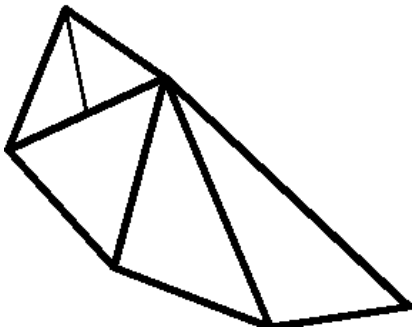
Documentazione Progetto “Raffinamento complesso”
corso di Programmazione e calcolo scientifico

a cura di Gianusso Fabio, Giussani Marta, Scibilia Enrico

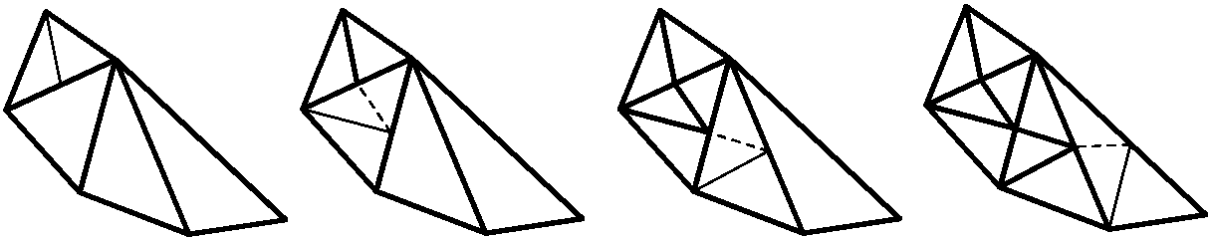
Descrizione del problema e algoritmo risolutivo

Il Raffinamento complesso richiede, data una mesh triangolare bidimensionale ammissibile, di restituire una mesh triangolare bidimensionale ammissibile più fine. Il criterio di ammissibilità è il seguente: presa una qualsiasi cella della mesh, le sue celle adiacenti possono esserle adiacenti soltanto per un lato intero o un solo vertice; non è quindi ad esempio ammissibile una mesh che presenti questa configurazione:



Esistono diversi modi per raffinare un triangolo, viene richiesto di implementare un algoritmo del tipo “bisezione del lato più lungo”; nel caso complesso si parte da un triangolo bisezionando il suo lato maggiore per il punto medio. La cella adiacente al lato maggiore del triangolo appena raffinato deve venire a sua volta raffinata seguendo il seguente schema:

A) se il lato maggiore della cella precedentemente raffinata è anche il lato maggiore della cella attuale, si biseziona la cella attuale dal punto medio del lato maggiore al vertice opposto. A questo punto la mesh è ritornata ammissibile e ci si ferma;
B) altrimenti in primo luogo si cerca il lato maggiore della cella attuale e la si biseziona dal punto medio al vertice opposto, poi si procede a unire il punto medio del lato maggiore della cella precedentemente raffinata con il punto medio del lato maggiore della cella attuale. A questo punto la mesh non è ancora ammissibile e si procede ripetendo lo stesso schema con la cella adiacente al lato maggiore della cella attuale appena raffinata.



L’algoritmo presenta quindi un carattere ricorsivo che ne facilita l’implementazione. Si osservi inoltre che la mesh generata seguendo questo algoritmo è necessariamente ammissibile.

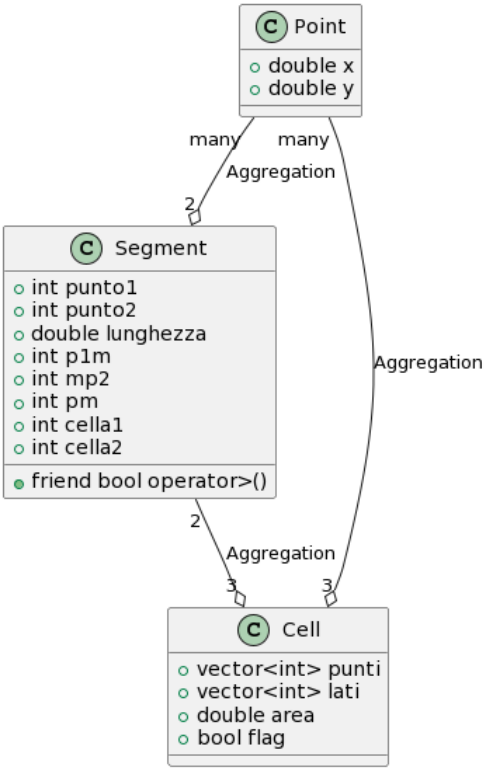
Scelta del paradigma di programmazione e della struttura logica

