# DOCUMENTAZIONE PROGETTO PROGRAMMAZIONE E CALCOLO SCIENTIFICO

## RAFFINAMENTO MESH TRIANGOLARE: metodo complesso

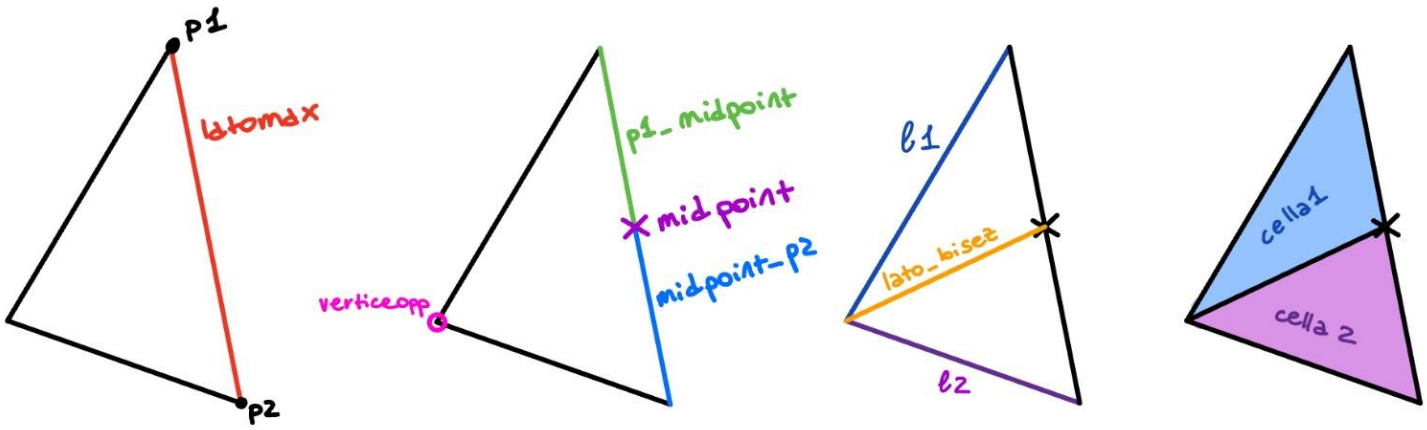
**Lorenzo de Gregorio, Andrea Grasso, Luca Mercuriali**

**Descrizione:**

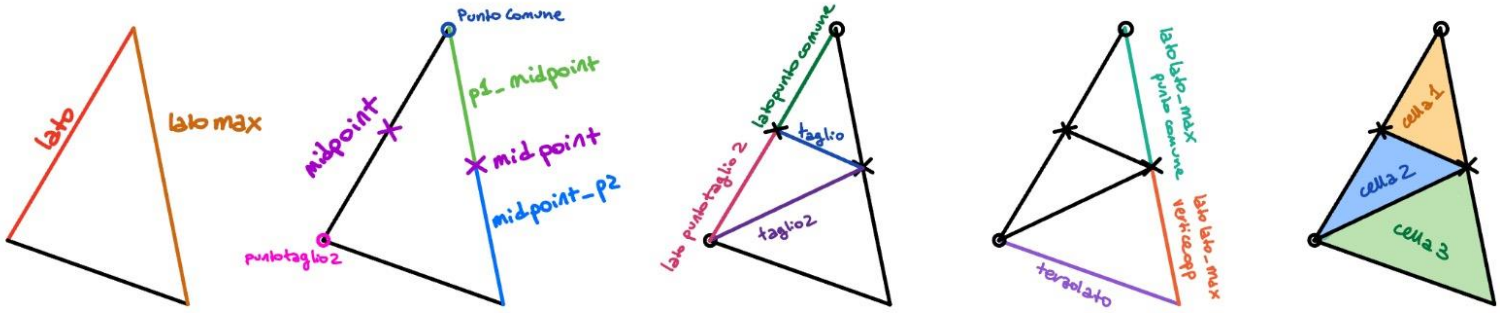
L'obiettivo del raffinamento è quello di generare una versione più dettagliata di una mesh triangolare bidimensionale, a patto che questa sia ammissibile. Perché una mesh sia considerata ammissibile, deve rispettare il seguente criterio: per ogni cella della mesh, le sue celle adiacenti possono condividere un lato completo o solo un vertice. Non è permesso che più di un lato completo o più di un vertice siano adiacenti a una cella.

