POLITECNICO DI TORINO

Progetto PCS: Raffinamento Complesso

Matteo Racca 283581
Giorgio Musso 282313
Davide Omento 281464

Anno Accademico 2022-2023

1 Processo di Raffinamento

Presentiamo un breve report in merito al problema di raffinamento complesso di una mesh, il cui obbiettivo principale è ottenere una partizione più fine dello spazio. La procedura generale si può suddividere nei seguenti passi:

(1) rintracciare il triangolo di area maggiore T;

(2) trovare il lato e^T più lungo di T;

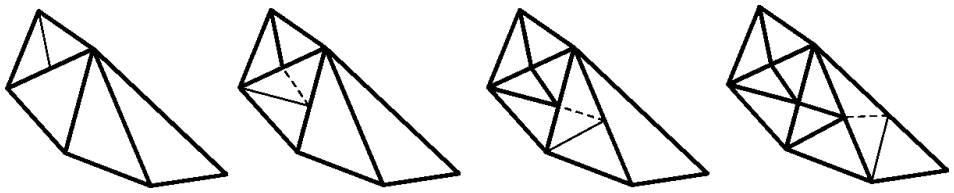
(3) calcolare il punto medio M_{e^T} di e^T e collegarlo al vertice opposto $O^T_{e^T}$;

(4) determinare il triangolo S adiacente al triangolo T attraverso e^T;

(5) ripercorrere i punti (2) e (3) per il nuovo triangolo S;

(6) unire i punti medi $O^T_{e^T}$ e $O^S_{e^S}$;

(7) iterare questo processo fintanto che la mesh non risulti ammissibile, cioè se due triangoli adiacenti hanno in comune o un intero lato o un solo vertice.



1.1 Struttura dati

Per affrontare questo problema abbiamo definito tre classi:

- **Vertices**: rappresenta i vertici della mesh i cui attributi sono l'id, il marker(utilizzato per il test di verifica dell'import del dataset) e le coordinate x e y.
- **Edges**: rappresenta i lati della mesh i cui attributi sono l'id, il marker, il vertice iniziale e finale, la lunghezza ed infine il parametro booleano inMesh per specificare la presenza del lato nella mesh finale. Inoltre la classe Edges è dotata del metodo *MidPoint*, che consente di creare il punto medio del lato.
- **Triangles**: rappresenta i triangoli della mesh i cui attributi sono l'id, gli id dei vertici e dei lati che lo compongono, l'area ed infine il parametro booleano inMesh per specificare la presenza del triangolo nella mesh finale. La classe presenta anche il metodo *FindMaxEdge*, che ricava il lato più lungo tra quelli del traingolo.

Abbiamo definito operatori di confronto e di output per le classi Vertices, Edges e Triangles. In particolare per gli operatori di confronto tra la lunghezza di due lati e l'area di due triangoli sono state utilizzate delle tolleranze.

Abbiamo creato le tre funzioni *ImportVertices*, *ImportEdges* e *ImportTriangles* che importano i dati della mesh iniziale dai file .csv forniti e creano dei vettori contenenti i vertici, i lati e i triangoli.

1.2 Funzioni ausiliarie

Vengono elencate di seguito le funzioni ausiliarie create per dividere il codice a blocchi e permetterci di effettuare dei test per verificarne il corretto funzionamento:

