RAFFINAMENTO DI UNA MESH TRIANGOLARE

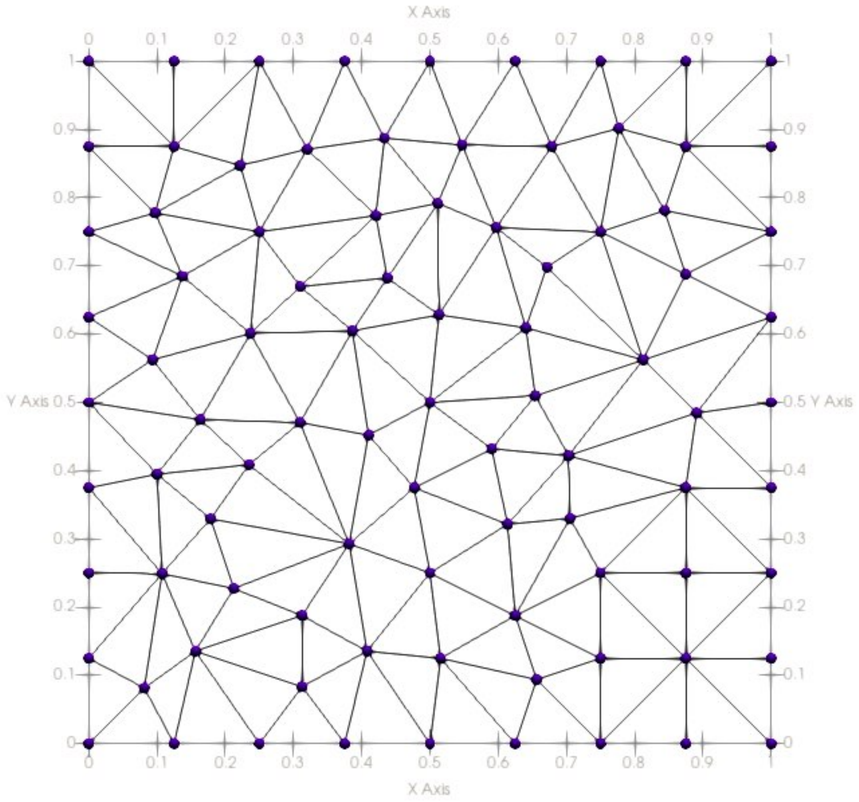
PROGRAMMAZIONE E CALCOLO SCIENTIFICO A.A.2022/2023

Beltrame Francesco 282019
Effretti Marco 269107
Lobascio Domenico 224512
Vittori Gabriele 270082

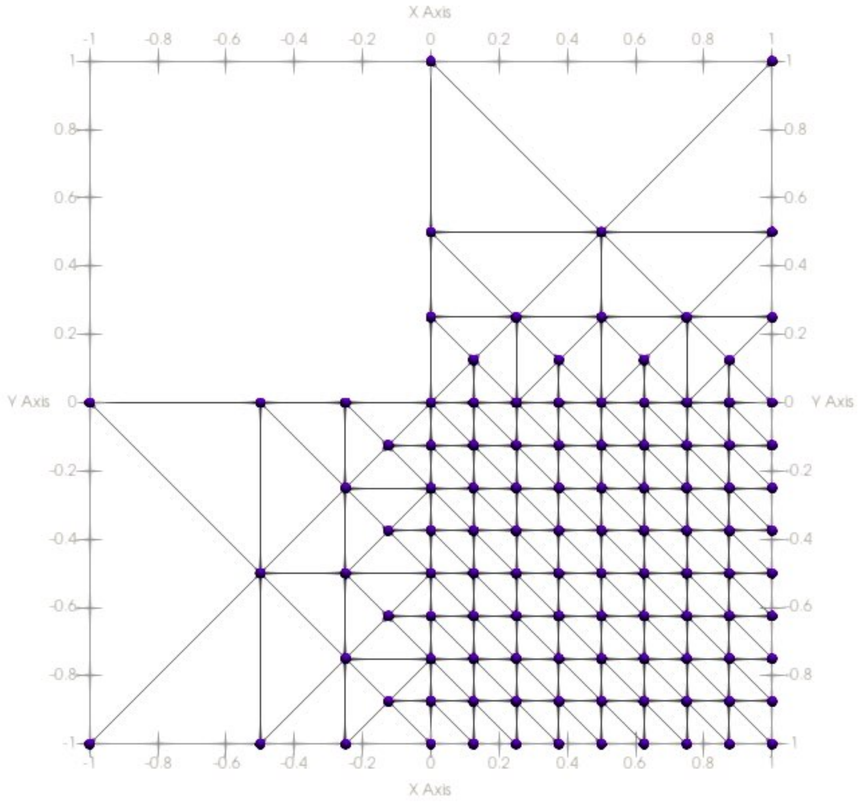
Data: 22.06.2023

Il problema (1/2)

