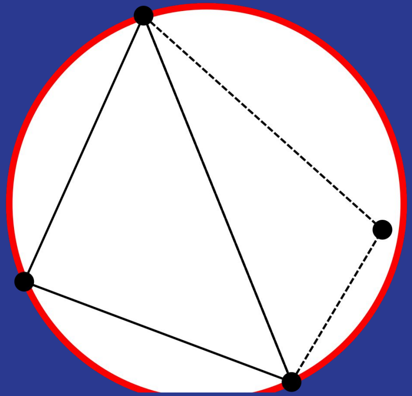


PROGETTO: DELAUNAY

Struttura e descrizione dell'algoritmo e del costo computazionale

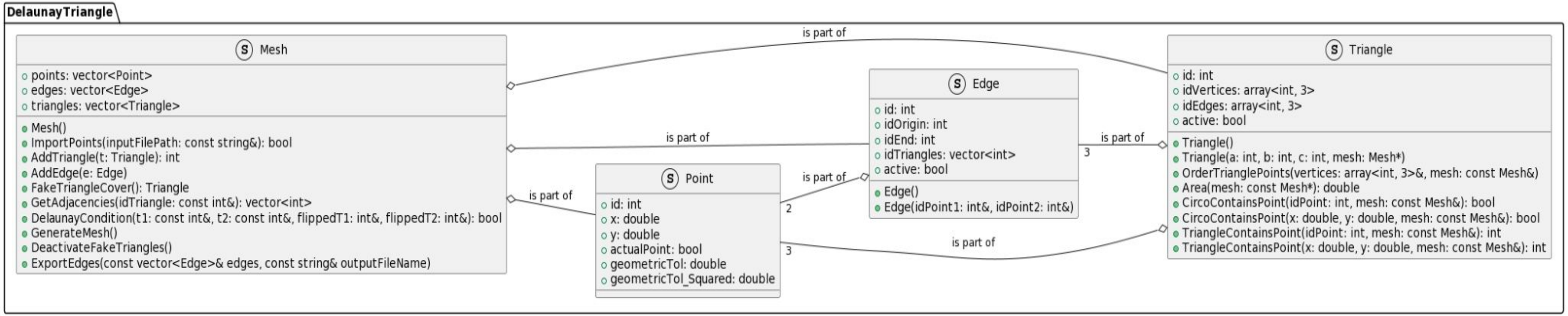
- Annalisa Belloni s281526
- Samuele Bocco s283197
- Sara Bonino s282836

Link al repository di GitHub

Politecnico di Torino
1859

Strutture Dati utilizzate

Namespace *DelaunayTriangle*



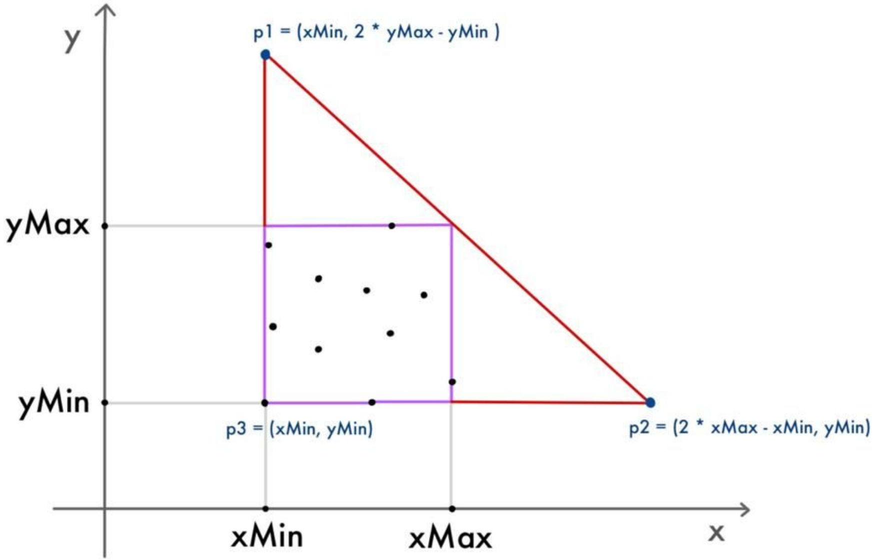
[Documentazione](#)

[Diagramma uml con note](#)

Il nostro algoritmo

1) Costruzione di un grande triangolo fittizio che contenga al suo interno (o sul bordo) tutti i punti.
2) Inserimento di tutti i punti all'interno della mesh, verificando la condizione di Delaunay.
3) Disattivazione di lati e triangoli che sono costruiti a partire dai punti fittizi del triangolo iniziale.