L'algoritmo deve trovare il lato maggiore e dividerlo a metà. si inizia selezionando un lato qualsiasi del triangolo:

- Se il lato selezionato è il lato maggiore divido la cella in 2. Dichiaro l1 il lato che congiunge punto 1 al vertice opposto ed l2 il lato che congiunge il punto 2 al vertice opposto. Creo 2 celle e riempio la matrice di adiacenza delle due celle.



- Se il lato selezionato non  il lato maggiore divido la cella in tre parti:
  Prendo il punto medio sia del lato maggiore che del lato selezionato. Poi trovo il punto comune tra i due lati e il vertice opposto del lato maggiore.
  Eseguo poi due tagli che partono entrambi dal punto medio del lato maggiore e uno va a puntotaglio2 e l'altro a pm del lato selezionato. Chiamo poi i lati opportunamente per poi dare un ordine preciso alle celle che creo.  Avendo diviso il triangolo in tre parti devo creare le tre nuove celle generate dalla separazione e riempire quindi l'adiacenza opportunamente.




**STRUTTURA LOGICA**

Il programma è diviso in 5 file :

- "main_test": C++ Source file (.cpp) - contiene le istruzioni per lanciare tutti i test;
- "main_program": C++ Source file (.cpp) -  istruzioni per importare i dati dai file dati e dare inizio al processo di raffinamento della mesh;
- "empty_class": C++ Header file (.hpp) - Dichiarazione delle classi, costruttori e metodi/funzioni degli oggetti.
- "empty_class": C++ Source file (.cpp) - contiene l'implementazione dei costruttori, dei metodi e delle funzioni dichiarate in "empty_class.hpp";
- "test_empty": C++ Header file (.hpp) – implementazione dei test;

La struttura dei cinque file è la seguente

**MAIN_PROGRAM.cpp**

viene dichiarata una Macro che permette di includere e utilizzare quanto scritto nel file “empty_class.hpp”.

Nel blocco centrale del main imposto due criteri di arresto: il numero massimo di iterazioni (max_it) e una tolleranza sull’area (theta). Ad ogni iterazione viene trovata e selezionata la cella con area maggiore e viene poi raffinata. Una volta raffinata questa cella si continuano a raffinare anche le celle adiacenti affinché la mesh risulti nuovamente ammissibile. Quindi ad ogni iterazione non abbiamo scelto di raffinare solo un triangolo o di raffinare tutta la mesh ma il minimo insieme di triangoli adiacenti affinché la mesh risulti affine a fine iterazione.

Per aprire i file dei punti, lati e celle come prima cosa inizializzo delle variabili che tengono traccia delle dimensioni dei vettori  ( vector<Point>points)

Tramite poi la chiamata del metodo reserve() del vettore points garantisco che il vettore abbia una capacità sufficiente per ospitare un certo numero di elementi, senza necessariamente aggiungere effettivamente quegli elementi al vettore. dunque “reserve()” alloca spazio per ospitare “points.size()+max_it” elementi senza che sia necessario fare riallocazioni frequenti di memoria. Questo è possibile farlo in questo caso perché conosciamo il numero approssimativo di elementi che si prevede di aggiungere al vettore, migliorando così le prestazioni.

Posso poi importare il file: “Cell0Ds.csv” scegliendo il percorso giusto.

Inizializzo poi il massimo identificatore di punti ponendolo uguale alla dimensione del vettore punti.

Ripeto questo procedimento anche per il vettore segmenti e celle modificando opportunamente le righe.

Nel caso dei segmenti lo spazio da allocare è edges.size()+4*max_it. mentre per le celle è di triangles.size()+3*max_it

Stampo l’area della cella maggiore e il numero di celle prima del raffinamento. Faccio il raffinamento usando le funzioni: MaxCelle, LatoMaggiore e RaffinaCella controllo poi i criteri di arresto e stampo i file di output su file “ .csv”

**EMPTY_CLASS.hpp**

" .hpp" vuol dire che questo file sta nell'header ovvero questo file sarà utilizzato per la definizione di metodi e funzioni.

Ho 9 righe di macro.