Le mesh iniziali



Test 1

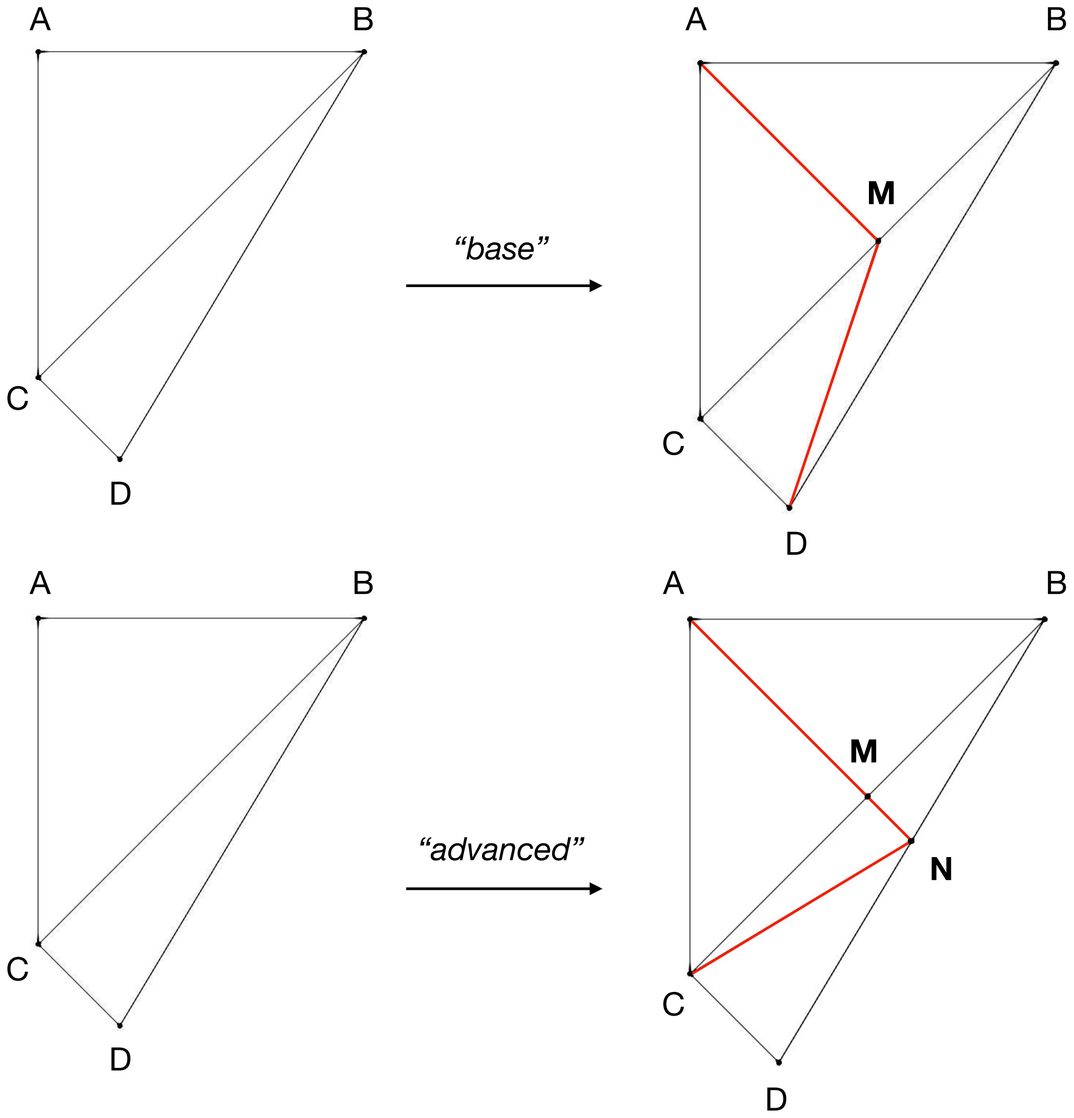


Test 2

Il problema (2/2)

Algoritmi di risoluzione

- **Base**: dopo aver diviso il triangolo di area maggiore rispetto al lato più lungo si collegano il punto medio al vertice opposto del triangolo adiacente
- **Avanzato**: dopo aver diviso il triangolo di area maggiore rispetto al lato più lungo, si ricorre allo stesso modo sul triangolo adiacente aggiungendo poi i lati mancanti



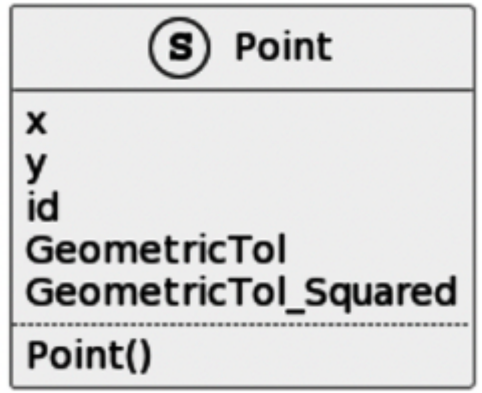
Struttura Dati (1/2)

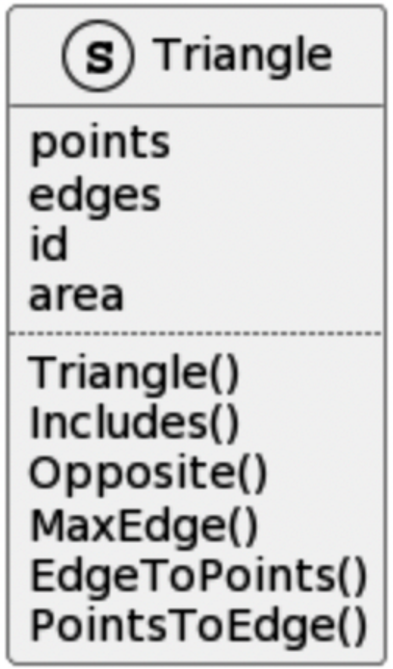
Strutture:

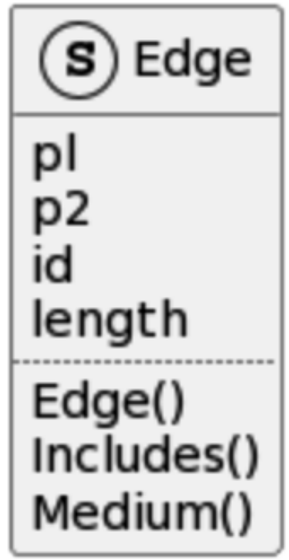
- **Point:** composto dalle coordinate x e y e un identificatore

- **Edge:** composto da due punti, gli estremi del segmento, da un identificatore e dalla sua lunghezza

- **Triangle:** composto da un array di tre punti, un vector di tre lati, un identificatore e la sua area







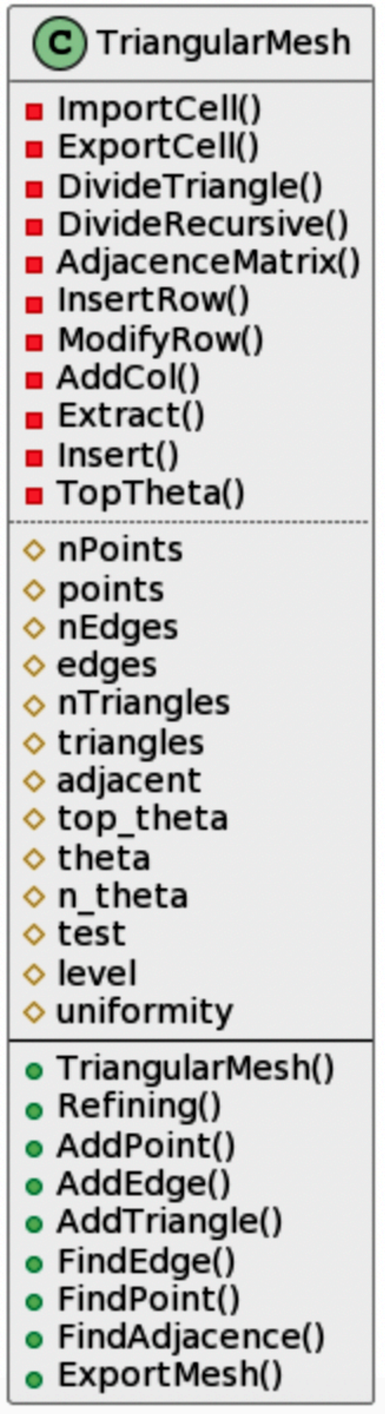
Struttura Dati (2/2)

Classi: TriangularMesh

Creata come classe per la necessità di suddividere i vari metodi e attributi in protetti(◇), privati(▪) e pubblici(●).