Il codice del programma è separato su un totale di cinque files:

1) “main_program”: C++ Source file (.cpp) - contiene le istruzioni per dare inizio all’import dati e al processo di raffinamento della mesh;
2) “empty_class”: C++ Header file (.hpp) - contiene la dichiarazione delle classi di oggetti e i relativi costruttori e metodi o funzioni della libreria ProjectLibrary;
3) “empty_class”: C++ Source file (.cpp) - contiene l’implementazione dei costruttori, dei metodi e delle funzioni dichiarate nell’Header “empty_class”;
4) “main_test”: C++ Source file (.cpp) - contiene le istruzioni per dare il run a tutti i test;
5) “test_empty”: C++ Header file (.hpp) - contiene l’implementazione dei test;

Il paradigma di programmazione che abbiamo scelto è la programmazione a oggetti, in quanto ci permette di scrivere un codice leggibile e facile da interpretare, inoltre permette una maggiore astrazione rispetto alla programmazione procedurale. È stata inoltre di fondamentale utilità la possibilità di avere degli oggetti come attributo di una classe e la possibilità di ridefinire alcuni operatori.

Di seguito il diagramma delle classi e delle relazioni tra oggetti che sintetizza la logica del nostro programma.



Abbiamo scelto la relazione di aggregazione tra lati e celle in quanto manteniamo i lati anche quando eliminiamo virtualmente, impostando al valore ‘False’ il flag, un triangolo. Abbiamo

scelto la relazione di aggregazione tra punti e celle in quanto anche la classe lati contiene dei punti come attributi.

Scelta della struttura dati e degli algoritmi

L'import dei dati

I dati della mesh in input sono forniti sotto forma di tre file csv 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.

Per salvare i dati in input abbiamo tre funzioni della libreria ProjecLibrary, che salvano gli oggetti in un vettore di oggetti std::vector<Object> a seconda della classe; la scelta della struttura dati è motivata dalla seguente tabella.

STRUCT DATI	**VANTAGGI**	**SVANTAGGI**
std::array	• accesso in O(1) se conosco la posizione; • allocazione della memoria definitiva, puntatori affidabili; • posizione=Id se non ordino il vettore;	• la dimensione è fissa e assegnata a compile-time;
std::vector	• accesso in O(1) se conosco la posizione; • posizione=Id se non ordino il vettore;	• allocazione della memoria variabile, puntatori inaffidabili;
std::map	• allocazione della memoria definitiva, puntatori affidabili;	• accesso in O(logn) se ordinata e si conosce posizione; • accesso in O(n) se non ordinata e si conosce la posizione;
std::list	• allocazione della memoria definitiva, puntatori affidabili;	• accesso in O(logn) se ordinata e si conosce posizione; • accesso in O(n) se non ordinata e si conosce la

		posizione;

Sarebbe risultato complesso creare una struttura dati che permettesse:

1) accesso in tempo costante O(1) una volta nota la posizione (per Id o per ordine);
2) accesso con puntatore all'oggetto senza che un'eventuale riallocazione della memoria rendesse vano il puntatore;
3) implementazione di un algoritmo di ordinamento per area svolto soltanto sulle celle della mesh iniziale, con inserimento ordinato di una nuova cella con costo O(logn).

Abbiamo preferito un vector a 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 vantaggio che essa corrisponde esattamente all'Id univoco nell'oggetto, anche per come abbiamo assegnato un Id ai nuovi oggetti creati dal raffinamento.
Abbiamo preferito un vector a una mappa in quanto l'area delle celle non è una potenziale chiave univoca, quindi l'unico vantaggio delle mappe rispetto ai vector che avremmo utilizzato sarebbe stato quello del puntatore, che abbiamo già risolto.
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; ciò comporta un costo computazionale più alto, abbiamo optato per una soluzione meno complicata da realizzare.