Passo 1: FakeTriangleCover( )



1) Itero sui punti: trovo ascisse e ordinate minime e massime.
2) Creo le coordinate (xMin, yMin), (xMin, 2 * yMax - yMin), (2 * xMax - xMin, yMin).
3) Itero sui punti: controllo se ci sono punti reali con le coordinate trovate. Se ci sono, ne salvo l'id.
4) Per le coordinate fittizie che non coincidono con quelle di punti reali, creo dei nuovi punti. Ne prendo l'id.
5) Costruisco il triangolo fittizio con gli id salvati.

Costo computazionale: O(2n)

Passo 2: GenerateMesh( ) - parte I

```
for punto in punti:
        alreadyEntered = false    // serve per i punti di bordo
         for triangolo in triangoli:
                if triangolo è attivo e punto non è esterno a triangolo:
                        disattivo triangolo
                        creo i tre nuovi triangoli
                        // per tutti e tre i nuovi triangoli
                        if triangolo_nuovo ha area non nulla
                                aggiungo triangolo_nuovo ai triangoli della mesh
                                aggiungo triangolo_nuovo ai triangoli su cui verificare Delaunay
                        else
                                // il punto è su un lato di triangolo
                                disattivo il lato di triangolo su cui il punto è di bordo
```

Passo 2: GenerateMesh( ) - parte II

```
while ci sono triangoli su cui verificare Delaunay
        estraggo l’ultimo triangolo dal vettore delle verifiche
        cerco i triangoli adiacenti
        for adiacente
                verifica Delaunay tra adiacente e triangolo da verificare
                if è avvenuto il flip
                        aggiungi i due nuovi triangoli a quelli da verificare
                        break

// se il punto è sul bordo di due triangoli,  il suo inserimento non è ancora terminato
if !alreadyEntered e punto è di bordo per triangolo
        alreadyEntered = true
        continue

break
```

Passo 3: DeactivateFakeTriangles( )

Costo Computazionale:
O(num_edges) = O(c * n), c ≈ 4, 5.

```
for lato in lati
    if (lato attivo e almeno uno dei
            due punti estremi è fittizio)
        disattiva lato
        disattiva i triangoli costruiti
                con lato
```

Costo computazionale teorico

Abbiamo già visto:
Costo computazionale di FakeTriangleCover( ) = **O(2*n)**.
Costo computazionale di DeactivateFakeTriangles( ) = **O(c*n)**.

Nella funzione GenerateMesh( ):

- 2 for annidati che scorrono rispettivamente sui punti non ancora inseriti e sui triangoli, hanno un costo computazionale di O(n*num_triangoli) ~ $O(n^2)$.
- Costo della verifica di Delaunay, che avviene per ogni nuovo punto inserito, (corrispondente al while sui *triangoli da verificare*, in numero << n) che si stima essere O(n*k), con k costante.
  Dunque, Costo computazionale di GenerateMesh( ) = $\mathbf{O(n^2)}$.