Identificati dal rombo giallo gli **attributi** sono:

- tre vector che contengono tutti i punti, i lati e i triangoli
- numero totale di punti, lati e triangoli
- “matrice” di adiacenza
- theta, ossia la percentuale di triangoli da modificare
- il test su cui applicare il raffinamento
- il livello a cui raffinare, se “base” o “advanced”
- parametro di uniformità

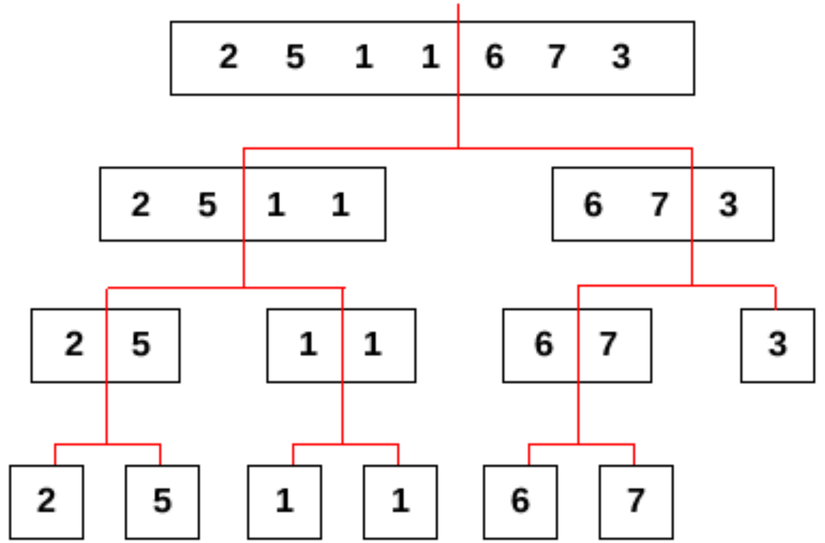


Algoritmo di Sorting

MergeSort

Scelto per il costo computazionale vantaggioso, pari a O(nlog(n)).

Utilizzato principalmente per ordinare i triangoli in ordine decrescente di area, in modo tale da estrarre quelli da raffinare (salvati nel vettore **top_theta**), ma anche per ordinare i lati di ognuno di essi per lunghezza.



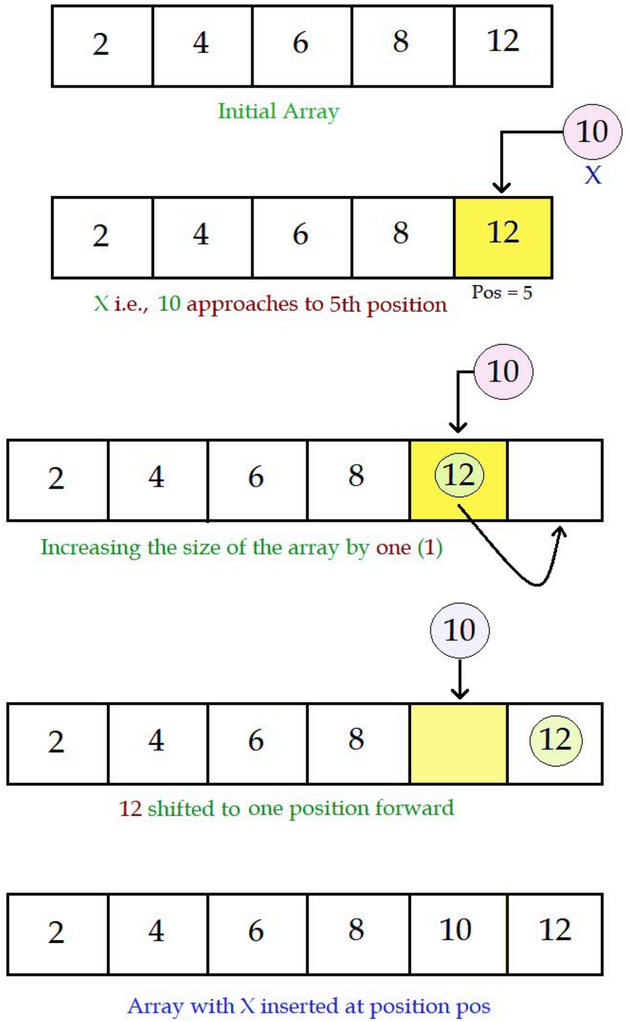
Inserimento ordinato

SortInsert

Si è cercato di ottimizzare il più possibile l'inserimento ordinato tramite un algoritmo di costo lineare (O(n)), con la scelta di partire dal fondo.

Di particolare rilevanza è la scelta di mantenere invariata la dimensione del vettore, riscalando opportunatamente dopo ogni inserimento.

La sua applicazione sarà chiarita in seguito.



Approfondimenti: Struttura Dati (1/4)

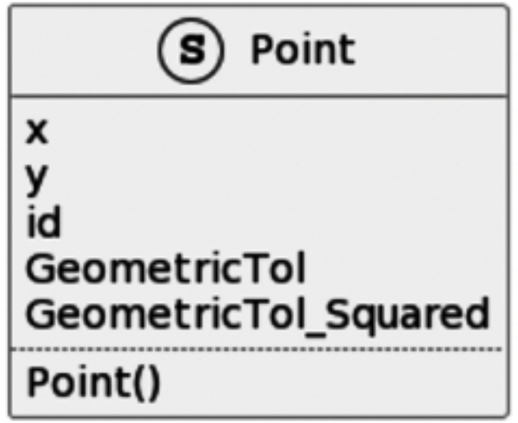
Struct: Point

Costruttori:

```
Point() = default;
Point(const double x, const double y, const unsigned int id): x(x), y(y), id(id){ ...}
Point(const Point& p): x(p.x), y(p.y), id(p.id){}
Point& operator=(const Point &p){x = p.x; y = p.y; id = p.id; return *this;}
```

Attributi rilevanti:

- GeometricTol: tolleranza minima in dimensione 1 (τ=1.0e-12)
- GeometricTol_Squared: tolleranza minima in dimensione 2 (max{τ^2, eps})



Approfondimenti: Struttura Dati (2/4)

