trovano a “grande distanza” se le loro bounding-box non si intersecano*.

In realtà, questo test *non* viene eseguito se i vertici del poligono si trovano tutti all’esterno del dominio cubico tetraedrizzato (quest’informazione viene processata nel *main* e salvata nel membro *skipTest2* della classe *FinderTetra*). In tal caso è *probabile,* ma non certo, che i lati del poligono siano esterni al dominio. Di conseguenza, il poligono sarebbe (ai fini del progetto) assimilabile a un piano, e quindi questo test risulterebbe più dispendioso che utile.

1. *FinalTest*: è un test di intersezione definitivo (nel senso che individua con “esattezza” se c’è o meno intersezione), ma è anche il più laborioso. Per questo motivo, è l’ultimo a essere effettuato.

Se n è il numero di celle presenti nella mesh, la **complessità in tempo** di questa sezione è **O(n)** operazioni di analisi, dove per analisi si intende l’esecuzione dei tre test; la **complessità spaziale**, invece, è **O(1)** memoria aggiuntiva.

**2.2 Visita della mesh a partire dal primo tetraedro**

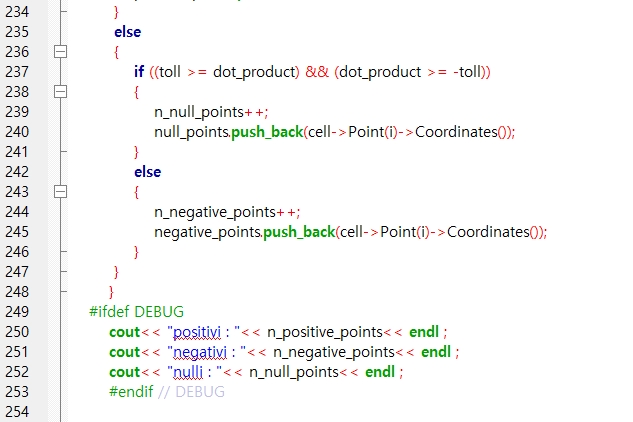
****

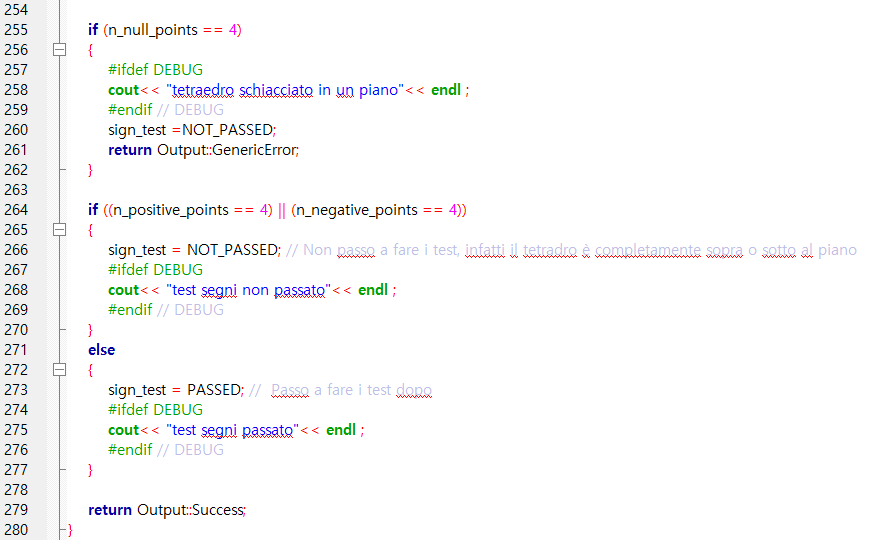
****

Il secondo blocco di istruzioni non è altro che una **visita in ampiezza** della mesh a partire dal primo tetraedro individuato nella sezione precedente. Si utilizza, perciò, come struttura ausiliaria una **coda**, denominata *myQueue*. Inseriamo il primo tetraedro trovato nella lista *CellFound* e i suoi vicini nella coda *myQueue*. Poi si esegue un altro while-loop che termina quando *myQueue* è vuota. A ogni iterazione, si estrae una cella dalla coda (nello specifico si eseguono i metodi *front* e *pop* della classe *queue*) e si verifica se c’è intersezione con il poligono. Prima di effettuare *FinalTest*, si esegue *SignTest* che è decisamente meno oneroso dal punto di vista computazionale. Non ha invece senso sottoporre la cella a *SphereTest* poiché, essendo adiacente a una cella intersecata, sarà sicuramente “vicina” al poligono. Nel caso la cella considerata sia tagliata dal poligono, le celle adiacenti -se ancora attive- vengono inserite nella coda. Inoltre, le celle vicine, prima di essere inserite nella coda, vengono disattivate. Grazie a questo accorgimento, ogni cella candidata viene inserita nella coda una sola volta. In questo modo, si risparmia anche in spazio. Più precisamente, questa sezione di codice presenta una **complessità di O(n) visite e O(n) celle di memoria** occupate. **3 SIGN-TEST**

Di seguito riportiamo il codice del metodo *ComputeSignTest*, localizzato nel file *TestBox.cpp:*







Lo scopo di questo metodo è quello di classificare i vertici di una data cella tetraedrica in base alla loro posizione relativa al piano su cui giace il poligono intersecante.

Nello specifico, si calcola il **prodotto scalare** fra la **normale del poligono** e **il vettore congiungente il centroide del poligono al vertice** considerato. Si osservi che, poiché la normale è un versore, il prodotto scalare non è altro che la distanza del punto dal piano. Quindi, se il risultato è più piccolo in modulo di una certa tolleranza, si assume che il punto appartenga al piano; altrimenti, in base al segno viene classificato come positivo o negativo. Ai fini del test sarebbe sufficiente conoscere quanti punti appartengono a ogni categoria. Tuttavia, questa classificazione è **sfruttata anche dal metodo *FinalTest***. Perciò, abbiamo scelto di salvare i punti nei vari vettori membri *null\_points*, *positive\_points* o *negative\_points*.

Il test ha esito positivo in due casi: o esistono punti di segno opposto, oppure esiste almeno un punto appartenente al piano, vengono quindi scartati i tetraedri che si trovano o interamente al di sopra o interamente al di sotto del piano. Invece, nel caso in cui tutti e quattro i vertici risultino nulli, il metodo restituisce un errore.

Per quanto riguarda il costo computazionale, **per ogni vertice** si effettuano **tre moltiplicazioni** oltre alla **memorizzazione** in uno dei tre vettori. Il costo non dipende dal numero di lati del poligono.

**4 SPHERE-TEST**

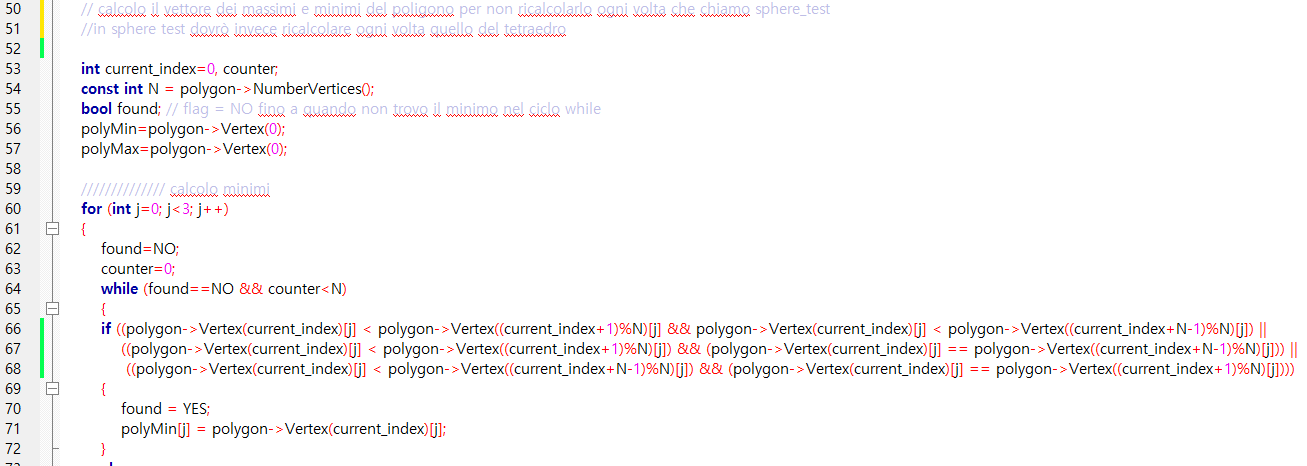
Lo scopo di questo metodo è quello di verificare se c’è o meno **intersezione fra i *bounding-box*** del poligono e di una data cella tetraedrica.