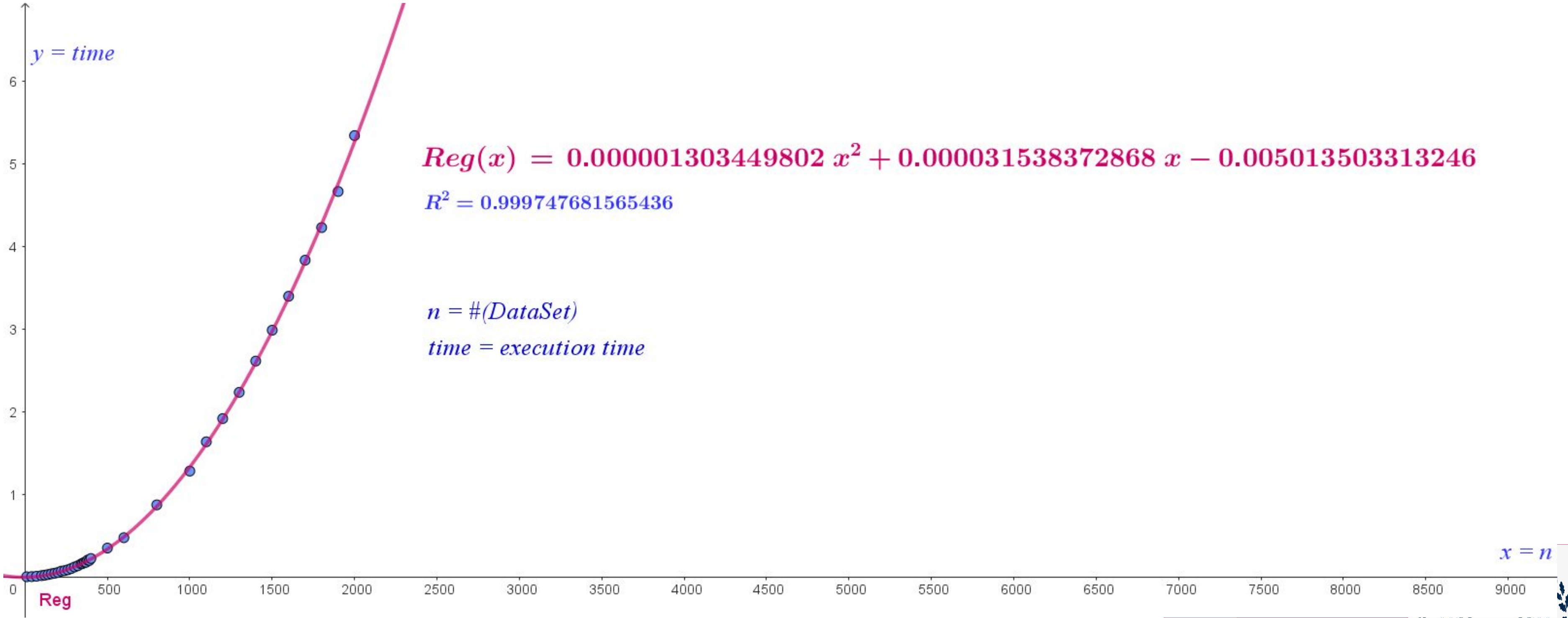
Complessivamente, il costo computazionale di tutto l'algoritmo è dell'ordine di $\mathbf{O(n^2)}$.

Costo computazionale sperimentale

Abbiamo messo in relazione la cardinalità del dataset di punti iniziali, con il tempo di esecuzione del programma (trascurando l'operazione di esportazione degli edge).
Tramite una funzione generatrice di DataSet casuali di punti, abbiamo fatto variare la cardinalità n del set e allo stesso tempo abbiamo memorizzato, tramite dei clock, l'execution time.

Infine, provando ad approssimare i dati così raccolti tramite regressione, abbiamo confermato i risultati teorici: la curva che meglio approssima i dati è proprio una parabola!

Regressione quadratica



Una prima versione del nostro algoritmo

1) Ricerca del triangolo di area massima.
2) Creazione di uno pseudo-ConvexHull, ottenuto “estendendo” il triangolo di area massima.
3) Inserimento dei punti non ancora visitati all’interno della mesh, verificando la condizione di Delaunay:
   se il punto è interno allo pseudo-ConvexHull → GenerateMesh()
   se il punto è esterno allo pseudo-ConvexHull → GestionePuntoEsterno().

GetMaxAreaTriangle( ) e pseudo-ConvexHull

- Costo computazionale di *GetMaxAreaTriangle( )*: O(n*log(n))
  Applicando il Teorema delle Ricorrenze, con alpha = beta = 2, f(n) = n.

- Costo computazionale di *FindPointsEstremi( )* + *AddCardinalPoints( )*: O(n)

1) Ricerca di MaxAreaTriangle. Algoritmo ricorsivo, che sfrutta il paradigma “divide et impera”.

2) Una volta trovati i punti più estremi nelle quattro direzioni cardinali, se diversi dai vertici di MaxAreaTriangle, questi si aggiungono alla mesh tramite la funzione GestionePuntoEsterno( ).
   I punti di bordo della mesh costituiscono lo pseudo-ConvexHull ricercato.

GestionePuntoEsterno( ) - parte I

```
if newPoint è esterno a tutti i triangoli già inseriti nella mesh
    si cercano i punti di bordo che possono essere collegati con newPoint
    si calcolano tutte le coppie ammissibili di punti di bordo, collegabili a newPoint
    for coppia di punti ammissibili
        creo hypotheticTriangle che ha per vertici newPoint e i due punti della coppia
        //verifichiamo se hypotheticTriangle può essere aggiunto o meno alla mesh
        valido = true
        if hypotheticTriangle è degenere (ha area nulla)
            valido = false
        if (valido)
            for punto in PuntiDiBordo
                if punto è interno ad hypotheticTriangle
                    valido = false
                    si memorizza punto come "non di bordo"
```

GestionePuntoEsterno( ) - parte II

if (valido)

aggiungiamo hypotheticTriangle alla mesh e verifichiamo Delaunay

//terminato il ciclo sulle coppie di punti ammissibili

eliminiamo da PuntiBordo tutti i punti che non lo sono più

aggiungiamo newPoint a PuntiBordo

Costo computazionale di *GestionePuntoEsterno( ) x n_esterni*:
almeno O(n_esterni*n_bordo*n_edges_bordo) ≈ O(1/9 * n_esterni^3) - ipotizzando n_bordo ≈ n_esterni/3 -.