Struct: Edge

Costruttori:

```
Edge() = default;
Edge(Point p1, Point p2, unsigned int id): p1(p1), p2(p2), id(id){ ...}
Edge(const Edge &E):p1(E.p1), p2(E.p2), id(E.id), length(E.length){ ...}
Edge& operator=(const Edge &E){p1 = E.p1; p2=E.p2; id = E.id; length=E.length; return *this;}
```

Metodi:

```
bool Includes(const Point p){return (p==p1 || p==p2);}
Point Medium(unsigned int id_p) {return Point((p1.x+p2.x)*0.5,(p1.y+p2.y)*0.5,id_p);}
```

Approfondimenti: Struttura Dati (3/4)

Struct: Triangle (1/2)

Costruttori:

```
Triangle() = default;
Triangle(vector<Edge> edges, unsigned int id){ ...}
Triangle(const Triangle &T): points(T.points), edges(T.edges), id(T.id), area(T.area){ ...}
Triangle& operator=(const Triangle &T){points=T.points; edges=T.edges; id=T.id; area=T.area; return *this;}
```

Metodi:

```
bool Includes(const Edge E){for(Edge &edge : edges) if(edge==E) return true; return false;}
bool Includes(const Point p){for(Point &pt : points) if(pt==p) return true; return false;}
Point Opposite(Edge E){ ...}
Edge Opposite(Point p){ ...}
Edge MaxEdge(){return edges[0];}
array<Point, 3> EdgesToPoints(){ ...}
Edge PointsToEdge(Point p1, Point p2){ ...}
```

Approfondimenti: Struttura Dati (3/4)

Struct: Triangle (2/2)

```
Triangle::Triangle(vector<Edge> edges, unsigned int id): id(id){
  MSort(edges);
  this->edges = edges; points=EdgesToPoints();
  area = AreaTriangle(points[0],points[1],points[2]);
  if(area<0){ points={points[1], points[0], points[2]}; area = abs(area); }
}
```

L'aspetto principale è l'ordinamento dei lati (per lunghezza) e di conseguenza anche dei punti, questi ultimi ordinati in senso antiorario a partire dagli estremi del lato più lungo.

Questo viene realizzato tramite il metodo:

```
array<Point, 3> EdgesToPoints(){ ...}
```

e un controllo sull'area.

Approfondimenti: Struttura Dati (4/4)

Class: TriangularMesh (1/2)

Costruttori:

```
TriangularMesh() = default;
TriangularMesh(const string cell0D, const string cell1D, const string cell2D, short int test);
```

In particolare il secondo è chiamato da main:

```
TriangularMesh M("Cell0Ds.csv","Cell1Ds.csv","Cell2Ds.csv",test);
```

A cui corrisponde, nel file .cpp:

```
TriangularMesh::TriangularMesh(const string cell0D, const string cell1D, const string cell2D, short int test): test(test){
    //importa la mesh triangolare
  if(!ImportCell0D(cell0D)){cerr<<"Error in import file"<<endl;}
  if(!ImportCell1D(cell1D)){cerr<<"Error in import file"<<endl;}
  if(!ImportCell2D(cell2D)){cerr<<"Error in import file"<<endl;}
  this->AdjacenceMatrix();
}
```

Approfondimenti: Struttura Dati (4/4)

Class: TriangularMesh (2/2)

Metodi a contorno:

```
void AddPoint(Point point, unsigned int indice=UINT_MAX){ ...}
void AddEdge(Edge edge, unsigned int indice=UINT_MAX){ ...}
void AddTriangle(Triangle triangle, unsigned int indice=UINT_MAX){ ...}
Edge FindEdge(Point p1, Point p2){ ...}
Edge FindEdge(unsigned int id_e){ ...}
Point FindPoint(unsigned int id_p){ ...}
Triangle FindAdjacence(Triangle &T, Edge E){ ...}
```

I metodi principali sono descritti più dettagliatamente nelle slide successive,

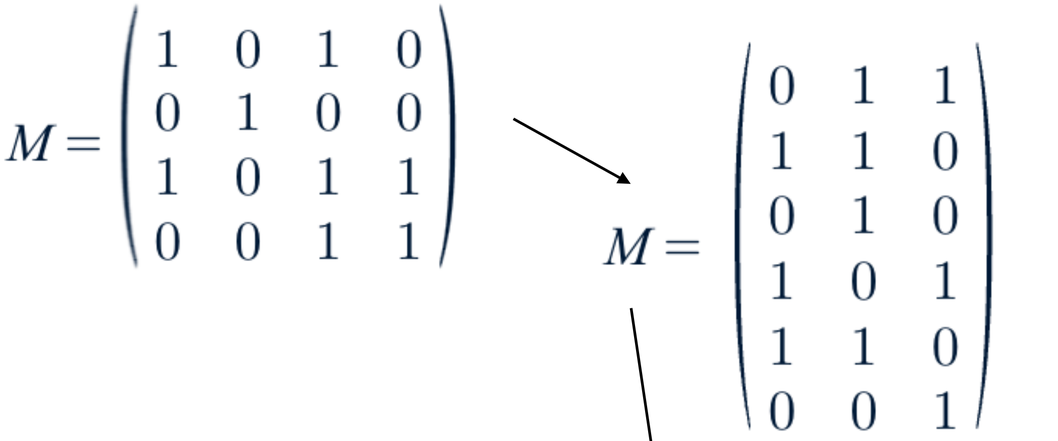
ma prima…

Matrice di Adiacenza

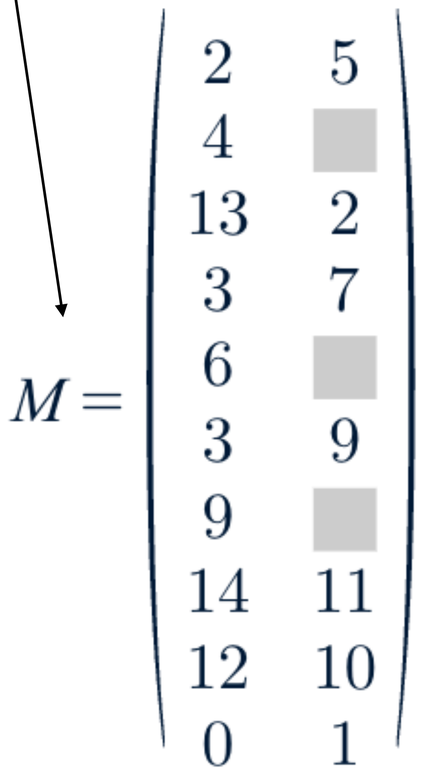
```
vector<vector<unsigned int>> adjacent;
```