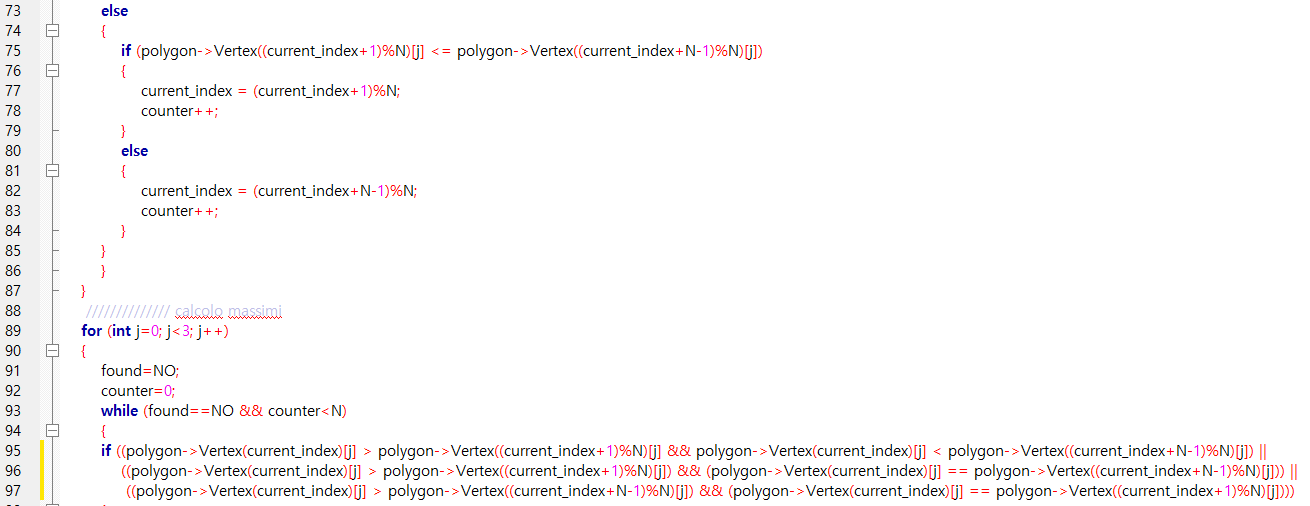
A differenza di cosa suggerirebbe il nome, il *bounding-box* impiegato è un parallelepipedo parallelo agli assi cartesiani e non una sfera. Questa lieve discrepanza fra il nome e le operazioni effettivamente eseguite è una conseguenza dell’approccio *top-down* impiegato in fase di progettazione. Inizialmente, infatti, pensando di utilizzare delle sfere, abbiamo scritto porzioni di codice che invocavano questo metodo, che al tempo non era stato ancora implementato.

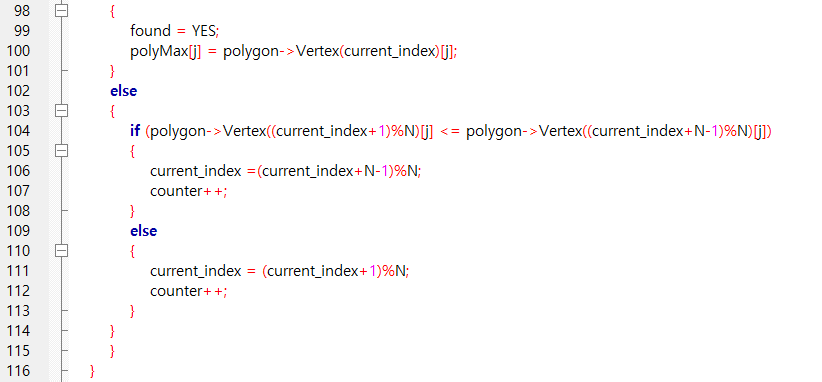
Concettualmente le operazioni eseguite sono le seguenti: calcolare le coordinate estreme (massime e minime) del poligono, calcolare le coordinate estreme della cella e, infine, effettuare un’intersezione che abbia senso dal punto di vista numerico.

**4.1 Coordinate estreme del poligono**

Il **calcolo delle coordinate estreme del poligono** può essere effettuato una sola volta. Per questo motivo, la sezione di codice che lo implementa si trova nel costruttore con argomenti della classe *TestBox*.







La ricerca implementata è leggermente più sofisticata della banale ricerca esaustiva in cui si effettuano sempre tutti i confronti. Infatti, si sfrutta **l’ipotesi di convessità** del poligono per ridurre la complessità temporale al caso ottimo e al caso medio.

Nello specifico il codice si compone di due for-loop (uno per i massimi e l’altro per i minimi) che scorrono le tre coordinate cartesiane. Per ogni coordinata, si esegue un while-loop (annidato nel for-loop) nel quale si cerca il massimo/minimo valore assunto fra i vertici del poligono. Nel seguito, descriveremo nel dettaglio l’algoritmo per individuare il minimo di una data coordinata; per il massimo si procede in maniera analoga.

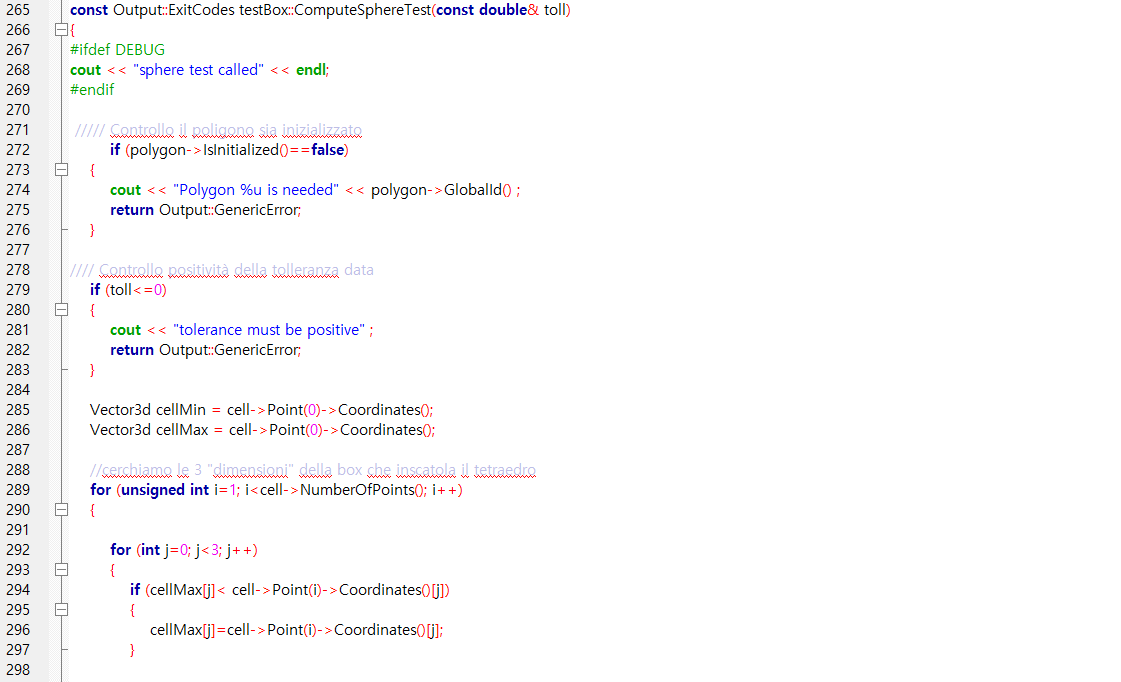
Dato un certo vertice si verifica una fra le seguenti:

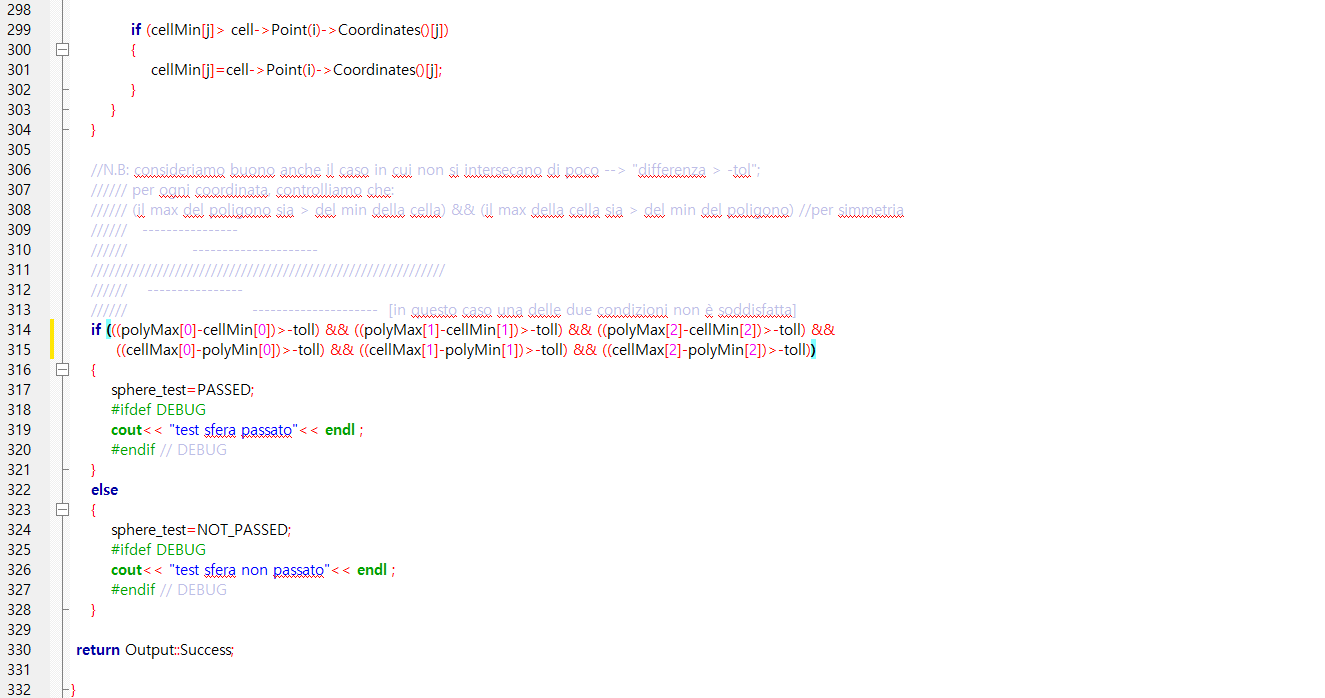
* Il vertice è di estremo per la coordinata considerata. La ricerca si arresta, e il valore della coordinata viene inserito nel vettore *polyMin* (membro di *TestBox)*. Ciò si verifica in due casi: se la coordinata del vertice è più piccola sia di quella del vertice successivo e sia di quello precedente, oppure se è più piccola di quella del vertice precedente e uguale a quella del successivo (o viceversa).
* Il vertice è compreso fra un vertice che ha coordinata minore e uno che ha coordinata maggiore. In questo caso, all’iterazione successiva il vertice considerato sarà quello con coordinata minore. Nel caso in cui tutti e tre i vertici abbiano la stessa coordinata, la scelta del vertice su cui eseguire la prossima iterazione non è importante.

Sia p il numero di lati del poligono; la **complessità temporale** è O(1) al caso ottimo, O(p/2) al caso medio e O(p) al caso pessimo.

**4.2 Coordinate estreme della cella e Intersezione**

Il resto del codice si trova nel metodo *ComputeSphereTest*:





Per il calcolo delle coordinate di estremo di una data cella si procede confrontando i valori dei quattro vertici, per un totale di sei confronti per coordinata (tre per il massimo e altrettanti per il minimo).

Infine, occorre verificare se i *bounding-box* si intersecano. Affinché ci sia intersezione, si deve avere **sovrapposizione lungo tutte e tre le direzioni cartesiane**. A sua volta, la condizione di sovrapposizione lungo una data direzione si esprime nel modo seguente: *il massimo valore del poligono deve essere maggiore del minimo valore della cella, e contemporaneamente il massimo valore della cella deve essere maggiore del minimo valore del poligono*.

Caso in cui la condizione è soddisfatta:

Caso in cui la condizione *non* è soddisfatta perché il massimo blu è minore del minimo arancione:

Per ottenere un algoritmo più sicuro dal **punto di vista numerico, “**ci siamo tenuti larghi”. Il test è superato anche se il massimo è leggermente più piccolo del minimo (in aritmetica infinita in questo caso non ci sarebbe sovrapposizione). Questa condizione si esprime imponendo che *la differenza fra massimo e minimo sia maggiore di una certa tolleranza cambiata di segno*.

Caso in cui il massimo blu è così vicino al minimo arancione che conviene assumere che ci sia sovrapposizione:

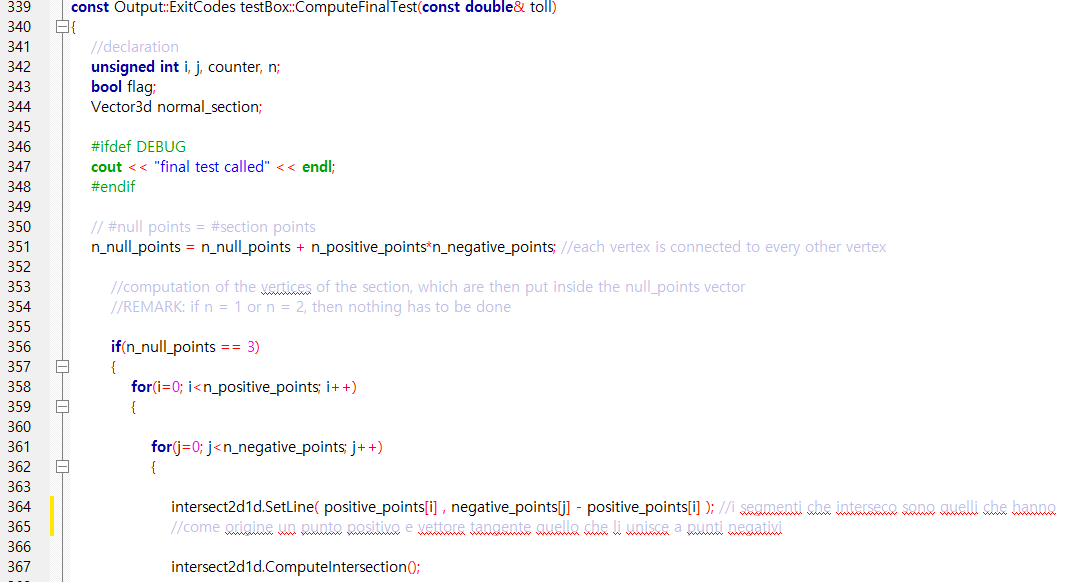
**5 FINAL-TEST**

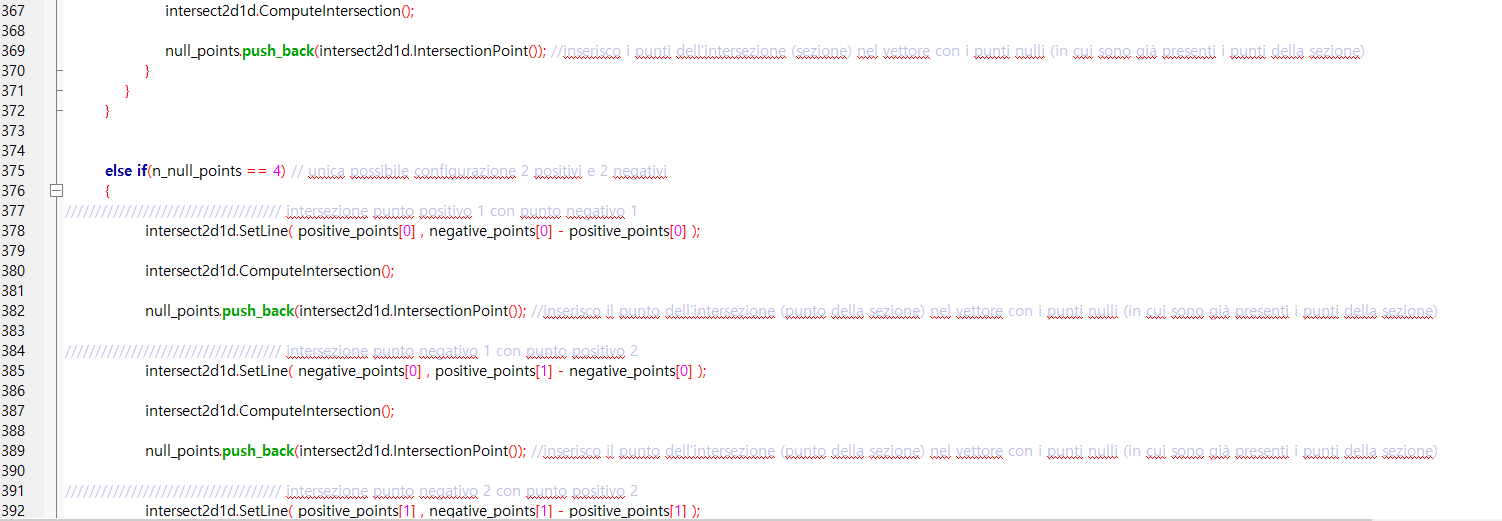
Lo scopo del metodo è quello di verificare l’esistenza o meno di un’**effettiva intersezione** tra il poligono/piano e il tetraedro considerato. Per far ciò, abbiamo pensato di ricostruire la sezione data dall’intersezione tra il piano del poligono e il tetraedro, per poi verificare l’intersezione tra la sezione e il poligono. In questo modo abbiamo trasformato il problema in origine tridimensionale in un **problema equivalente bidimensionale**.

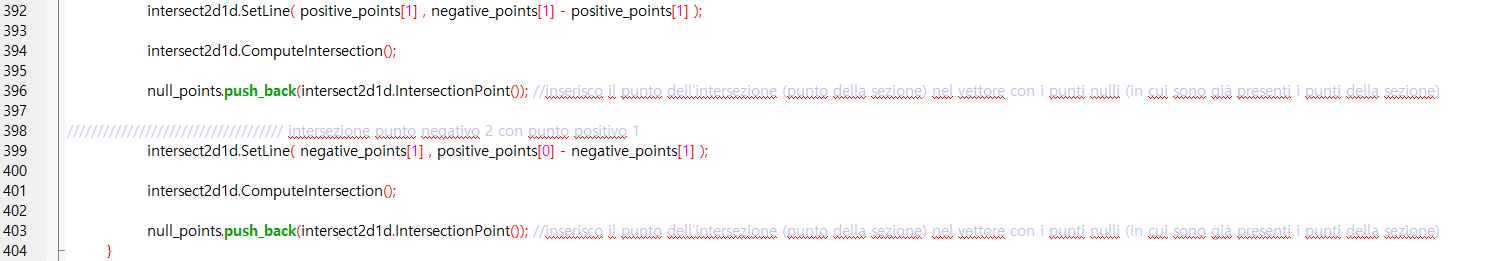
Nota bene: considereremo il tetraedro tagliato dal poligono anche se il poligono non interseca facce o lati del solido ma ne taglia comunque il volume.

**5.1 Calcolo della sezione**

Qui di seguito si riporta il codice, localizzato in *TestBox.cpp*, che permette di calcolare il vettore contenente i punti della sezione:







È facile notare come il numero di punti della sezione è dato dalla **formula**:

Dove , e sono già stati calcolati quando abbiamo eseguito il metodo *SignTest*. La formula vale poiché in un tetraedro ogni vertice è collegato ad ogni altro vertice. Di conseguenza, ad ogni coppia di vertici di cui uno positivo e uno negativo, corrisponde un nuovo punto di intersezione che si andrà ad aggiungere ai punti nulli calcolati in precedenza.

Una volta che si conosce il numero di punti della sezione, si procede **dividendo in due casi principali**:

* Sezione di tre vertici
* Sezione di quattro vertici

Nel primo caso, come già visto, per ogni coppia di punti di cui uno positivo e uno negativo, si calcola l’intersezione e la si aggiunge al vettore *null\_points*, usato d’ora in poi come vettore di punti della sezione. Si noti che, in questo caso, i punti sono automaticamente memorizzati nel vettore in maniera tale che **a vertici contigui corrispondano indici contigui** (tranne per il primo e l’ultimo), in quanto il vettore ha solo tre elementi. Nel secondo caso, dalla formula considerata si può notare che le uniche due possibilità sono o che il numero di punti nulli sia già quattro (tetraedro inammissibile) o che ci siano due punti positivi e due punti negativi come in figura.

P1

N1

N2

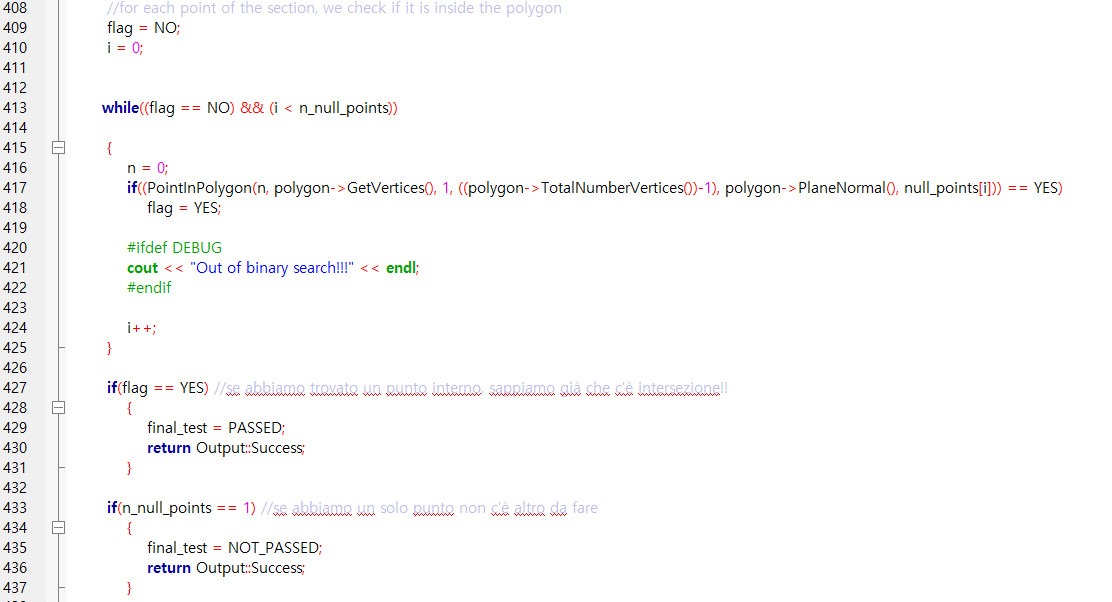
P2

In questo disegno il piano blu è quello su cui giace il poligono, *P1* e *P2* sono i due punti positivi e *N1*, *N2* i due negativi. Intersecando le rette per *P1* e *N1*, quella per *N1* e *P2*, quella per *P2* e *N2* e infine quella per *N2* e *P1* con il medesimo piano, si ottengono i quattro punti di intersezione, che saranno inseriti a mano a mano nel vettore *null\_points*. Il **vettore dei punti** **nulli** risulterà allora **ordinato**, e questo accorgimento ci permetterà di semplificare di molto i controlli successivi.

Ovviamente, nel caso in cui i punti della sezione siano 1 o 2, non è necessario svolgere alcuna operazione per costruire la sezione.

**5.2 Check punto della sezione interno al poligono**

Il passo successivo consiste nel controllare **se esiste un** **punto della sezione interno al poligono**. Si riporta qui di seguito il codice che segue il calcolo della sezione.