#ifndef_EMPTY_H è una direttiva che indica al preprocessore la compilazione condizionata. Testa la condizione, se risulta verificata, viene incluso per la compilazione il codice della riga successiva (#define_EMPTY_H) fino all'ultima riga (#endif)

#include<Eigen/Eigen> il nomefile (Eigen/Eigen) viene ricercato in un insieme di directory standard definite dall'implementazione. Un altro modo per poterlo fare era scrivere #include "Eigen/Eigen", in questo modo il file sarebbe stato ricercato nella directory corrente e poi se non trovato, la ricerca continuava nelle directory standard.

Dichiaro la classe Point e gli passo due attributi public (possono quindi essere accessibili a tutti): double x (ascissa), double y (ordinata).

Con la riga di codice Point(double& asc, double& ord) creo un costruttore non vuoto in cui dichiaro due double (asc, ord) e dento il costruttore gli assegno il valore di x e di y rispettivamente. la "&" è un operatore di indirizzo che opera su una variabile e restituisce l'indirizzo della variabile. Quando il costruttore viene chiuso esso si distrugge automaticamente.

Facciamo poi stesso procedimento per la classe Segment e Cell.

Nella classe Segment passo, come public, gli attributi: int punto1, int punto2, double lunghezza, int p1_midpoint=-1, int midpoint_p1=-1, int midpoint=-1, int cella1=-1, int cella2=-1. Questi ultimi vengono posti tutti = -1 perché all'inizio non esistono ma verranno creati successivamente, gli assegno quindi un valore di default iniziale.

Nella riga di codice:

friend bool operator>(const Segment& lato1, const Segment& lato2)

il metodo friend è definito fuori dalla classe ma può accedere a tutti i membri public e protected della classe. In questo caso dico che è amico dell'operatore booleano il quale serve per verificare se lato1 è più lungo di lato2.

Nella classe Cell passo come public, gli attributi: vector<int>points, vector<int>edges (vettori dei punti e dei lati), double area e bool flag (operatore booleano che servirà poi per distruggere la cella)

Chiamo poi la funzione Cell e aggiungo (tramite push_back) i tre punti al vettore points e i tre lati al vettore edges e poi pongo l'operatore booleano= vero e calcolo l'area del triangolo.

Dichiaro poi metodi per trovare il lato maggiore, creare il punto medio del lato massimo, trovare il vertice opposto al lato massimo e trovare il vertice comune tra due lati

per esempio la funzione TrovaLatoMaggiore:

inline int TrovaLatoMaggiore(Cell& cella, std::vector& univedges) {

la riga definisce la funzione TrovaLatoMaggiore:
Il modificatore inline dichiara che la funzione è semplice e il corpo della funzione può sostituire la funzione stessa. vengono passati tramite referenza la cella selezionata e il vettore dei lati.

i. se il lato 0 è più grande del lato 1 e se il lato 1 è più lungo del lato 2→ latomax (lato_maggiore)= lato 0
ii. altrimenti se il lato 1 è maggiore del lato 2 → latomax=lato 1
iii. altrimenti (→lato 2 più grandi lato 0 e di lato 1)→ latomax=lato2

Definisco la funzione per creare il PUNTO_MEDIO:
calcolo il punto medio su x e il punto medio su y e aggiungo il punto medio al vettore dei punti
DIFFERENZA NELLA DICHIARAZIONE DELLE DUE FUNZIONI
inline int TrovaLatoMaggiore
inline void CreaPuntoMedio
La prima funzione deve restituire il lato maggiore (return lato_max), la seconda invece non deve restituire niente.
Dichiaro la funzione per trovare VERTICE_OPPOSTO:
prendo in input la cella e il lato, inizializzo a 0 un contatore e prendo come punto il punto della cella con quel contatore. uso ciclo while (→ finchè il punto = al primo punto del lato oppure il punto = al secondo punto del lato (ovvero se io prendo un vertice e quel vertice è uno dei due vertici del lato selezionato)) aumento di 1 il contatore e pongo di nuovo il punto come punto della cella con quel contatore.
dovendo restituire un valore se non trovo il vertice scrivo " return -1;"
Dichiaro la funzione per trovare VERTICE_COMUNE:
prendo in input i primi due lati e inizializzo come intero un punto (vertice comune).

i. se il primo punto del lato 1 = al primo punto del lato 2 → il vertice comune è il punto 1.
ii. se il primo punto del lato 1 = secondo punto del lato 2 → il vertice comune è il punto 1.
iii. altrimenti il punto comune è il punto 2.