Evoluzione

- **Matrice booleana sparsa 1**:
  -dimensione: n x n con n=nTriangles;
  -indice di riga e colonna = id dei triangoli;
  -costo in memoria elevato (O(n^2));
  -non ottimizzata per la ricerca;



- **Matrice booleana sparsa 2**:
  -dimensione m x n con m=nEdges;
  -indice di riga = id dei lati, indice di colonna = id dei triangoli;
  -costo in memoria ancora più elevato (O(m x n));
  -più adatta alla ricerca data l'informazione lato $\in$ triangolo;



- **Vector di vector**:
  -dimensione massima: m x 2;
  -indice di riga = id dei lati, ogni riga è un vector di id dei triangoli adiacenti rispetto al lato considerato;
  -costo in memoria ridotto (O(m));
  -ricerca ottimizzata ulteriormente (O(1));

“Matrice”: Riempimento e modifica

- **Riempimento**: all’interno del metodo `void TriangularMesh::AdjacenceMatrix()` per ridurre al minimo i tempi di riempimento si scorrono per ogni triangolo i lati, e con la seguente operazione

```
adjacent[edge.id].push_back(triangle.id);
```

  viene costruita la matrice, conseguentemente a un opportuno riscalamento;

- **Modifica**: facilitato dall’accesso diretto, l’aggiornamento delle adiacenze risulta ottimizzato anche grazie all’utilizzo di alcuni metodi specifici:

```
void InsertRow(const vector<unsigned int> &t, unsigned int id_edge=UINT_MAX){ ...}
void ModifyRow(unsigned int id_t_old, unsigned int id_t_new, unsigned int id_edge){ ...}
void AddCol(unsigned int id_tr, unsigned int id_edge){ ...}
```

Refining

```
void Refining(double theta, string level="base", string uniformity="non-uniform"){ ...}
```

Il metodo da cui parte il raffinamento:

```
void TriangularMesh::Refining(double theta, string level, string uniformity){
  this->theta = theta; this->level = level; this->uniformity = uniformity;
  TopTheta();
  while(n_theta > 0){
    if(level=="base" || level=="advanced") DivideTriangle();
    else {cerr<<"Error: invalid argument"<<endl; throw(1);}
  }
}
```

Come specificato nel sorting, i primi triangoli verranno salvati in un vettore **top_theta**, da cui verranno estratti e divisi nel ciclo successivo.

Particolare attenzione è stata data alla versatilità della fruizione del metodo, infatti dal main è possibile specificare:

- la percentuale di raffinamento: dettata dalla scelta di **theta** $\in$ (0,1);
- il livello: *“base”* o *“advanced”* specificato in **level**;
- possibilità di incrementare l’uniformità della mesh risultante attraverso la parola chiave *“uniform”* (default: *“non-uniform”*), l’implementazione del metodo **Insert** in questo caso risulta determinante;

Approfondimento: top_theta

Extract e Insert

Data la centralità del vettore **top_theta** nel progetto, si vogliono approfondire i seguenti metodi:

```
bool Extract(Triangle &T){ ...}
```
:

-estrae dal **top_theta** il triangolo appena diviso;

```
bool Insert(Triangle &T){ ...}
```
:

-inserisce in **top_theta** uno alla volta tutti i nuovi triangoli creati, riscalando il vettore in modo da mantenere inalterata la sua dimensione;

-questo tipo di reinserimento viene applicato esclusivamente quando è richiesta l'uniformità;

DivideTriangle_base

```
void DivideTriangle_base(){ ...}
```

Pseudo Codice

```
1
2  medio = MaxEdge().Medium();
3  newTriangle1 = Triangle({median1,split1,third1});
4  newTriangle2 = Triangle({median1,split2,third2});
5
6  if(exist(AdjacentTriangle)){
7      newTriangle3 = Triangle({median2,split1,third3});
8      newTriangle4 = Triangle({median2,split2,third4});
9  }
10 Update(TriangularMesh);
11 Update(AdjacenceMatrix);
12 Extract(CurrentTriangle, AdjacentTriangle);
13
```

La versione base consiste in una funzione semplice che opera su un singolo triangolo e il suo adiacente. Di conseguenza, verranno creati al massimo 4 nuovi triangoli, 2 per ciascuno dei triangoli originali.

Per evitare lo svuotamento dei vettori e della matrice di adiacenza dovuto alla cancellazione dei dati, si è optato per un oculato riutilizzo degli id (sia per i triangoli che per i lati).

DivideTriangle_advanced

```
void DivideTriangle_advanced(){ ...}
```

Pseudo Codice (1/2)

A differenza della versione base quella avanzata, si compone di due parti:

- La prima è uguale alla versione base, al netto di cambiamenti d'ordine;

```
1
2  medio = MaxEdge().Medium();
3  newTriangle1 = Triangle({median1,split1,third1});
4  newTriangle2 = Triangle({median1,split2,third2});
5
6  Update(TriangularMesh);
7  Update(AdjacenceMatrix);
8  Extract(CurrentTriangle);
9
10 if(exist(AdjacentTriangle)){
11   DivideRecursive(AdjacentTriangle);
12 }
13
```

- La seconda ha come differenza principale l'uso di una funzione ricorsiva:

```
void DivideTriangle_recursive(Triangle &T, Point p1, Edge &Split1, Point p2, Edge &Split2, Point &old_m){ ...}
```

analizziamola…

Pseudo codice (2/2)

- **Condizione di terminazione**: tratta il caso finale, ritornando all'istanza precedente;
- **Corpo**: divisione del triangolo e ricorsione sul triangolo adiacente;
- **Labor limae**: costruzione del lato mancante e dei relativi triangoli;

```
1
2  if(CurrentTriangle.MaxEdge()==OldTriangle.MaxEdge()){
3    newTriangle1 = Triangle({median,split1,third1});
4    newTriangle2 = Triangle({median,split2,third2});
5
6    Update(TriangularMesh); Update(AdjacenceMatrix);
7    Extract(CurrentTriangle);
8    return;
9  }
10 medio = CurrentTriangle.MaxEdge().Medium();
11 newTriangle1 = Triangle({median,split1,third1});
12 newTriangle2 = Triangle({median,split2,third2});
13
14 if(exist(AdjacentTriangle)){
15   DivideRecursive(AdjacentTriangle);
16   MtoM = LinkMiddlePoints();
17   newTriangle3 = Triangle({MtoM,median,oldSplit1});
18   newTriangle4 = Triangle({MtoM,split2,oldSplit2});
19
20   Update(TriangularMesh); Update(AdjacenceMatrix);
21   Extract(CurrentTriangle);
22 }
23
```