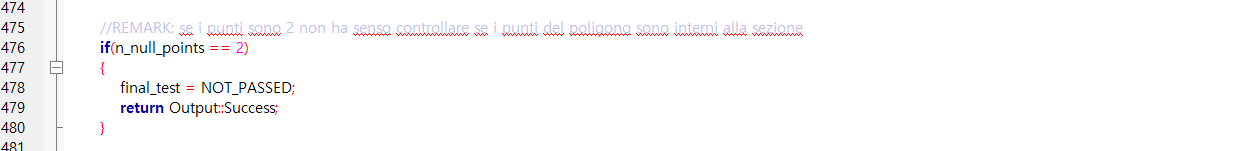
Per ogni punto in *null\_points*, fino a quando non si trova un punto della sezione interno al poligono o non si esauriscono i punti, si sfrutta la funzione ***PointInPolygon*** che restituisce vero se *null\_points[i]* è interno al poligono, falso altrimenti. Rimandiamo la spiegazione della funzione *PointInPolygon* alla successiva sezione. Al termine del ciclo, se abbiamo trovato un punto interno (ovvero *PointInPolygon* ha restituito *true*), il test è automaticamente passato e l’intersezione è assicurata. In caso contrario, il test *non* è definitivo a meno che il punto nullo non sia unico: solo in tal caso il *result* del test può essere posto a *NOT\_PASSED* e la possibilità di un’intersezione è esclusa. Un esempio del fatto che ci potrebbe essere intersezione tra la sezione e il poligono anche se nessun punto della sezione è interno al poligono è il seguente.

Come si nota, in questo caso, nessun punto della sezione (rettangolo) è interno al poligono (pentagono), tuttavia c’è comunque intersezione e sovrapposizione tra le due figure.

**5.3 Check intersezioni dei lati**

Il successivo e ancora non definitivo check consiste nel verificare la presenza di **intersezioni tra i lati della sezione e quelli del poligono**. Si riporta qui di seguito il codice con il quale si effettua il controllo.



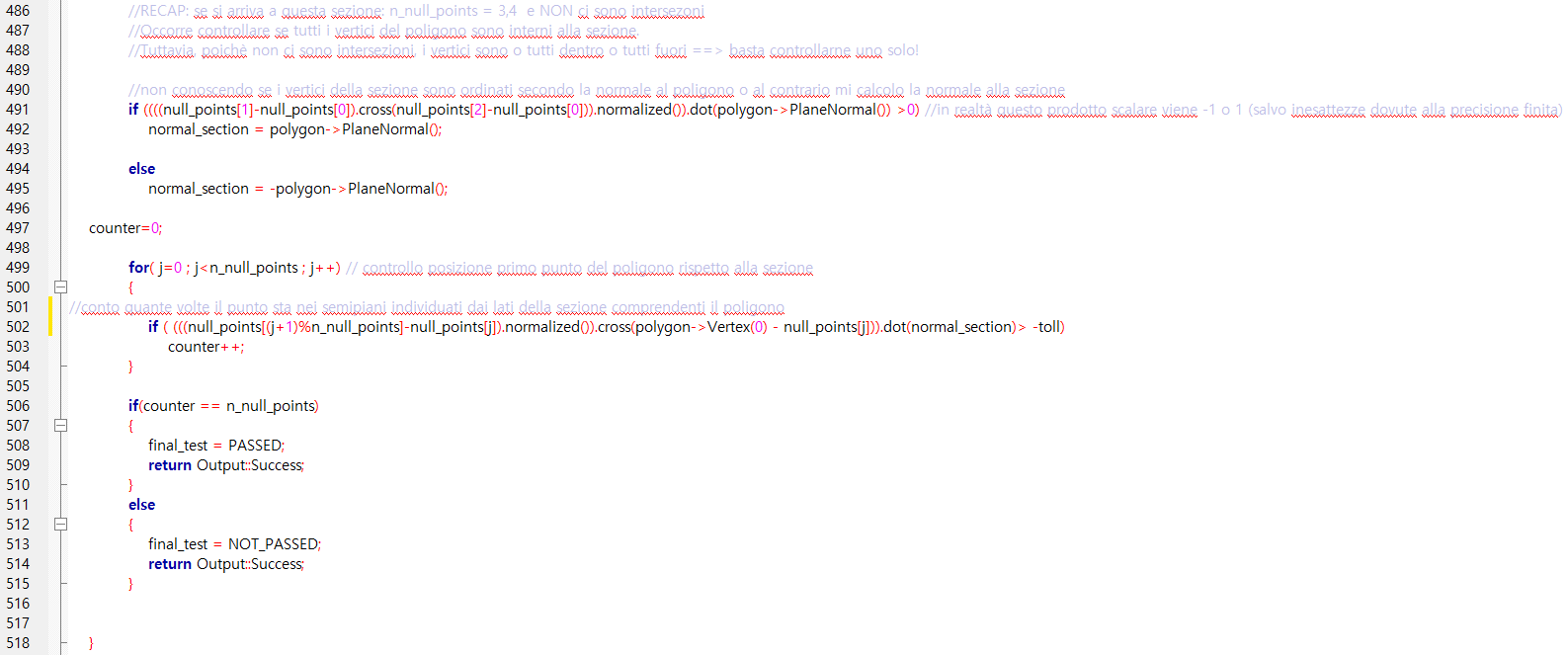


È facile notare che non basta infatti controllare che nessun punto della sezione giaccia all’interno del poligono e che nessun punto del poligono giaccia all’interno della sezione per concludere l’assenza di intersezioni tra il poligono e il tetraedro. Un caso come quello sottostante infatti **richiede necessariamente la verifica delle intersezioni**.

Come si può evincere dal frammento di codice, per verificare l’intersezione tra il poligono e la sezione si usano i metodi della classe *Intersector1D1D*. Tuttavia, questi metodi operano sul piano *Oxy* e quindi è necessario **ruotare** i punti del poligono e della sezione prima di impostare i parametri dell*’Intersector*. A tal proposito abbiamo usato i metodi *RotatedVertex* della classe *GenericDomain* per il poligono e il metodo *RotatePoint* della classe *IRotation* per la sezione. Infatti, un generico punto della sezione e ogni punto del poligono giacciono nel medesimo piano: quindi, è possibile ruotare i punti della sezione tramite i metodi già implementati per il *GenericDomain*. Si procede allora calcolando l’intersezione tra ogni lato del poligono ed ogni lato della sezione. Se l’intersezione cade all’interno di uno dei due segmenti, allora il *result* del test è posto a *PASSED*. In caso contrario, il *result* è posto a *NOT\_PASSED* solo nel caso in cui il numero di punti nulli sia pari a due. Infatti, nei restanti casi non siamo ancora in grado di affermare la mancanza di intersezione, come dimostra il seguente caso limite in cui il poligono è interamente contenuto nella sezione.

**5.4 Check punto del poligono interno alla sezione**

**L’ultimo controllo** necessario a discernere l’effettiva sovrapposizione della sezione e del poligono consiste nel **verificare se un punto del poligono è interno alla sezione**. Qui di seguito si riporta il codice situato sempre nel metodo *ComputeFinalTest* in *TestBox.cpp*.



Per prima cosa calcoliamo la **normale alla sezione.** La cosa più semplice da fare sarebbe quella di calcolare la normale come prodotto vettoriale di due lati successivi della sezione e poi normalizzare. Tuttavia, questo calcolo risulterebbe leggermente inaccurato in quanto dipenderebbe, in precisione finita, dai due lati consecutivi scelti. Abbiamo optato allora per calcolare il prodotto scalare tra la normale al poligono e il prodotto vettoriale normalizzato tra due lati consecutivi della sezione. Questo prodotto scalare, in precisione infinita, potrebbe risultare solamente più o meno uno poiché le due figure giacciono sul medesimo piano. Allora si pone se tale prodotto scalare è maggiore di zero, altrimenti si pone .

P

1

0

Con riferimento alla figura posta sopra, si calcola ora il prodotto scalare tra la normale al poligono e il prodotto vettoriale (normalizzato) tra i due vettori e . Se tale prodotto risulta essere positivo, allora siamo sicuri che il punto del poligono si trovi nel semipiano sinistro, delimitato a destra dalla retta passante per e e contenente la zona gialla in figura. Ripetendo lo stesso ragionamento per ogni punto della sezione, se il prodotto considerato è **sempre positivo** allora il punto si troverà **all’interno della sezione**. Inoltre, vista l’assenza di intersezioni tra le due figure, il poligono risulterà interamente contenuto nella sezione; poniamo allora il risultato a *PASSED*, in caso contrario a *NOT\_PASSED*.

Nota bene: anche in questo caso abbiamo usato le tolleranze per definire se il punto è interno o meno alla sezione. In generale, un prodotto misto si può riscrivere come dove α è l’angolo compreso tra i vettori , β quello tra i vettori e e sono ora le norme dei vettori. Nel nostro caso abbiamo che poiché e sono normalizzati e =0 (perché i vettori e sono paralleli). Nel nostro caso l’espressione del prodotto misto diventa quindi e notiamo che questa è proprio la distanza con segno del punto *P* dalla retta passante per i vertici *0,1* in figura.