Chiamo le funzioni che importano i punti, i segmenti e le celle, la funzione DividiCella e quella RaffinaCella (che saranno poi definita nel "empty_class.cpp")

**EMPTY_CLASS.cpp**

" .cpp" vuol dire che è in Sources (serve per l'implementazione) ci sono quindi le funzioni che vengono implementate. Si separano " .hpp" e " .cpp" per permettere di nascondere l'implementazione e di tenere nell'hpp solo le definizioni di funzioni.

1. Dichiaro le funzioni per importare gli oggetti:
    a. dichiaro la funzione per importare i PUNTI: essa prende due parametri, nomefile (rappresenta il nome del file da cui si intende importare dei dati ed è di tipo string) e points (di tipo vettore). Quest'ultimo è una referenza ad un vettore di oggetti. Nella riga
    istringstream conv(line);
    viene creata una variabile conv di tipo istringstream che è un tipo di stream di input basato su stringa. E' usato per estrarre dati dalla riga letta in precedenza.
    Nella riga
    conv >> id >> marker >> x >> y;
    estraggo dati dalla stinga conv, che rappresenta la riga letta del file e li memorizzo nelle variabili 'id', 'marker', 'x', 'y'.
    poi aggiungo le nuove coordinate al vettore point e chiudo il file
    b. dichiaro e sviluppo nello stesso modo la funzione per importare i SEGMENTI
    c. dichiaro e sviluppo nello stesso modo la funzione per importare le CELLE. Poi costruisco le adiacenze (se il lato 1,2,3 ha cella1 o cella2= - 1 metto l'id di cella in cella1 o cella2)

2. dichiaro la funzione DIVIDICELLA:

a. Prima di tutto controllo, se il lato è già stato diviso uccido la cella, altrimenti la divido.
    Se il puntatore dal primo vertice al punto medio oppure se il puntatore dal secondo vertice al punto medio o se il puntatore al punto medio sono minori di zero
    ➔ creo il punto medio, aggiungo 1 al massimo ID dei punti e creo nuovi lati, che saranno quelli dal primo vertice al punto medio e dal secondo vertice al punto medio;
    poi aggiungo 2 al contatore dei segmenti.
b. Dichiaro come intero il vertice opposto usando la funzione VerticeOpposto. Questa funzione agisce sulla base della distinzione in due casi illustrata precedentemente: uno è il caso in cui il lato selezionato sia il lato maggiore del triangolo, l'altro è il caso in cui non sia il lato maggiore.

3. dichiaro la funzione RAFFINA_CELLA:
   trovo il lato maggiore tramite la funzione TrovaLatoMaggiore, poi cerco la cella successiva. Se la tra i lati della cella selezionata non c'è il lato più lungo della prima cella→ la cella successiva è la prima cella, altrimenti è la seconda
   chiamo la funzione DividiCella poi controllo se CellaDopo>-1 (vuol dire che l'ho trovato nel passaggio precedente), se il lato maggiore corrisponde con quello trovato tramite la funzione TrovaLatoMaggiore allora chiamo la funzione DividiCella. altrimenti chiamo la funzione RaffinaCella.

4. dichiaro la funzione TROVACELLAAREAMAX: in cui certo e restituisco il triangolo con l'area maggiore.