TEST

Testing…

```
    //TEST SORTING
TEST(TestSorting, TestMergeSortInc) {...}
TEST(TestSorting, TestMergeSortDec) {...}
    //TEST INSERT
TEST(TestInsert, TestInsertFalse) {...}
TEST(TestInsert, TestInsertTrue) {...}
    //TEST EDGE
TEST(TestEdge, TestCostructor) {...}
TEST(TestEdge, TestMedium) {...}
TEST(TestEdge, TestLength) {...}
TEST(TestEdge, TestIncludes) {...}
    //TEST TRIANGLE
TEST(TestTriangle, TestEqualTrue) {...}
TEST(TestTriangle, TestMaxEdge) {...}
TEST(TestTriangle, TestOrderEdges) {...}
TEST(TestTriangle, TestOrderPoints) {...}
TEST(TestTriangle, TestArea) {...}
TEST(TestTriangle, TestEqualFalse) {...}
TEST(TestTriangle, TestIncludesTrue) {...}
TEST(TestTriangle, TestIncludesFalse) {...}
TEST(TestTriangle, TestOppositeTrue) {...}
TEST(TestTriangle, TestOppositeError) {...}
TEST(TestTriangle, TestEdgesToPointsTrue) {...}
TEST(TestTriangle, TestEdgesToPointsFalse1) {...}
TEST(TestTriangle, TestEdgesToPointsFalse2) {...}
TEST(TestTriangle, TestPointsToEdgeTrue) {...}
TEST(TestTriangle, TestPointsToEdgeFalse) {...}
```

Applicazioni

L’esigenza dei test nasce dalla necessità di verificare la consistenza dei metodi implementati all’interno delle varie strutture dati, controllando la veridicità dei risultati.

Per oggetti più complessi, come la mesh triangolare, è stato utilizzato un controllo visivo tramite l’esportazione di file grafici (paraviewfile, vtk), in aggiunta ad un’attenta analisi con Debug, nonché alla costruzione di un Test0 semplificato.

Risultati

& considerazioni finali

ExportMesh

```
void ExportMesh(vector<short int> cells={0}, string all=""){ ...}
```

- **Parametri**:
  -*cells*: specifica i file associati alle celle da esportare;
  -*all*: shortcut per esportare tutti i file;
- **Funzionalità**: esporta i file scelti nella directory di destinazione personalizzata:

```
string cell0D = "./../Project/Dataset/Test"+to_string(test)+"Completed/"+level+"New0D"+uniformity+"_t"+to_string(percentage)+".csv";
```

Ad es. con **test**=1, **level**=“base”,**uniformity**=“uniform”,**percentage**=75%

```
string cell0D = "./../Project/Dataset/Test1Completed/base/New0Duniform_t75.csv";
```

ExportParaviewfile e ExportVTK

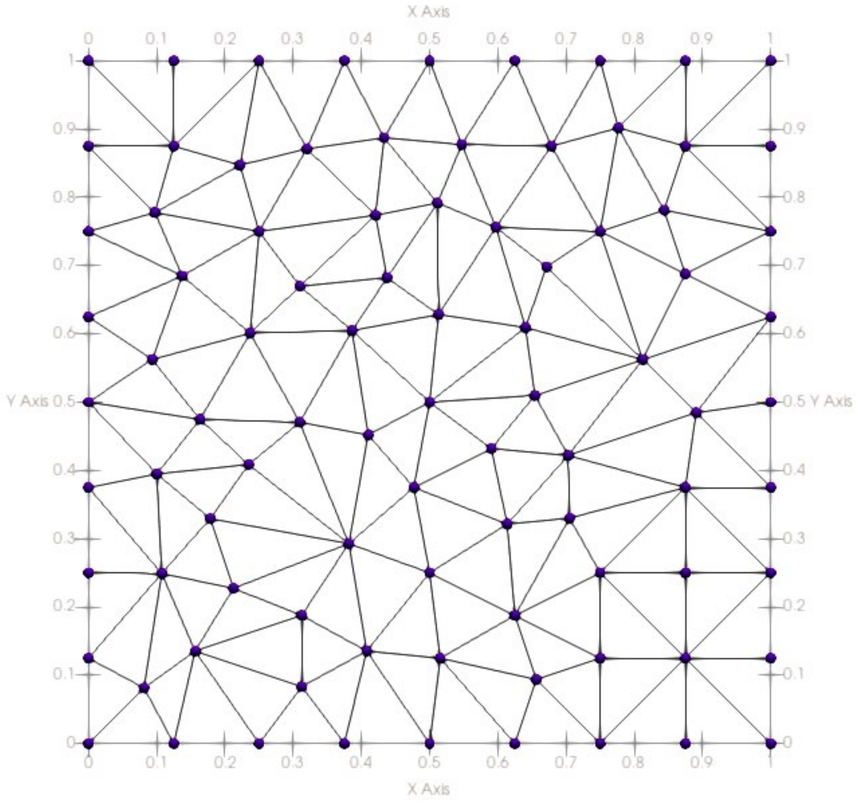
```
void TriangularMesh::ExportParaviewfile(){ ...}
void TriangularMesh::ExportVTK(){ ...}
```

Il primo esporta un file *.csv* completo di cell0D e cell1D che verrà utilizzato nel software di grafica Paraview, tramite aggiunta di appositi filtri.

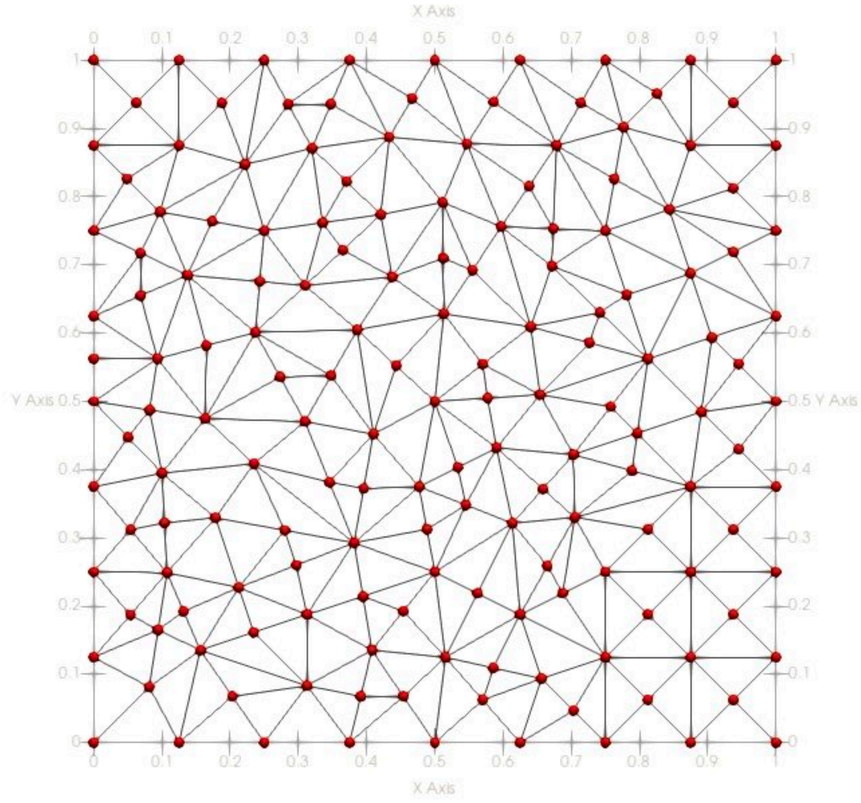
Il secondo, più generico e immediato, permette un'esportazione completa dei risultati senza la necessità di apporre alcun filtro.