Allora, imponendo il **abbiamo considerato punti interni anche punti leggermente al di fuori della sezione** (la loro distanza con segno poteva essere anche leggermente minore di zero) e punti molto vicini agli estremi della sezione. Come mostra la seguente figura, il punto è leggermente al di fuori della sezione (in giallo) ma è comunque considerato al suo interno poiché la sua distanza con segno (tratteggiata) è negativa ma tende a zero.

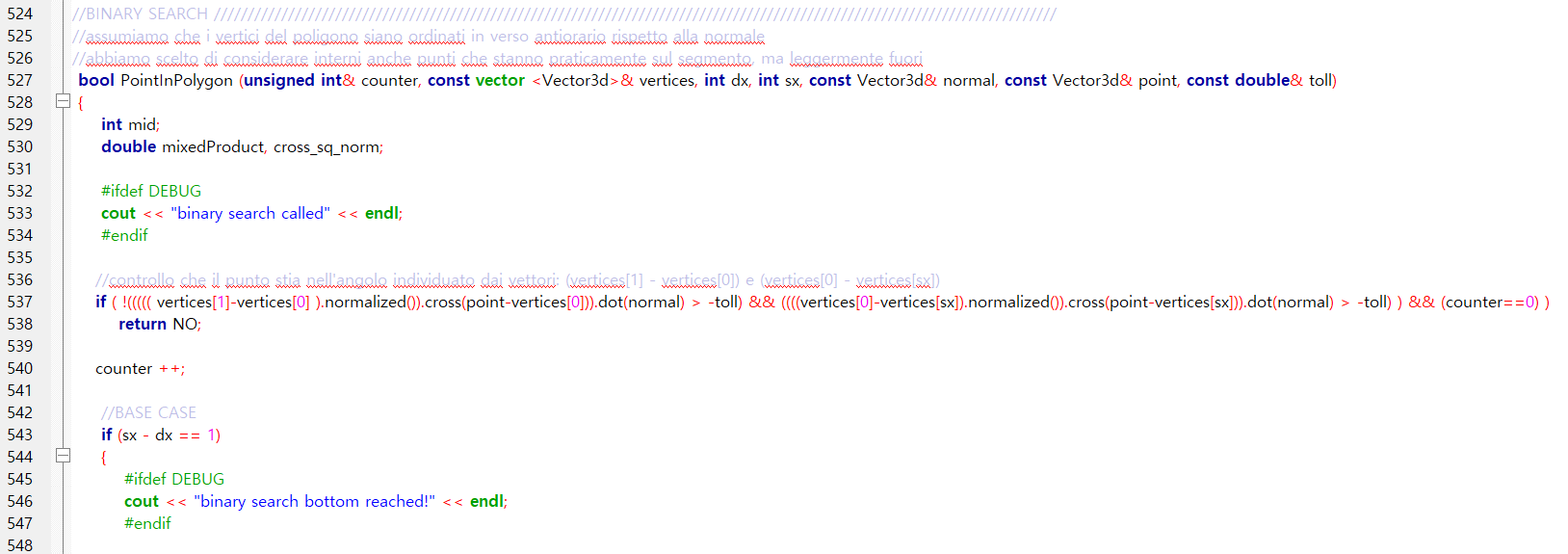
1

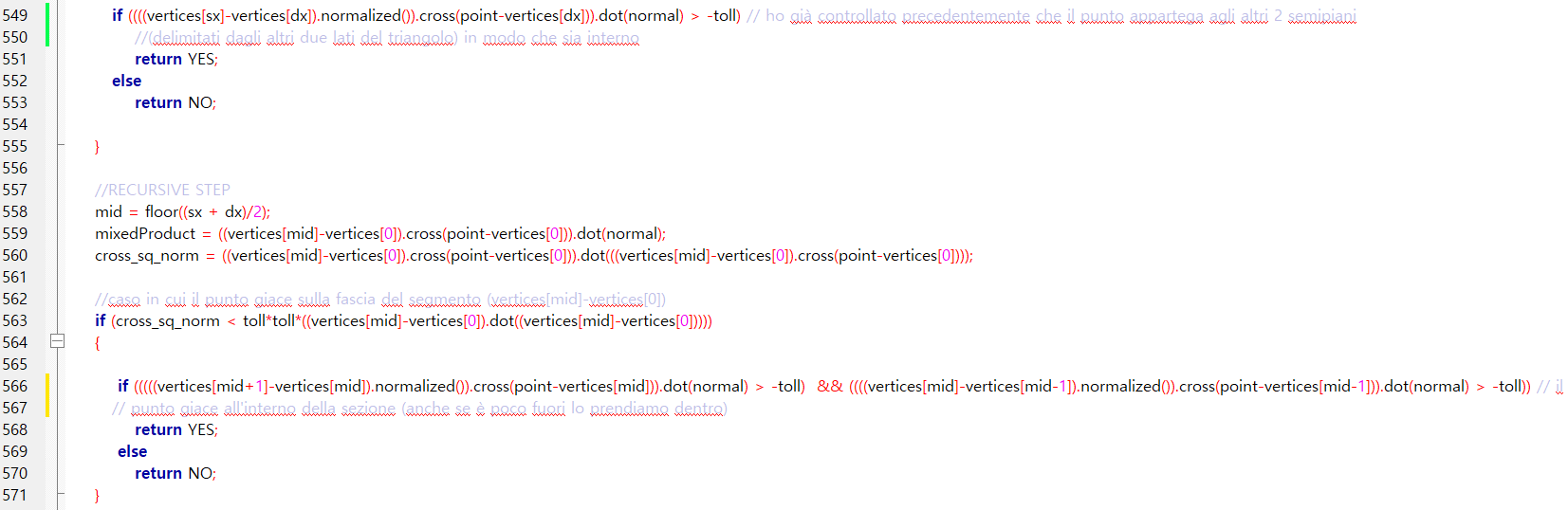
0

P

**5.5 La funzione PointInPolygon**

Per controllare se un punto giace o meno all’interno del poligono, sfruttiamo la funzione *PointInPolygon* che restituisce un booleano rappresentante l’esito della verifica. Qui di seguito riportiamo il codice della funzione, situato in calce al file *TestBox.cpp*.







La funzione sopra riportata è una funzione ricorsiva che si può modellizzare come **ricerca binaria.** I parametri di ingresso sono i seguenti:

* Un *counter* passato per riferimento che conterà quante volte sarà chiamata la funzione ricorsiva
* I vertici del poligono
* Due indici *dx*, *sx* che indicheranno di volta in volta in quale porzione del poligono potrebbe trovarsi il punto
* La normale al poligono
* Il punto che vogliamo controllare
* Una tolleranza

Per comprendere l’algoritmo, si può notare che **ogni vertice del poligono, tranne il vertice , è ordinato per angolo α crescente,** come mostrato in figura.

3

4

2

α

1

0

Definiamo ora l’angolo β come quello associato al punto *P*, e siano *sx* e *dx* due vertici del poligono tali per cui siamo sicuri che il punto *P* abbia angolo compreso tra quello associato a *sx* e quello associato a *dx*. L’obiettivo dell’algoritmo è spostare gli estremi *sx* e *dx* in maniera da giungere alla situazione finale mostrata in figura nella quale l’angolo di *sx* è il minimo degli angoli maggiori di quello di P e quello di *dx* è il massimo degli angoli minori di quello di P. Questo caso sarà il **caso base** della nostra ricerca binaria.

sx

dx

P

β

Inizialmente i vertici *sx* e *dx* sono rispettivamente il primo e l’ultimo vertice del poligono. Il primo controllo da fare (solo alla prima chiamata) è che il poligono giaccia nel semipiano indicato dalle due frecce blu nel disegno sottostante.