Il main dà inizio all'import dei dati creando tre vettori: un vettore di oggetti di tipo Point, un vettore di oggetti di tipo Segment, un vettore di oggetti di tipo Cell; che riempie chiamando le funzioni di import. I tre metodi sono tutti funzioni void che prendono in input il percorso del file d'interesse e il rispettivo vettore da riempire passato per referenza; si usa un oggetto di tipo istringstream.
Il costruttore degli oggetti è direttamente nell'argomento del push_back perché il push_back è utilizzato dentro a delle funzioni; se si chiamasse il costruttore all'interno della funzione creerebbe una variabile locale, che smetterebbe di esistere una volta terminata la chiamata della funzione. Quindi quello che andiamo ad aggiungere al vettore tramite push_back verrebbe immediatamente perso. In questo modo invece noi creiamo una variabile globale che non cessa di esistere al termine della chiamata della funzione.
Una volta costruita una cella, di cui sono forniti gli Id dei lati, si rintracciano (grazie alla corrispondenza dell'Id con la posizione nel vettore) i lati per mettere l'Id della cella come attributo di ogni suo lato, in modo da agevolare la ricerca delle celle adiacenti nel programma, con un costo computazionale O(1) di accesso a un vettore per posizione.

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, un Booleano 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, possiamo quindi trovare semplicemente la dimensione del vettore in cui gli elementi sono salvati), ciò ci tornerà utile in seguito.

Il Raffinamento

Apriamo un ciclo for sul numero totale di iterazioni fondamentali del processo di raffinamento; a ogni iterazione viene trovata la cella di area massima tra tutte le celle aggiornate, incluse le nuove celle create all'iterazione precedente; ne viene trovato il lato maggiore e su di lei viene chiamata la funzione RaffinaCella.

Di seguito lo pseudocodice della funzione ricorsiva RaffinaCella, che riproduce l'algoritmo del Raffinamento complesso partendo da una cella e da un suo lato, finché la mesh non è ammissibile.

```
// PSEUDOCODICE DIVIDICELLA //
// data cella e un lato appartenente a cella:
DividiCella(cella, lato) {
      if (lato == cella.lato_massimo) {
            crea nuovo segmento dal
            punto medio di lato al
            vertice opposto a lato;
            crea le 2 nuove celle;
      }
      else {
            crea nuovo segmento dal punto
            medio di cella.lato_massimo
            al vertice opposto a cella.lato_massimo;
            crea nuovo segmento dal punto
            medio di lato al punto medio di
            cella.lato_massimo;
            crea le 3 nuove celle;
      }
}


// PSEUDOCODICE PER RAFFINACELLA //
// data cella e un lato appartenente a cella:
RaffinaCella(cella, lato) {
      cella_successiva = cella.adiacente_latomax;
      DividiCella(cella, lato),
```

```
        if (cella_successiva.lato_massimo == cella.lato_massimo) {
            DividiCella(cella_successiva, lato);
        }
        else {
            RaffinaCella(cella_successiva, lato);
        }
}
```