Test 1 (1/2)

Prima



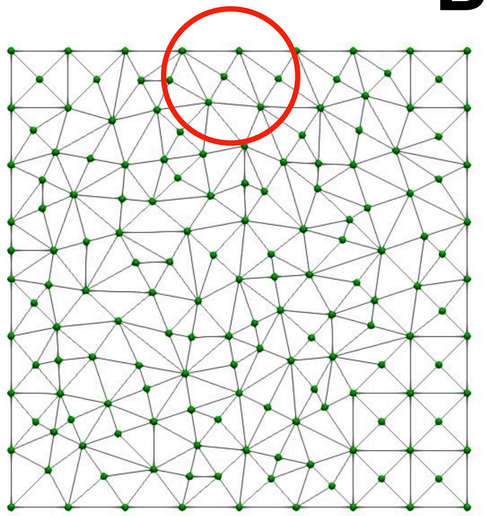
Dopo



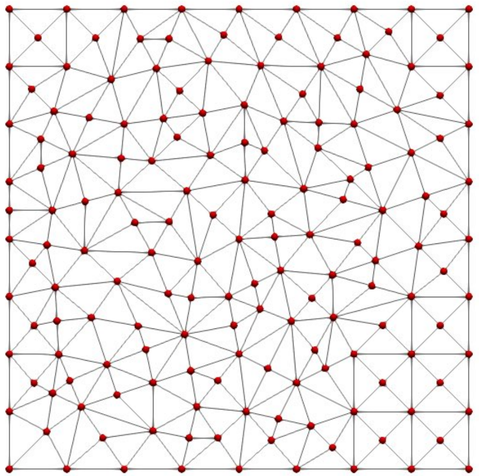
La mesh raffigurata è raffinata con una percentuale del 75%, in versione *“base”* e *“uniform”*

Test 1 (2/2)