Confronto tra i Costi Computazionali dei due algoritmi

Per n grande, al caso peggiore:

- Il metodo dello pseudo-convex hull ha costo $O(n*log(n)) + O(n) + O(n^3) \approx O(n^3)$ (nel caso peggiore la maggior parte dei punti non ancora inseriti è esterna allo pseudo-convex hull → n_esterni $\approx$ n).
- Il metodo del triangolo fittizio ha costo $O(n^2)$.

Il costo computazionale dell'algoritmo con triangolo fittizio è inferiore ed è la ragione per cui è stato preferito all'altro.

Tolleranza Geometrica

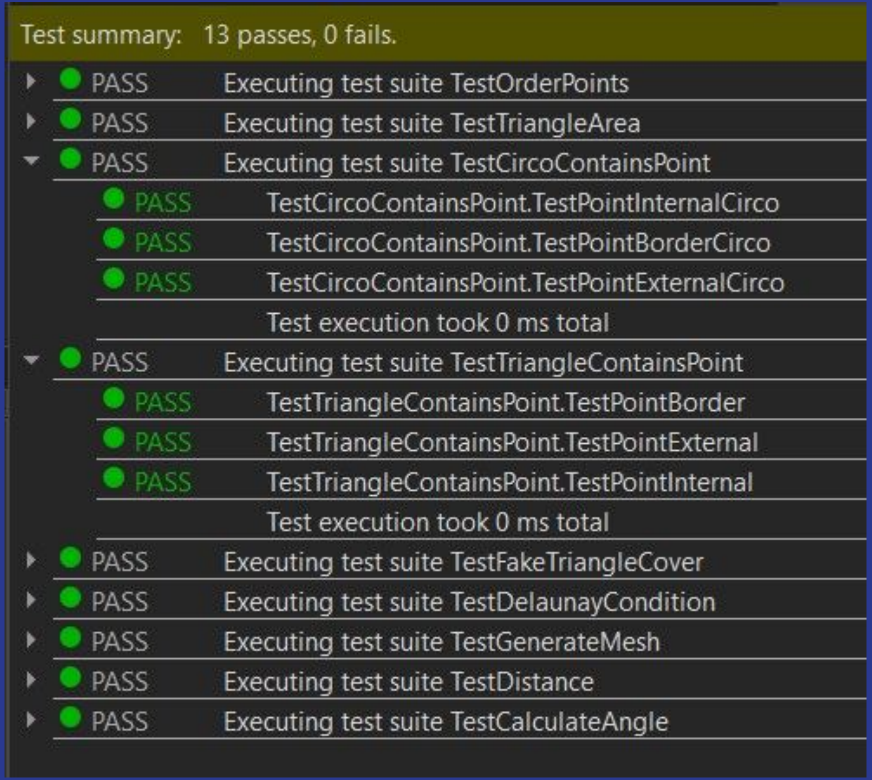
```
static constexpr double geometricTol = 1.0e-12;

static constexpr double geometricTol_Squared =
        max_tolerance(Point::geometricTol * Point::geometricTol,
        numeric_limits<double>::epsilon( ));
```