Infatti, se così non fosse, il punto sarebbe certamente fuori dal poligono e non avrebbe senso iniziare con la ricerca. Si prosegue controllando se la differenza tra l’indice di sinistra e quello di destra sia pari ad uno, in tal caso ci troveremmo nel caso base sopra descritto. Se dovessimo essere nel caso base si controlla che il punto P sia nel semipiano indicato nella figura sottostante e, se anche questo controllo dà esito positivo, ritorniamo vero ed usciamo, altrimenti ritorniamo falso.

sx

dx

Se non ci troviamo nel caso base, calcoliamo l’indice del poligono denominato ***mid*** con la formula riportata nel codice. *Mid* sarà l’indice del vertice che avrà **angolo associato a metà tra quelli associati a sx e a dx**. Si controlla ora se il punto *P* ha distanza quasi nulla dalla retta che congiunge il vertice *0* al vertice *mid* (all’interno della striscia in figura). In caso affermativo, si controlla che il punto appartenga al semipiano mostrato in figura, e se supera anche questo test la funzione ritorna vero, altrimenti ritorna falso.

mid

sx

dx

0

Escluso questo caso limite, basta ora controllare se il punto si trova nel semipiano a destra o in quello a sinistra rispetto alla retta passante per gli estremi *0* e *mid* del poligono. Nel primo caso (in figura sottostante) si richiama la funzione con ***sx = mid***, nel secondo caso si richiama la funzione con **dx = mid**.

Sx=mid

dx

0

È facile vedere che stiamo operando proprio una ricerca binaria del punto *P* rispetto all’angolo α sopra descritto; questo è possibile solamente grazie alla proprietà di **convessità** del poligono. Vedremo nella prossima sezione come il costo computazionale di questa funzione la renda preferibile rispetto a controllare che il punto giaccia a sinistra di ogni lato (considerando il lato orientato secondo la normale) come in figura.

Nota bene: l’uso delle tolleranze è analogo a quanto discusso in precedenza per il *check punto del poligono interno alla sezione*. Inoltre, l’assenza dell’uso di tolleranze nel controllo che *mixed\_product* sia maggiore o minore di zero non deve stupire. Infatti, abbiamo già trattato il caso in cui *mixed\_product* sia circa zero quando abbiamo considerato che il punto stesse circa sulla retta congiungente *0* con *mid*.

**5.6 Considerazioni sulla complessità computazionale**

Lo scopo di questa sezione è quello di ricapitolare la complessità computazionale delle varie parti di cui si compone il metodo *ComputeFinalTest*, nonché giustificare in maniera più approfondita le scelte fatte. Nel seguito indicheremo con il numero di lati del poligono intersecante.

Per quanto riguarda la costruzione della sezione, si tratta di calcolare al più **quattro intersezioni** del tipo **retta-piano**. Veniamo ora alla complessità dei vari controlli successivi.

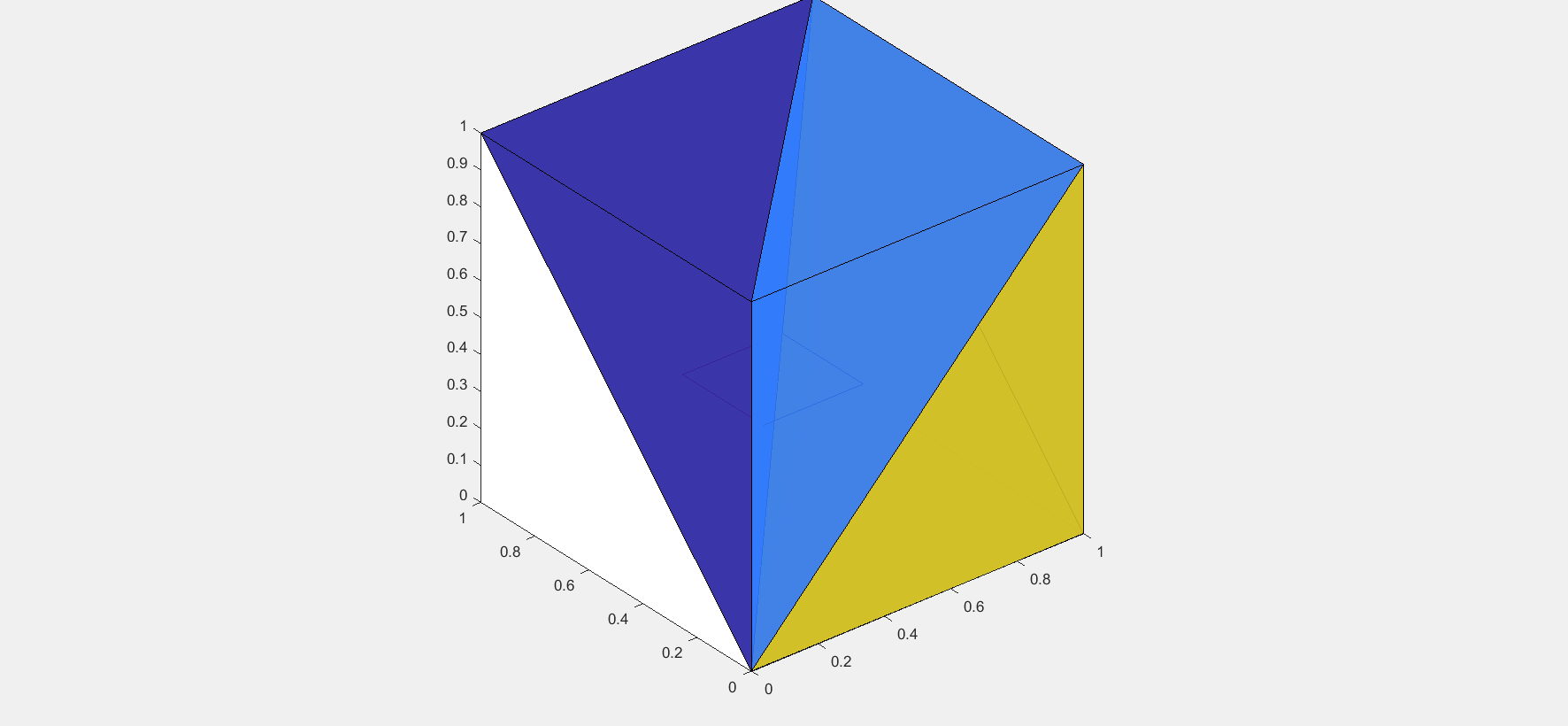
Il primo controllo che eseguiamo (punti della sezione interni al poligono) non a caso è anche il più efficiente. Asintoticamente, questa parte ha una complessità di **O(logn**) poiché, come già anticipato, la funzione *PointInPolygon* ha complessità logaritmica. Si è già fatto notare come questa funzione ricalchi passo passo l’algoritmo classico della *binary search*; da quest’osservazione segue quanto affermato sulla complessità temporale. Alternativamente, si sarebbe potuto scrivere una relazione di ricorrenza per il costo della funzione e giungere allo stesso risultato chiamando in causa il *master theorem.* Omettiamo i dettagli poiché, come è facile intuire, la relazione corrisponderebbe quasi perfettamente con quella della *binary search*.

L’algoritmo procede con il calcolo delle eventuali intersezioni fra i lati della sezione e quelli del poligono. Questa parte è nettamente la più onerosa: si tratta, infatti, di eseguire **O(n) intersezioni retta-retta**. Un’opzione alternativa era quella di anticipare l’ultimo controllo (punti del poligono interni alla sezione) al fine di effettuare le intersezioni solo nei casi in cui nessuna delle due figure avesse un punto interno all’altra. Nonostante in certi casi questa soluzione presenti dei vantaggi, al caso pessimo risulta meno performante di quella implementata. Infatti, in totale si effettuerebbero O(n) intersezioni e O(n) controlli del tipo “punto-interno-a-figura”. Invece, con l’attuale implementazione si eseguirebbe (oltre alle O(n) intersezioni) **un solo controllo del tipo “punto-interno-a-figura”**.