Base

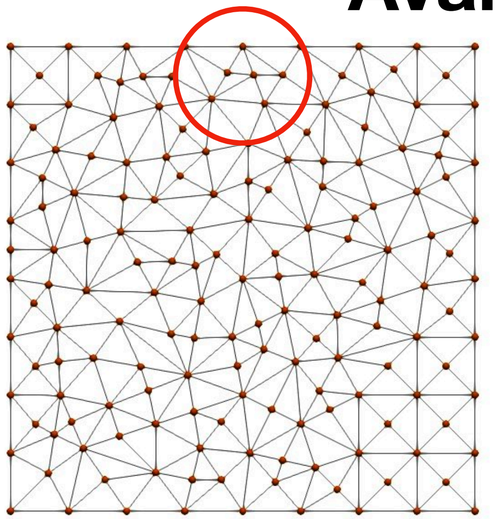


Non uniform

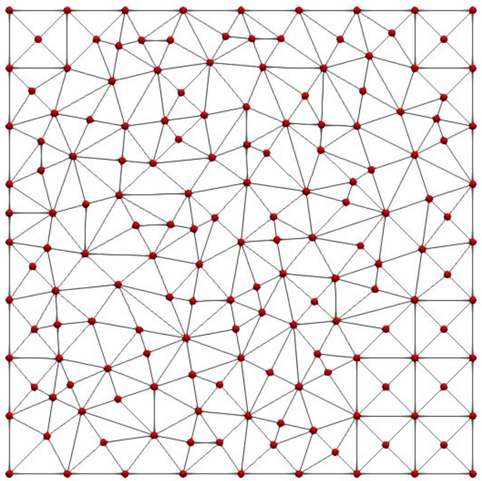


Uniform

Avanzato



Non uniform

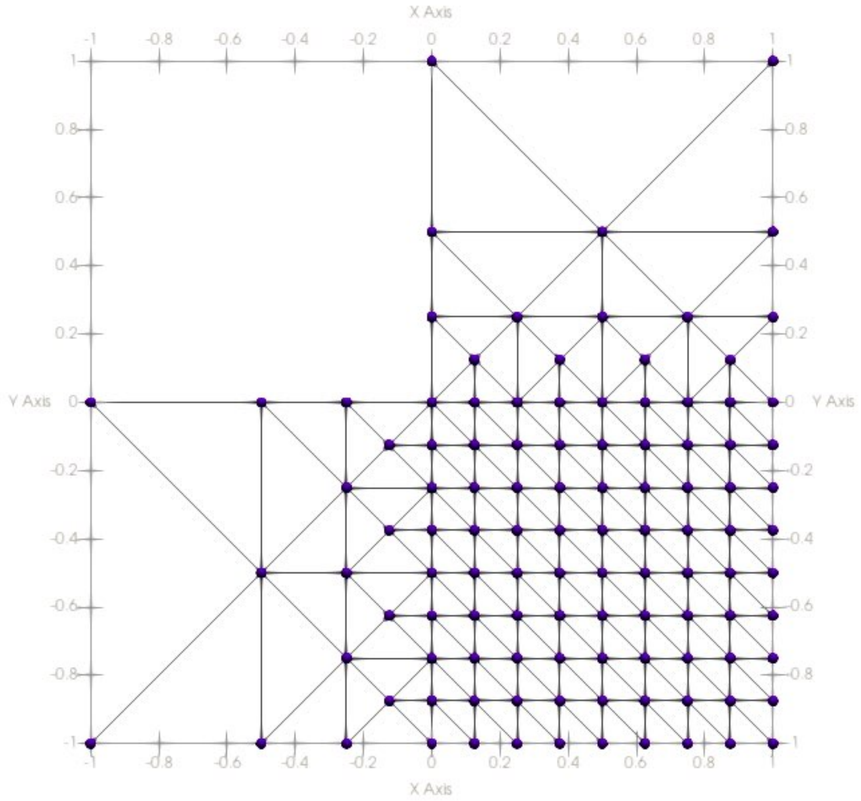


Uniform

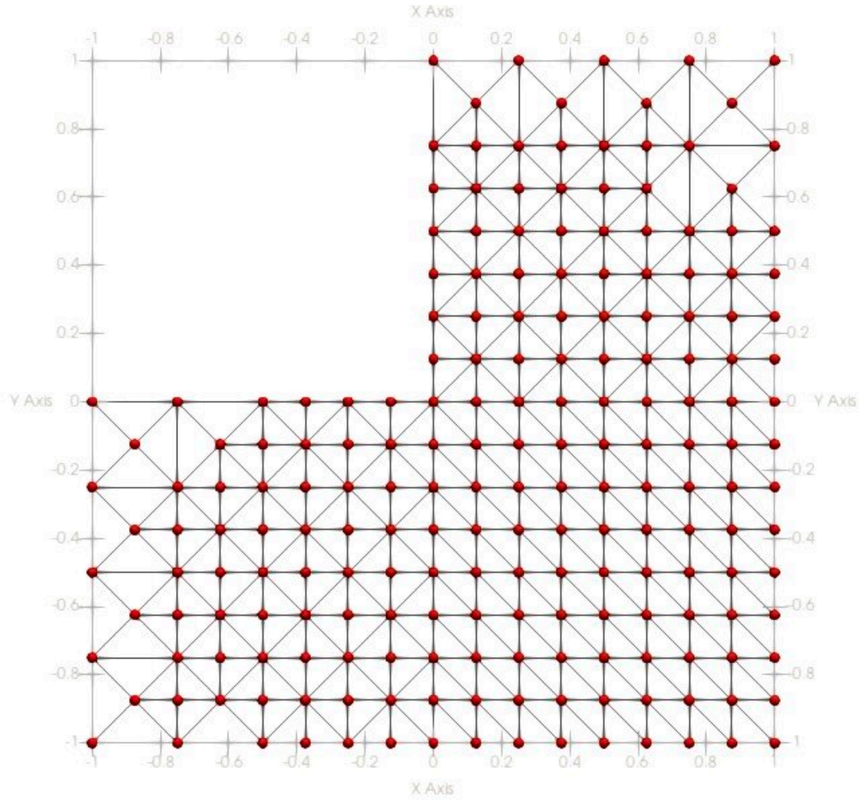
In questo **test** la differenza tra le versioni **uniforme** e **non** risulta minima, ciò è dovuto all'uniformità intrinseca della mesh.
Le differenze tra la versione **base** e quella **avanzata** risultano invece più apprezzabili.
Un esempio è rappresentato dalla zona cerchiata.

Test 2 (1/2)

Prima



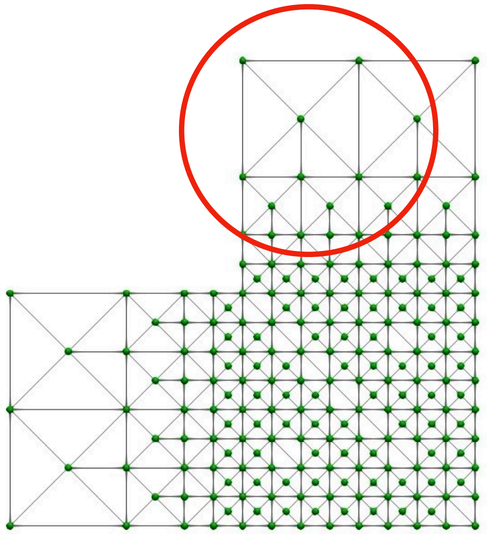
Dopo



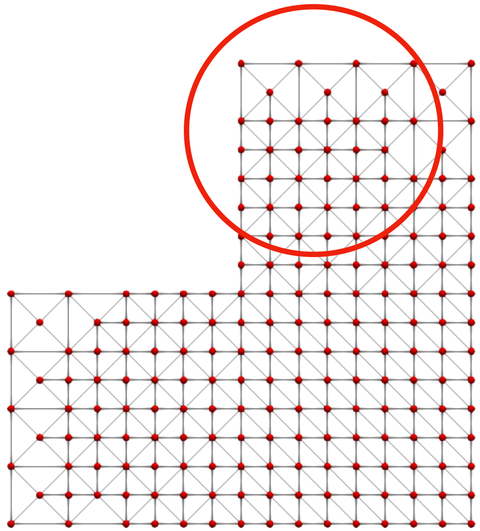
La mesh raffigurata è raffinata con una percentuale del 75%, in versione *“base”* e *“uniform”*

Test 2 (2/2)

Base

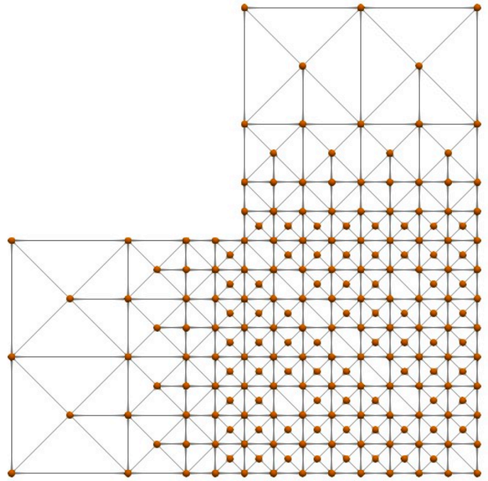


Non uniform

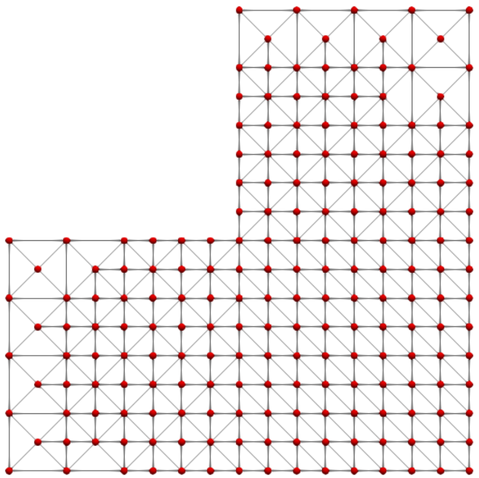


Uniform

Avanzato



Non uniform



Uniform

Contrariamente al **test** precedente, la marcata disuniformità della mesh, fa sì che risultino più visibili le differenze tra la versione **non uniforme** e quella **uniforme**, allo stesso tempo la particolare struttura simmetrica dei triangoli azzera i vantaggi della versione **avanzata**.

```
cout<<"Grazie per l'attenzione!"<<endl;
return 0;
```