- **findAdiacenceEdge**: Presi in input un elemento della classe *Edges* e il vettore dei triangoli *triangles*, la funzione controlla all'interno del vettore quali triangoli utilizzano il lato e restituisce un vettore contenente gli id di tali triangoli.
- **findOppositeIdVertices**: Presi in input un elemento della classe *Edges* e uno della classe *Triangles*, la funzione restituisce l'id del vertice del triangolo opposto al lato dato.
- **findIdEdgeBetweenVertices**: Presi in input l'id di due vertici, un elemento della classe *Triangles* e il vettore dei lati, la funzione restituisce l'id del lato contenuto tra i due vertici.
- **findTriangleMaxArea**: Preso in input il vettore dei triangoli restituisce il triangolo di area maggiore tra quelli presenti nella mesh (inMesh = true).
- **divideTriangleIn2**: La funzione prende in input un elemento *Triangles* T, un elemento *Edges* l, alcune variabili di riferimento come "idPrecMidPoint", "idPrecFirstHalf" e "idPrecSecondHalf", utili per il processo iterativo, e i vettori di vertici, lati e triangoli.
  - Tramite la funzione *findAdiacenceEdge* crea il vettore degli id dei triangoli adiacenti a l.
  - Vengono aggiunti alla mesh il punto medio di l, creato utilizzando il metodo *MidPoint*, e i due nuovi lati generati dalla sua divisione.
  - Con la funzione *findOppositeIdVertices* trova il vertice opposto a l e aggiunge alla mesh il lato che collega il punto medio di l e il vertice appena trovato.
  - Aggiunge i due nuovi triangoli, generati dal taglio di T, alla mesh ed elimina l e T (attributo inMesh = false) dalla mesh.
  - Aggiorna le variabili di riferimento per poter richiamare la funzione in modo iterativo.
  - La funzione restituisce il vettore con gli id dei triangoli adiacenti a l, che viene utilizzato per gestire le eventuali inammissibiltà.
- **divideTriangleIn3**: La funzione prende in input un elemento *Triangles* T, un elemento *Edges* l, alcune variabili di riferimento come "idPrecMidPoint", "idPrecFirstHalf" e "idPrecSecondHalf", utili per il processo iterativo, e i vettori di vertici, lati e triangoli.
  - Tramite la funzione *findAdiacenceEdge* trova gli id dei triangoli adiacenti a l.
  - Aggiunge alla mesh il punto medio di l e i due nuovi lati creati dalla sua divisione.
  - Con la funzione *findOppositeIdVertices* trova il vertice opposto a l e aggiunge alla mesh il lato che collega il punto medio di l e il vertice trovato.
  - Elimina dalla mesh il lato l e il triangolo T (inMesh = false).
  - Aggiunge alla mesh il lato che unisce il punto medio del lato più lungo del triangolo dell'iterazione precedente e il punto medio di l.
  - Crea i tre nuovi triangoli ottenuti e aggiorna le variabili di riferimento.
  - La funzione restituisce il vettore con gli id dei triangoli adiacenti a l, che viene utilizzato per gestire le eventuali inammissibiltà.

1.3 Descrizione del processo iterativo

In questa sezione presentiamo il corpo del nostro codice.

La funzione principale è *raffinamentoComplesso* che svolge tutte le operazioni necessarie per la costruzione della mesh raffinata elencate a inizio report, iterando il procedimento un numero di volte pari al parametro θ fornito nel *main-program*.

Diamo una descrizione più approfondita della funzione **raffinamentoComplesso**:

1. Si definisce un ciclo for per impostare il criterio di terminazione. Il programma svolge un numero di iterazioni pari al parametro θ.

   I. Utilizzando la funzione *findTriangleMaxArea* si estrae il triangolo T_{max} di area maggiore nella mesh.

   II. Utilizzando il metodo *findMaxEdge* su T_{max} si ottiene il lato più lungo l_{max}.

   III. Con la funzione *divideTriangleIn2* si effettua una divisione di T_{max} in due nuovi triangoli, collegando il punto medio di l_{max} al vertice opposto.

   IV. Si inserisce un nuovo ciclo while annidato per valutare se la mesh è ammissibile. Il ciclo si conclude quando il numero di triangoli adiacenti a l_{max} è diverso da 2.

      i. In caso di inammissibilità, si seleziona il triangolo $T^{(2)}_{max}$ adiacente a T_{max} rispetto a l_{max} e si prende il lato più lungo $l^{(2)}_{max}$ tramite la funzione *findMaxEdge*.

      ii. Se l_{max} e $l^{(2)}_{max}$ coincidono, si effettua una divisione di $T^{(2)}_{max}$ in due nuove triangoli seguendo sempre la regola di bisezione sul lato lungo. A questo punto la mesh ottenuta è ammissibile, perciò il ciclo viene terminato con il comando break.

      iii. Se invece i lati lunghi non sono coincidenti, il triangolo $T^{(2)}_{max}$ viene diviso in tre nuovi triangoli richiamando la funzione *divideTriangleIn3*.

      iv. Nel caso del punto [iii.] la mesh potrebbe ancora risultare inammissibile, è dunque necessario aggiornare le variabili per poter eseguire nuove iterate del ciclo.