**TEST_EMPTY.hpp**
Vengono implementati i test.
Si devono testare 9 test:
I primi tre test (TestImportPoints, TestImportSegments, TestImportCells) verificano che le funzioni di importazione leggano ccorrettamente i dati da file ' .csv' e riempiano i vettori.
I test successivi (TestCreaPuntoMedio, TestTrovaVerticeOpposto, TestTrovaLatoMaggiore, TestTrovaVerticeComune) verificano il corretto di alcune funzioni geometrich
Gli ultimi due test (TestDividiCellaPerTrovaLatoMaggiore, TestDividiCellaPerLatoQualsiasi) verificano la divisione di una cella in base a un lato specifico o a un lato qualsiasi. Questi test includono anche la verifica dell'area delle celle risultanti.
Nel test: TestDividiCellaPerTrovaLatoMaggiore
viene creato un vettore 'ascisse' contenente tre valori: 0.0, 1.0, 0.5 e un vettore 'ordinate' con i valori 0.0, 1.0, 1.0. Essi rappresentano le coordinate x e y dei punti
Viene creato un vettore 'poinnts' che contiene gli oggetti della classe 'Point'.
Vengono inizializzati tre punti ('punto1', 'punto2', 'punto3') utilizzando le coordinate dei vettori 'ascisse' e 'ordinate', quindi vengono inseriti nel vettore 'points'.
Vengono definiti tre identificatori 'p1_id', 'p2_id', 'p3_id' per rappresentare gli ID dei punti
Vengono creati tre oggetti della classe Segment ('lato1', 'lato2', 'lato3') che rappresentano i segmenti tra i punti. Questi segmenti sono costruiti utilizzando gli ID dei punti e il vettore 'points'
Viene creato un vettore 'edges' e vengono inseriti al suo interno i tre segmenti creati ('lato1', 'lato2', 'lato3')
Vengono definiti tre identificatori 'l1_id', 'l2_id', 'l3_id' per rappresentare glii ID dei segmenti.

Viene creato un oggetto della classe Cell denominato ‘triangolo’ rappresentante una cella con tre punti (p1_id’, ‘p2_id’, ‘p3_id’) e tre segmenti (‘l1_id’, ‘l2_id’, ‘l3_id). Questo oggetto rappresenta un triangolo e viene inserito in un vettore chiamato ‘triangles’
Viene definito un identificatore ‘c1_id’ per rappresentare l’ID della cella
Vengono definiti tre identificatori ‘IDP’, IDE’, ‘IDC’ utilizzati successivamente come parametri della funzione ‘DividiCella’
Viene invocata la funzione DividiCella’ con i parametri ‘c1_id’ e ‘latomax’ per dividere il triangolo. Questa funzione modifica il vettore’ triangles’, creando nuove celle risultanti dalla divisione
Viene definito un vettore ‘resultID’ contenente gli ID previsti per le nuove celle risultanti dalla divisione, e un vettore ‘resultAREA’ contenente le aree previste per le nuove celle.
Viene eseguito un ciclo ‘for’ che verifica se le nuove celle del vettore ‘triangles’ hanno indicatore ‘flag’ impostato su ‘true’. Se lo sono vengono confrontati gli ID e le aree dele ceelle con i valori previsti nei vettori ‘resultID’ e ‘resultAREA’.

**Scelta della struttura dati e degli algoritmi e import dei dati**
I dati della mesh sono forniti da tre file .csv in input contenenti rispettivamente la tabella dei punti, la tabella dei lati e la tabella delle celle.

- Di ogni punto è fornito il Marker (intero positivo), l’Id (intero positivo univoco all’interno dei punti), l’ascissa X (numero reale), l’ordinata Y (numero reale).
- Di ogni lato è fornito il Marker (intero positivo), l’Id (intero positivo univoco all’interno dei segmenti), l’Id del punto di origine del lato (pensato come vettore), l’Id del punto di fine del lato (pensato come vettore).
- Di ogni cella è fornito l’Id (intero positivo univoco all’interno delle celle), gli Id dei tre vertici della cella, gli Id dei tre lati della cella.

La classe Segment ha come attributi gli Id dei suoi due punti estremi, del suo punto medio (inizializzato al valore -1 che significa che il lato non è stato ancora diviso), gli Id dei suoi due mezzi lati (inizializzati al valore -1, che significa che il lato non è stato ancora diviso), gli Id delle due celle adiacenti per quel lato (inizializzati al valore -1).

La classe Cell ha come attributi un vettore contenente i tre Id dei suoi vertici, un vettore contenente i tre Id dei suoi lati e un Booleano (flag) che indica se la cella è stata già raffinata o non ancora.

Una volta importati i dati della mesh di partenza, memorizziamo per ogni classe il primo valore Id non ancora utilizzato, sfruttando il fatto che gli Id in input vanno ordinatamente da 0 al numero totale degli elementi -1.

Una volta costruita una cella, si rintracciano i suoi lati grazie all'Id, in modo da agevolare la ricerca delle celle adiacenti con un costo computazionale O(1) di accesso a un vettore per posizione.