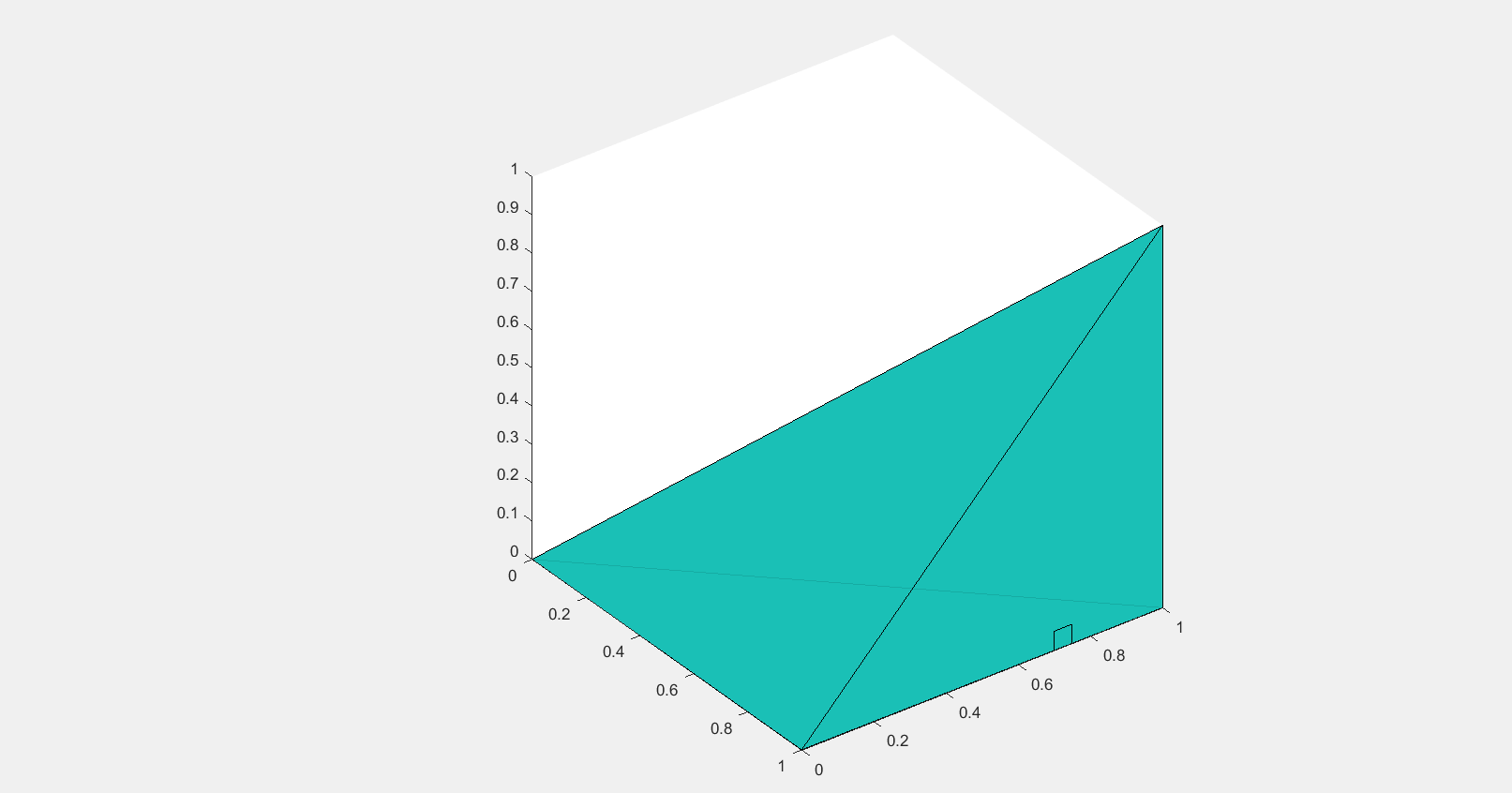
**6 Test e risultati finali**

Una volta conclusa la fase di progettazione e scrittura dell’algoritmo, abbiamo iniziato quella di valutazione e **correzione di bachi**. Per far ciò, abbiamo effettuato numerosi test cambiando i vertici in input, introducendo la *VERBOSITY* (compilazione condizionale) e utilizzando il *debugger*. Una volta che l’algoritmo è risultato funzionante, abbiamo inserito **le coordinate dei vertici del poligono in un file** e le abbiamo memorizzate dallo stesso file aperto in lettura. Infine, visto che inizialmente il file *matlab* non presentava i **comandi per il plot del poligono**, presentava solo quelli dei tetraedri intersecati, li abbiamo aggiunti in scrittura sul file *plotCellFound.m* generato. Di seguito si riportano alcuni dei risultati ottenuti tenendo presente che il poligono è quello di lati color nero.

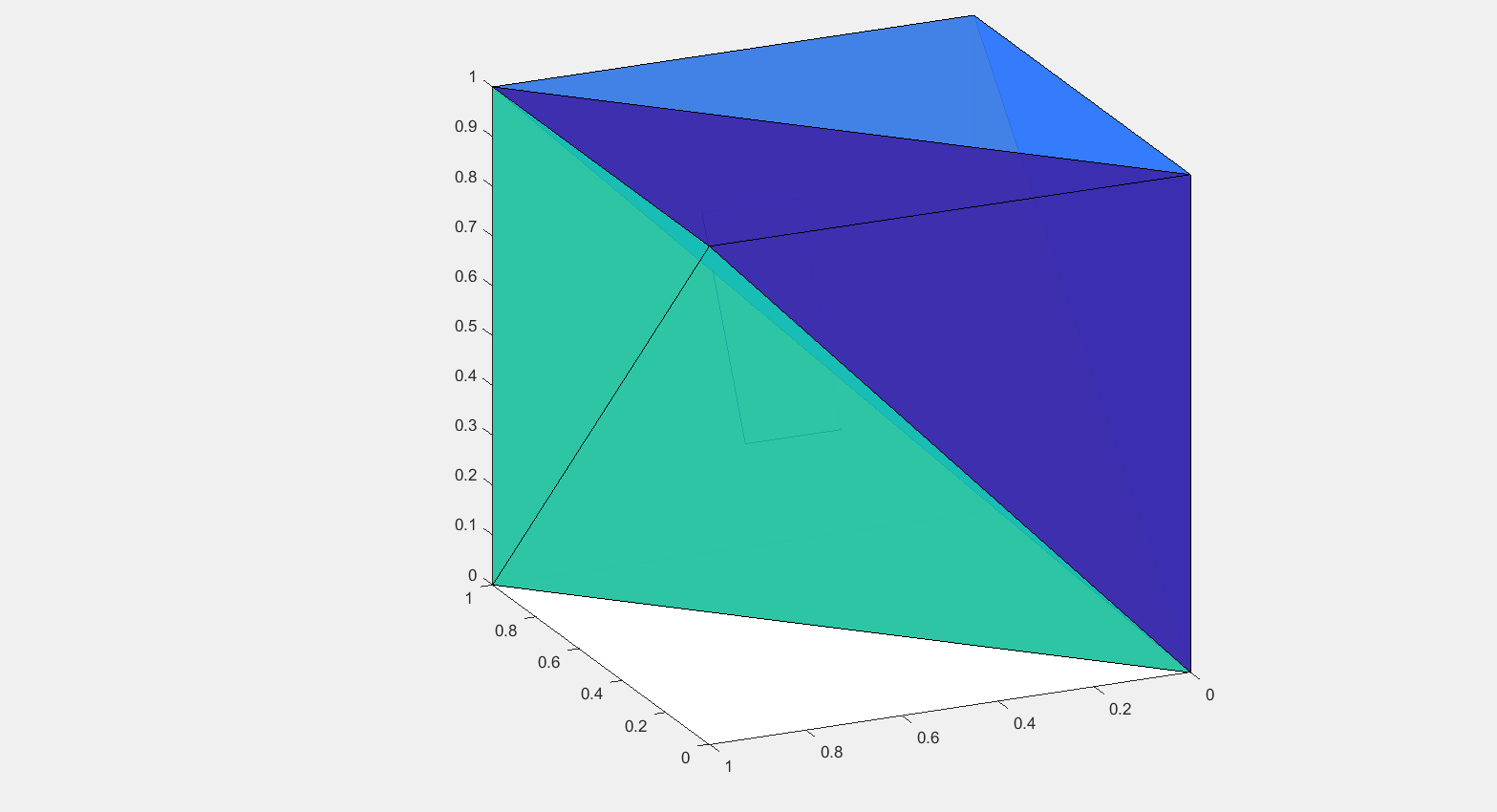
* Esempio 1 -> poligono di quattro lati sul piano parallelo a z=0.



* Esempio 2->poligono di quattro lati sul piano parallelo a x=0.



* Esempio 3->poligono di quattro lati non giacente su un piano parallelo agli assi.



Purtroppo, l’immagine fissa non rende bene il risultato in questo caso.

* Esempio 4->poligono di sei lati giacente su un piano parallelo a z=0.