Analisi del costo computazionale e dell'utilizzo della memoria

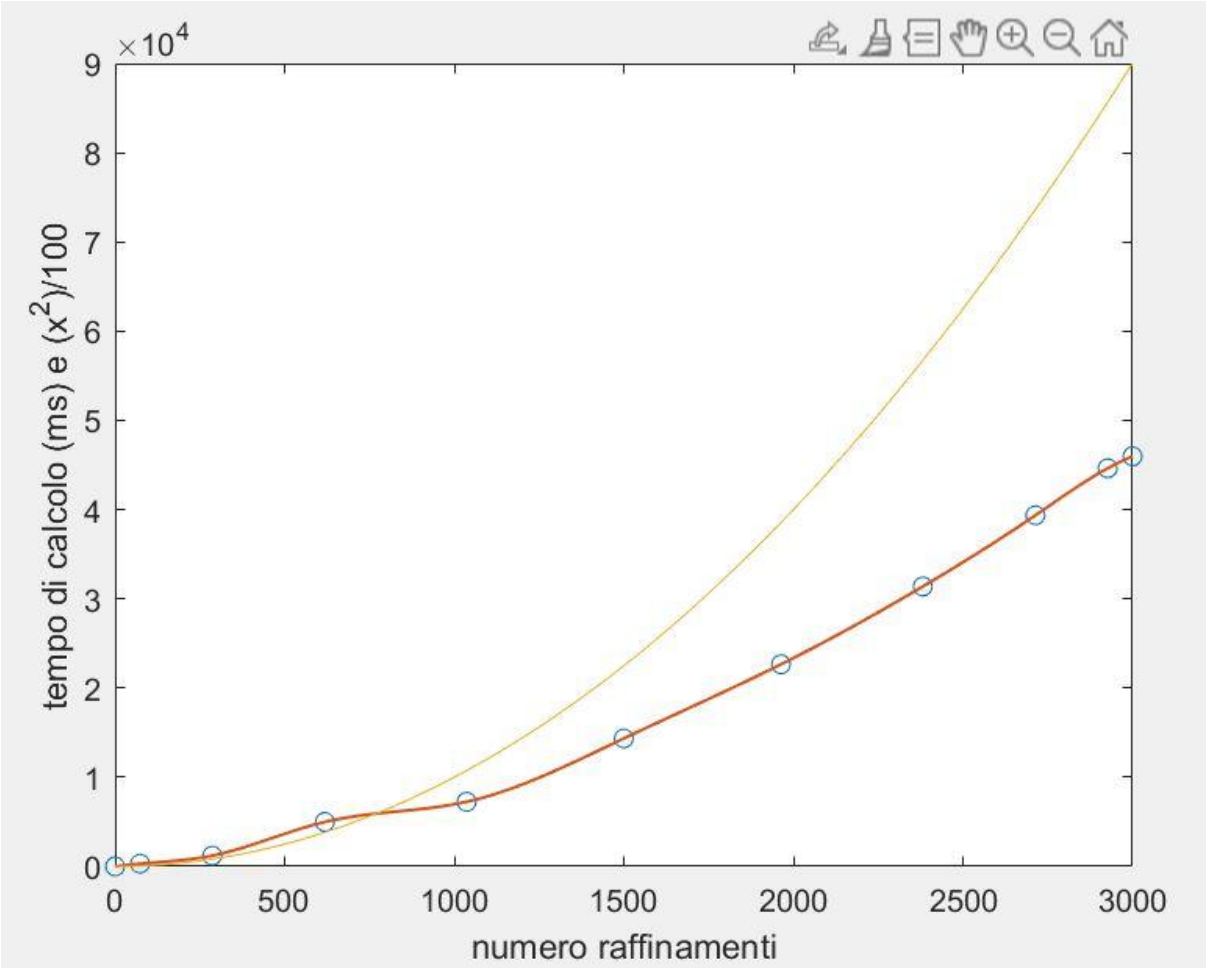
Il costo computazionale del programma è dato fondamentalmente dalla ricerca della cella di area massima tra tutte le celle attualmente esistenti, che si ripete a ogni iterazione fondamentale dell'algoritmo.
Per k = numero di raffinamenti (iterazioni), il costo dell'algoritmo è:
$\sum_{i=1}^{k} O(n_i)$ dove $n_i = \#celle\ esistenti\ alla\ fine\ dell'iterazione\ (i-1)-esima$
Il numero di celle create a ogni iterazione può essere pensato come una variabile aleatoria $\Theta(\mu, \sigma^2)$ che non dipende né dal numero di celle iniziali, né dal numero di celle attuali, né dal numero di iterazioni svolte. Chiamiamo μ la media della variabile aleatoria, allora abbiamo che
$E[n]_i = n_{i-1} + \mu = n_1 + (i-1)\mu$, conseguentemente si ha che in media
$\sum_{i=1}^{k} O(n_i) \approx \sum_{i=1}^{k} O(n_1 + (i-1)\mu) = O(kn_1) + O(\frac{k(k-1)}{2}\mu) \approx O(kn_1) + O(k^2)$

Scelta e implementazione dei test

Tutte le funzioni presenti nel programma, a eccezione di RaffinaCella, sono state testate atomicamente, in modo da poter individuare eventuali problemi anche quando in una funzione sono chiamate più funzioni diverse. Per testare l'import dati abbiamo mantenuto in un file la prima riga di ogni file dal dataset Test1 fornito. Per testare la funzione DividiCella abbiamo testato sia la divisione della cella in due che in tre.