Abbiamo salvato gli oggetti in un vettore std::vector<Object>. L'utilizzo di un vettore ha il vantaggio di permettere l'accesso a un suo elemento in O(1) se si conosce la sua posizione, che può essere uguale al suo Id se si usa un vettore non ordinato. Di contro, i vettori sfruttano un'allocazione della memoria variabile, attraverso l'utilizzo dei puntatori, talvolta poco affidabili.

Abbiamo scartato l'idea di utilizzare un array in quanto risulta difficile prevedere quanta memoria allocare per l'array, e si può ovviare al problema dei puntatori utilizzando la posizione dell'elemento nel vector, sfruttando il fatto che questa corrisponde esattamente all'Id univoco. Abbiamo anche scartato l'uso di una mappa in quanto l'area delle celle non è una potenziale chiave univoca.

Abbiamo deciso di calcolare il massimo dell'area delle celle esistenti a ogni iterazione al posto di ordinare inizialmente le celle per area e mantenere in seguito un inserimento ordinato, anche se ciò comporta un costo computazionale più alto.

Il costruttore degli oggetti è direttamente nell'argomento del push_back perché utilizzato dentro a delle funzioni; Creiamo quindi una variabile globale permanente. Se si chiamasse il costruttore all'interno della funzione creerebbe una variabile locale che smetterebbe di esistere una volta terminata la chiamata della funzione, finendo per perdere quello che andiamo ad aggiungere al vettore tramite push_back.

**Analisi del costo computazionale e dell'utilizzo della memoria**

Il programma è principalmente caratterizzato dal calcolo del costo computazionale associato all'individuazione della cella con l'area massima tra tutte le celle presenti al momento. Questa operazione si ripete ad ogni iterazione. Per k = numero di raffinamenti (iterazioni), il costo dell'algoritmo è:

$$\sum_{i=0}^{k} O\,(n_i)$$

($n_i$ numero di celle esistenti alla fine della (i-1)-esima iterazione)

Il numero di celle create ad ogni iterazione può essere pensato come una variabile aleatoria $\theta(\mu,\sigma^2)$ perché non dipende dalle celle iniziali, attuali o dal numero di iterazioni. Esse dipendono solamente dall'insieme di celle che bisogna prendere ad ogni iterazione affinché la mesh torni ad essere ammissibile.

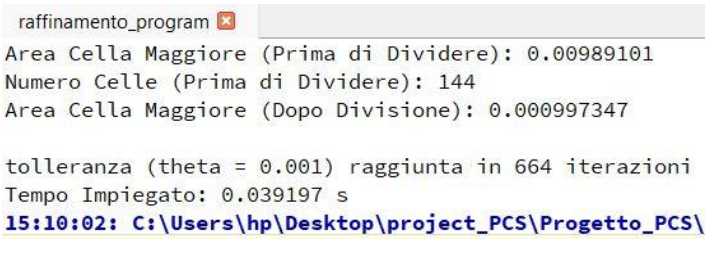
$\sum_{i=0}^{k} O\,(n_i) \approx \sum_{i=0}^{k} O\,(n_1 + (i-1)\mu) = O\,(kn_1) + O\left(\frac{k(k-1)}{2}\,\mu\right) \approx O(kn_1) + O(k^2)$