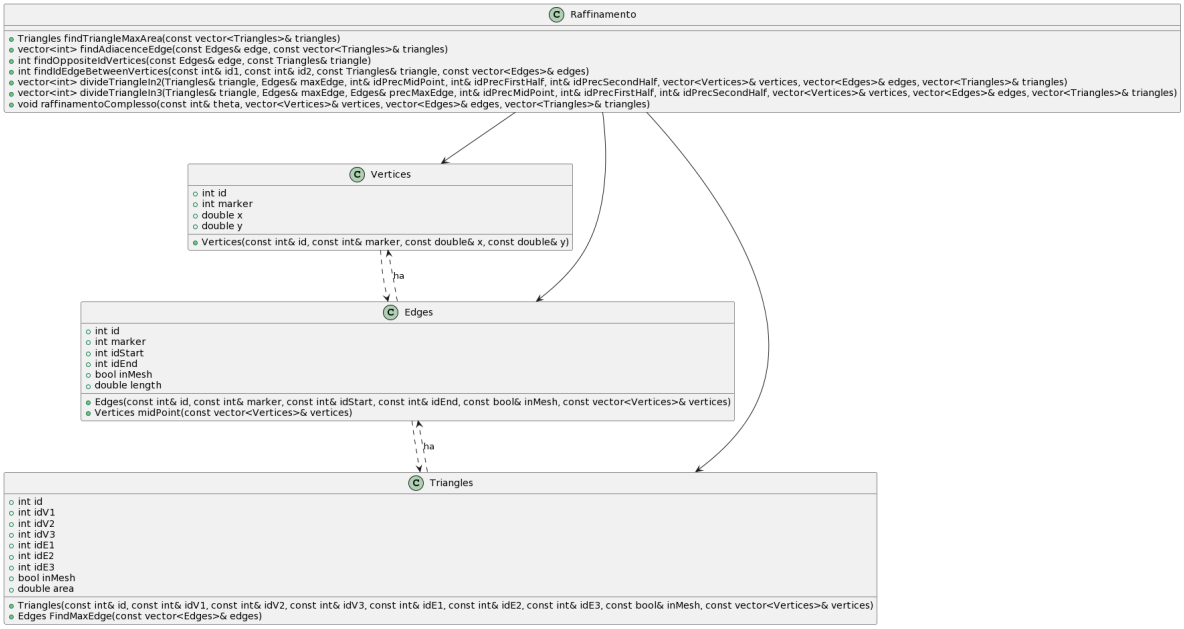
1.4 Test di verifica del codice

Durante la fase di scrittura del codice, in seguito all'implementazione di un metodo di una classe o di una funzione, abbiamo provveduto a creare dei test che ci hannno permesso di verificarne il corretto funzionamento. Abbiamo costruito tali test in modo che ci fornissero in output ciò che ci aspettavamo dalle funzioni o metodi.

Ad esempio, per verificare le funzioni di import della mesh iniziale abbiamo utilizzato gli attributi marker in modo analogo a quanto già fatto nell'esercitazione 4.

In particolare i test sono stati fondamentali per valutare il funzionamento di *divideTriangleIn2* e *divideTriangleIn3*, le funzioni cardine del processo di raffinamento che effettuano diverse operazioni di modifica e aggiornamento dei vettori di vertici, lati e triangoli che compongono la mesh.

1.5 Visualizzazione UML del codice



2 Analisi del costo computazionale

In questa sezione valutiamo il costo computazionale delle funzioni precedentemente introdotte:

- **ImportVertices, ImportEdges, ImportTriangles**: Il costo computazionale di queste funzioni dipende dal numero di righe nel file specificato dal parametro *nameFile*. Se il file contiene n righe, la complessità sarà $\mathcal{O}(n)$, in quanto viene eseguita un'operazione di estrazione dei dati per ogni riga letta.
- **findAdiacenceEdge**: Questa funzione ha un costo computazionale $\mathcal{O}(t)$, dove t è il numero di triangoli presenti nel vettore *triangles*. La funzione scorre tutti i triangoli e verifica se l'*edge* dato fa parte di ciascun triangolo.
- **findOppositeIdVertices**: Questa funzione ha un costo computazionale $\mathcal{O}(1)$, poiché esegue un numero costante di operazioni indipendentemente dalla dimensione dei dati in input.
- **findIdEdgeBetweenVertices**: Questa funzione ha un costo computazionale $\mathcal{O}(1)$, poiché esegue un numero costante di operazioni indipendentemente dalla dimensione dei dati in input.
- **findTriangleMaxArea**: Questa funzione ha un costo computazionale $\mathcal{O}(t)$, dove t è il numero di triangoli presenti nel vettore *triangles*. La funzione itera su tutti i triangoli confrontandone l'area per trovare quello con l'area massima.
- **divideTriangleIn2**: Questa funzione ha un costo computazionale $\mathcal{O}(t)$, dove t è il numero di triangoli presenti nel vettore *triangles*, perchè al suo interno richiama la funzione *findAdiacenceEdge* ed esegue altre operazioni indipendentemente dai dati in input.
- **divideTriangleIn3**: Questa funzione ha un costo computazionale $\mathcal{O}(t)$, dove t è il numero di triangoli presenti nel vettore *triangles*, perchè al suo interno richiama la funzione *findAdiacenceEdge* ed esegue altre operazioni indipendentemente dai dati in input.
- **RaffinamentoComplesso**: Questa funzione ha un costo computazionale $\mathcal{O}(t)$, dove t è il numero di triangoli presenti nel vettore *triangles*, perchè al suo interno richiama le funzioni *divideTriangleIn2* e *divideTriangleIn3* che hanno entrambe costo computazionale $\mathcal{O}(t)$.

In seguito all'analisi teorica dei vari costi computazionali, abbiamo deciso di valutare graficamente la relazione che c'è tra il numero di triangoli della mesh finale e il tempo di esecuzione:

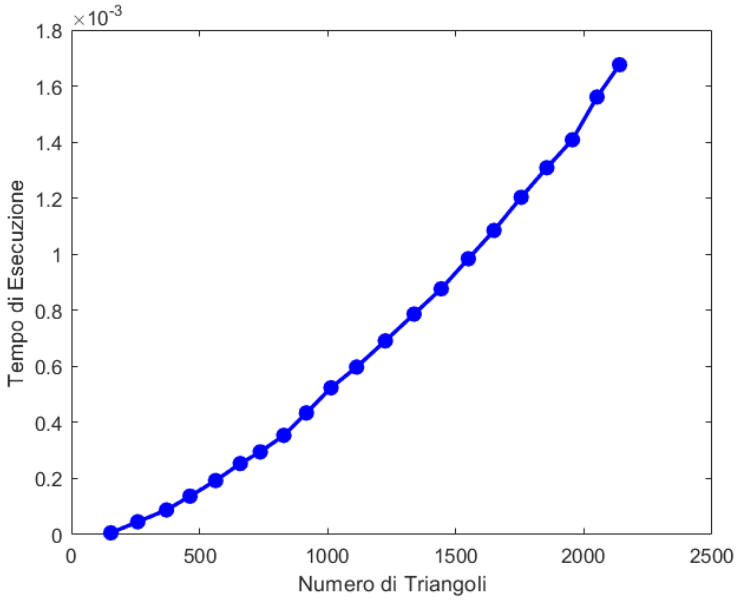


Figure 1: Tempo di esecuzione in funzione del numero di triangoli

Abbiamo anche creato altri due grafici per vedere la relazione che c'è fra l'aumento del parametro θ e il tempo di esecuzione del programma nella *Figure* 2 e il numero totale di iterazioni eseguite nella *Figure* 3:

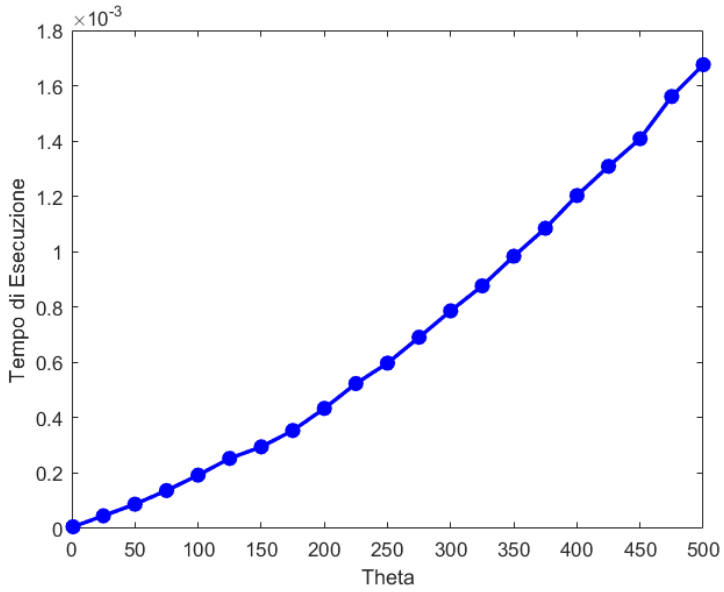


Figure 2: Tempo di esecuzione in funzione di θ

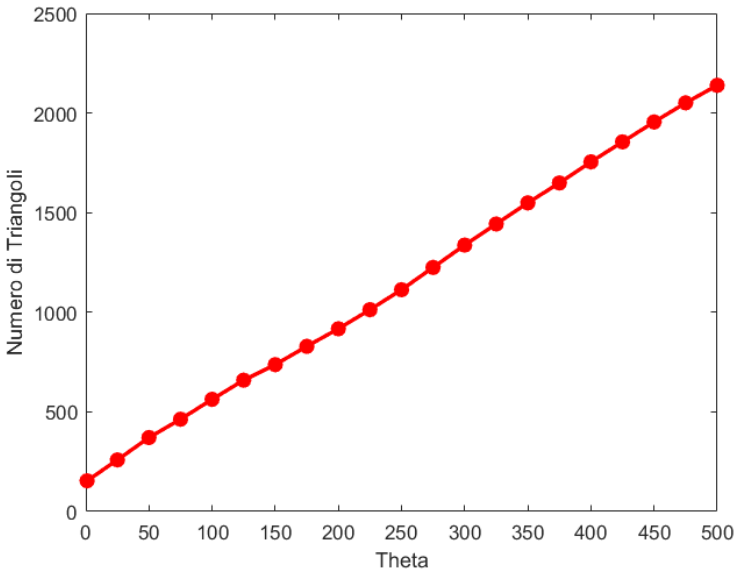


Figure 3: Numero di iterazioni in funzione di θ

3 Risultati ottenuti dal processo di raffinamento