Prestazioni sperimentali e risultati ottenuti

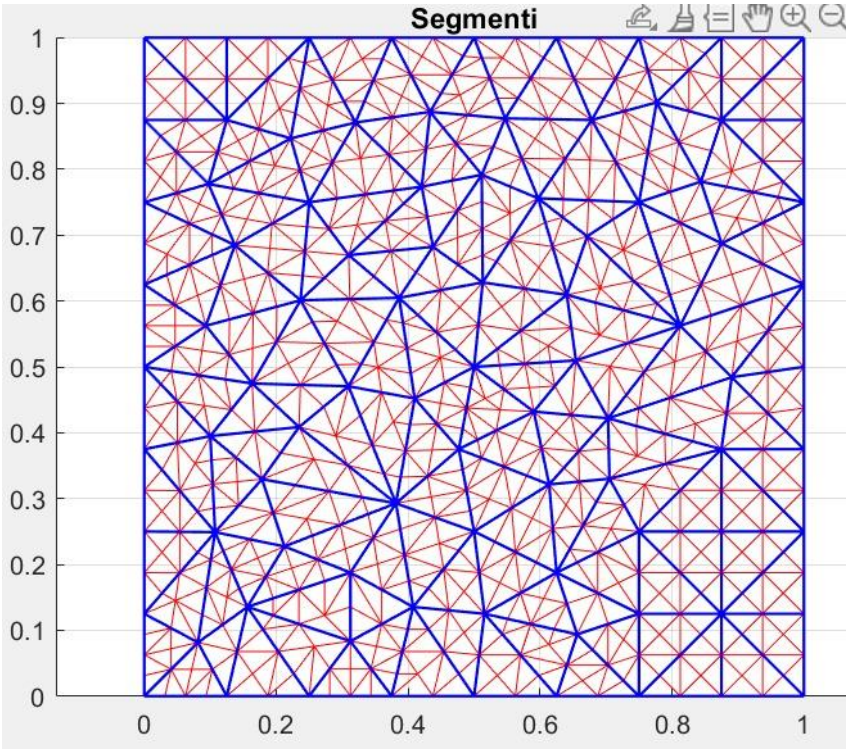
La trattazione che segue fa riferimento a risultati ottenuti su Test1. Assegnato un numero di iterazioni fondamentali (dove una iterazione prende l'attuale cella di area massima e inizia il processo di raffinamento fino a ottenere una mesh ammissibile), analizziamo sperimentalmente la relazione tra numero di iterazioni e tempo di esecuzione in modalità *Release*,



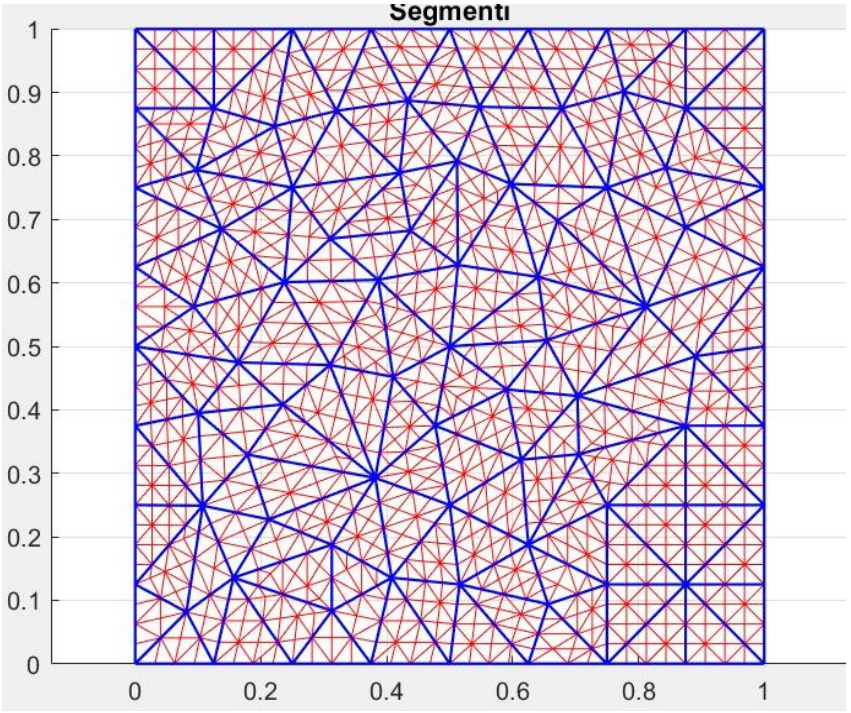
Coefficienti del polinomio interpolante (calcolato su nodi di Chebyshev tra 0 e 3000):
termine noto: 0
lineare: 10.65
quadratico: $-\ 0.13$
altri: $o(10^{-4})$

Si osserva che dominano i termini lineari e quadratici sugli altri di grado più alto, in accordo con i risultati teorici.