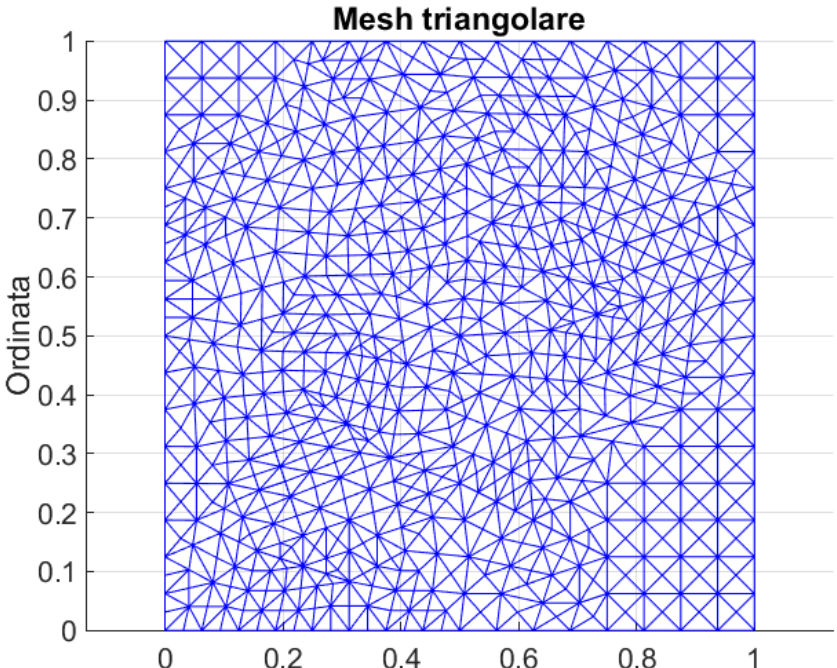
**RISULTATI:**

nel main_program.cpp abbiamo imposto due criteri di arresto. un numero di interazioni massime, max_it, e un valore theta di tolleranza sull'area.
modificando questi due parametri otteniamo mesh sempre più fini (aumentando max_it e diminuendo theta) oppure più grossolane (diminuendo max_it e aumentando theta).
Qui sotto mostriamo delle immagini delle mesh finali in base a questi due criteri d'arresto

theta=10^-3

max_it=800



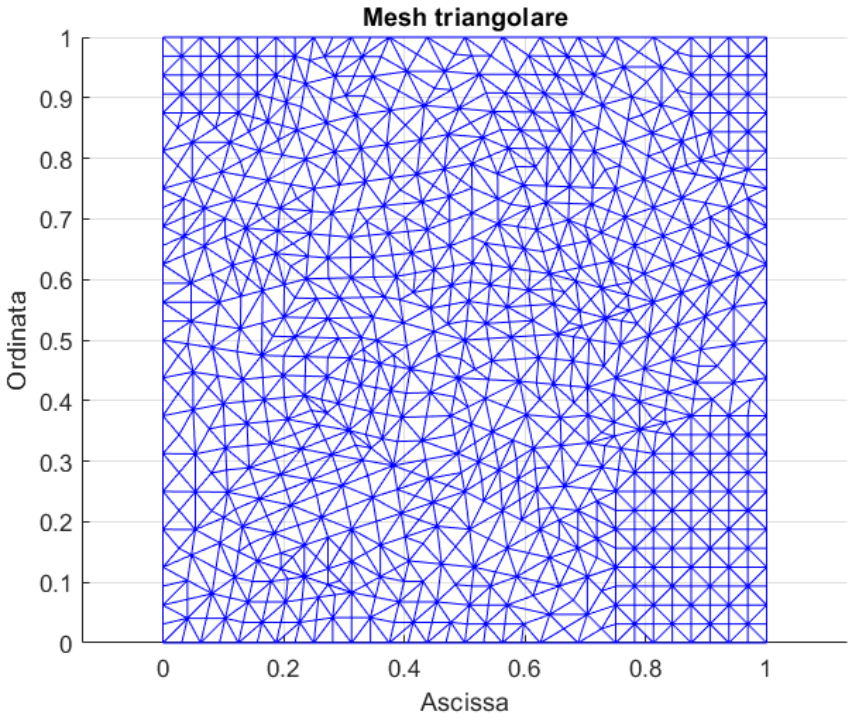
theta=10^-4

max_it=800

raffinamento_program

```
12:14:51: Starting C:\Users\hp\Desktop\project_PCS\
Area Cella Maggiore (Prima di Dividere): 0.00989101
Numero celle (Prima di Dividere): 144
Area Cella Maggiore (Dopo Divisione): 0.000969008

raggiunto limite massimo di iterazioni (800),
12:14:55: C:\Users\hp\Desktop\project_PCS\Progetto_
```




theta=10^-4

max_it=2000

raffinamento_program

```
12:16:42: Starting C:\Users\hp\Desktop\project_PCS\
Area Cella Maggiore (Prima di Dividere): 0.00989101
Numero celle (Prima di Dividere): 144
Area Cella Maggiore (Dopo Divisione): 0.000366211

raggiunto limite massimo di iterazioni (2000),
12:16:44: C:\Users\hp\Desktop\project_PCS\Progetto_
```