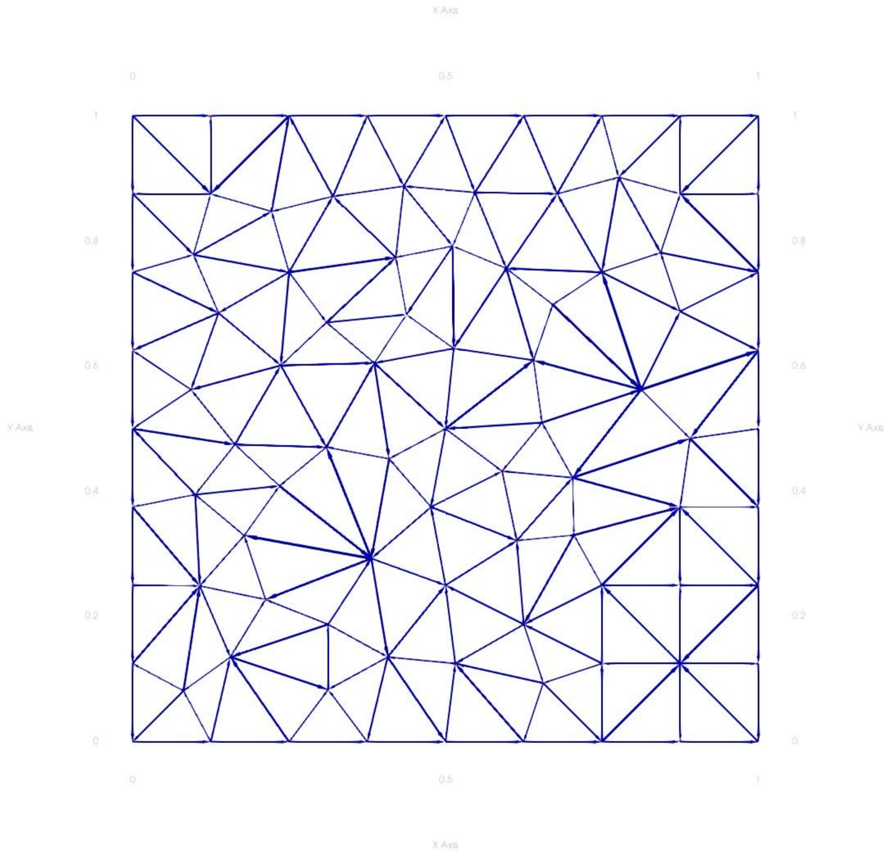
Con i dataset dati in input, è sufficiente che la tolleranza sia dell'ordine di 10^{-5} per ottenere il risultato corretto.

Points.csv → massima tolleranza 1.0e-04,
Test2.csv → massima tolleranza 1.0e-05

Test

13 test per verificare le nostre funzioni. Tutti corretti

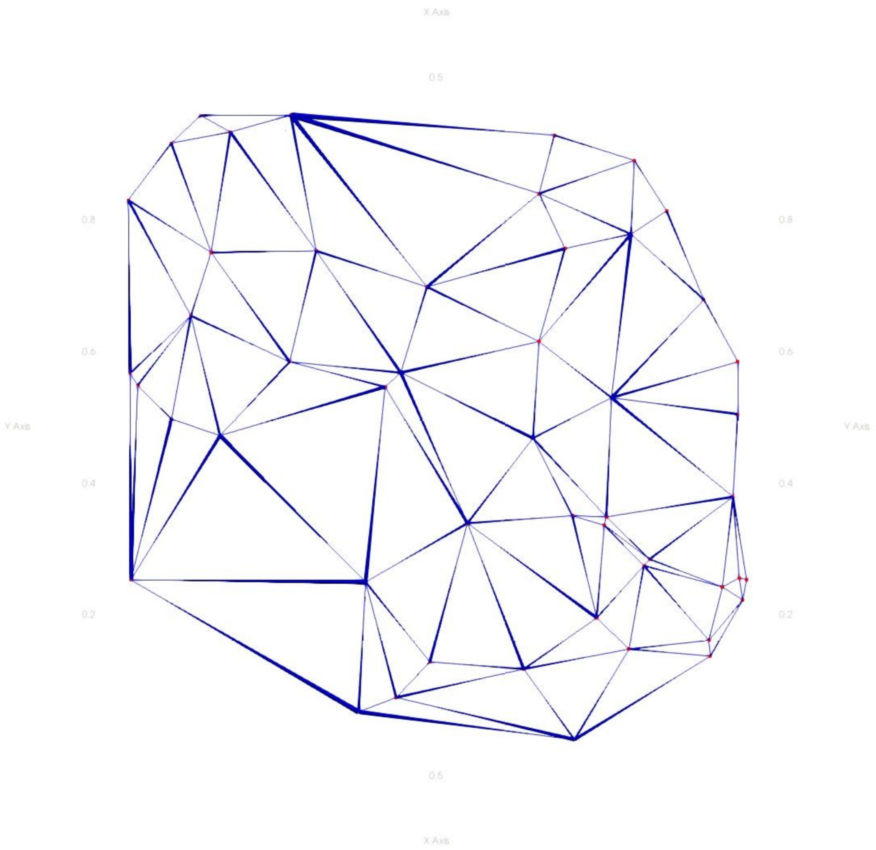




Risultati

Dataset: Points.csv

Stampato con Paraview



Risultati

Dataset: Test2.csv

Stampato con Paraview