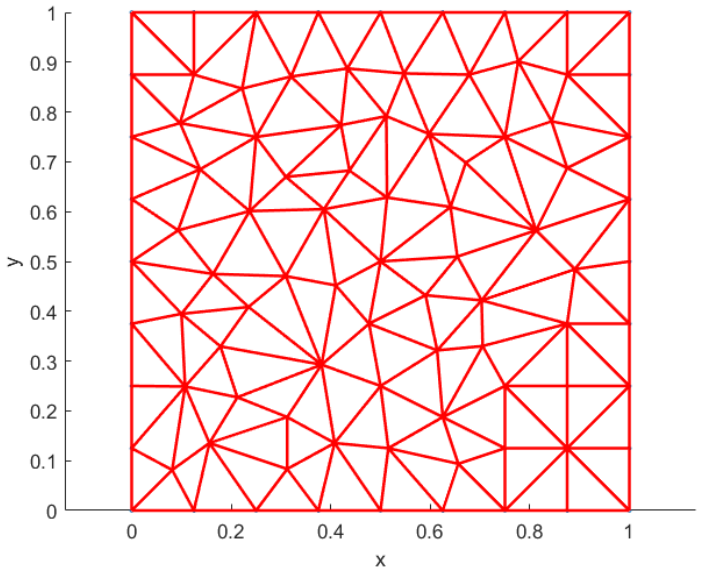


Figure 4: Mesh iniziale non raffinata

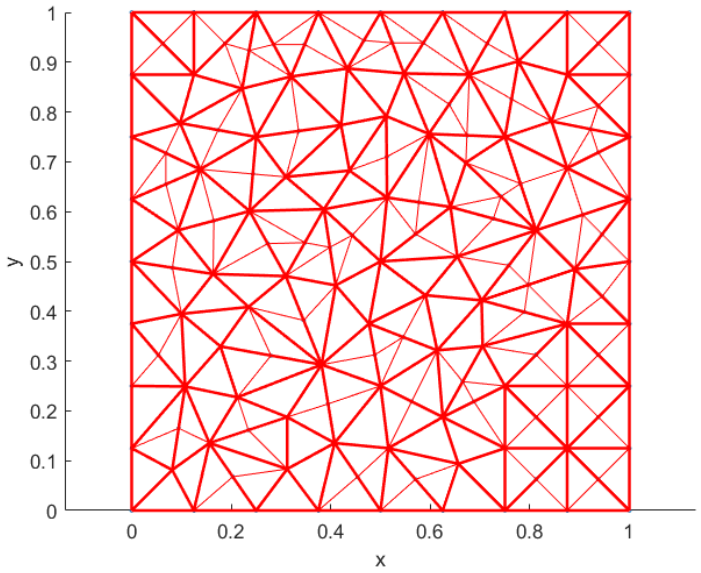


Figure 5: Mesh ottenuta con $\theta = 100$

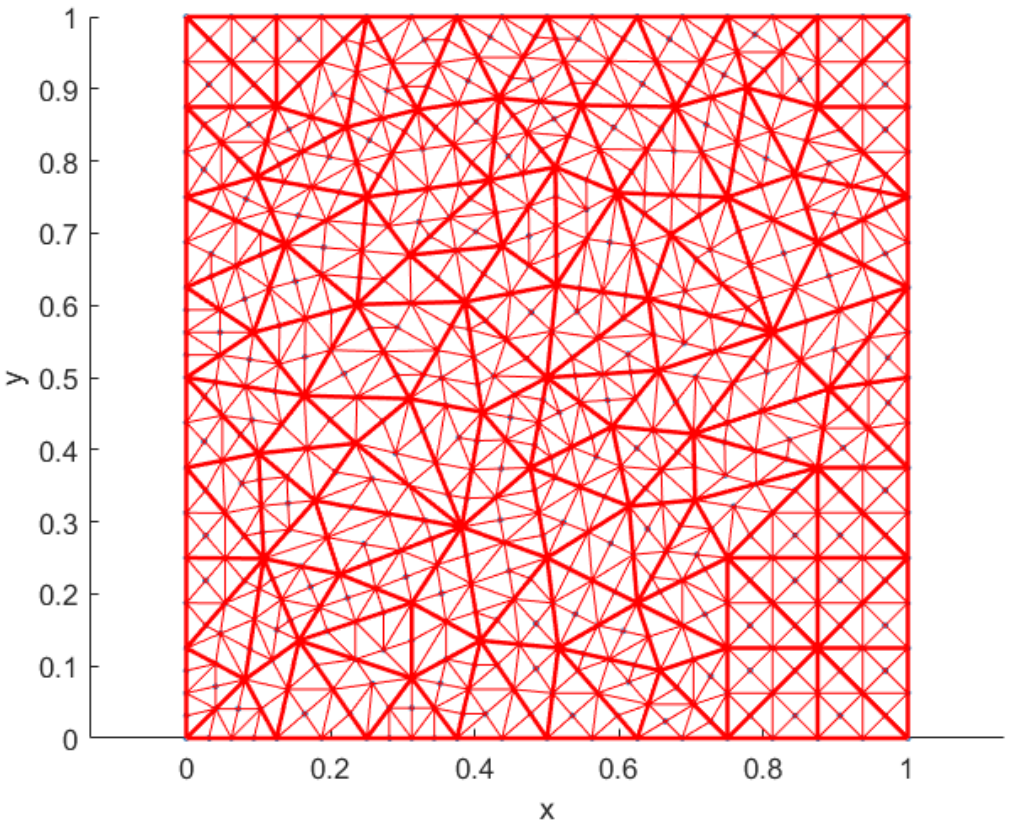


Figure 6: Mesh ottenuta con $\theta = 500$