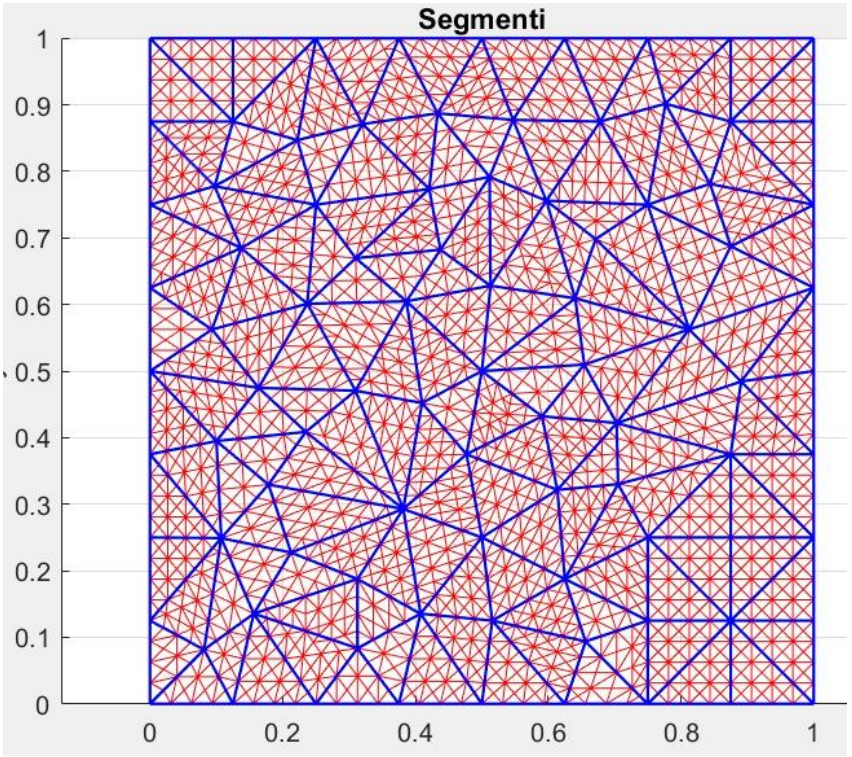
Di seguito alcune mesh ottenute dal raffinamento di Test1.



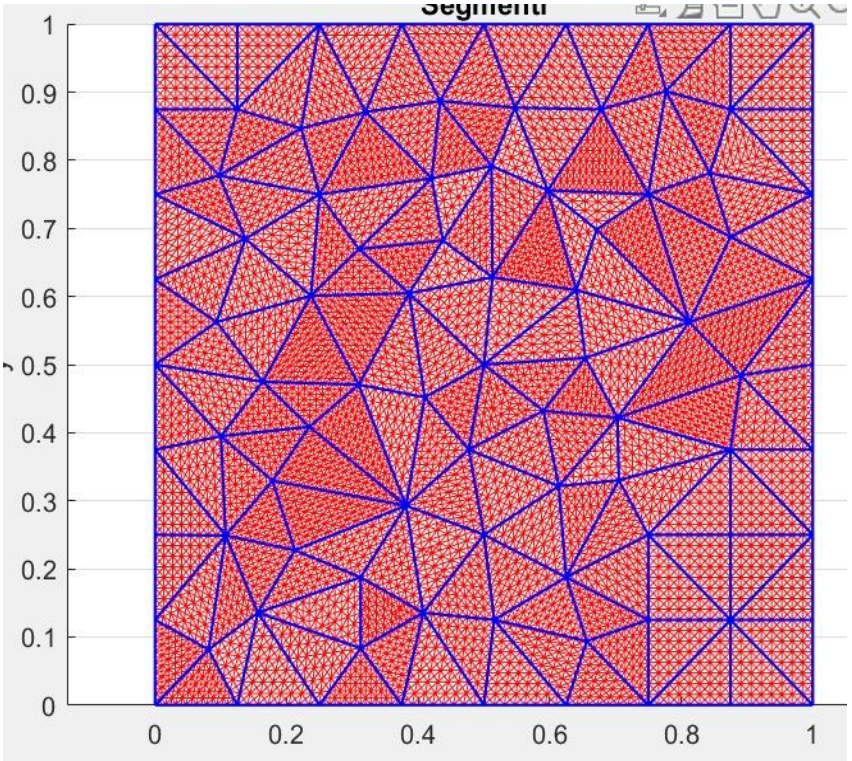
500 iterazioni in 0.00176s



1000 iterazioni in 0.004379s



2000 iterazioni in 0.013031s



10000 iterazioni in 0.241035s