© *J.-B.A.K. <jean-baptiste.apoung@math.u-psud.fr>*

Gestion de maillage

L'objectif de ce projet est de familiariser l'étudiant avec un outil incontournable du calcul scientifique : le maillage. Le but est d'explorer dans un langage de programmation accessible aux étudiants de L3, une des structures de données pour la gestion de ces maillages dans la modélisation des surfaces. C'est la structure de données dite **Half-Edge** *en anglais ou demi-arete en français. Pour une simplicité d'appréhension nous nous limitons aux surfaces planes en dimension 2.*

Unités d'enseignement requises : Math 312

Partie - 1 *Motivations et besoins*

Le maillage joue un rôle fondamental dans le calcul scientifique. Il permet en effet de décrire la géométrie du domaine sur lequel on souhaite résoudre l'équation aux dérivées partielles. Il est aussi beaucoup utilisé en infographie pour optimiser l'usage des ressources à disposition. Il est présent dans les logiciels de représentation graphique des scènes (jeux vidéo, scanner médical, ...), où les principales attentes sont la simplification des détails, le rendu en temps réel des scènes dynamiques, la représentation graphique des solutions approchées des équations aux dérivées partielles et bien d'autres.

Un maillage est essentiellement décrit par deux notions principales :

- La **topologie**. Qui permet entre autres d'identifier les entités présentes dans le maillage, que l'on classe suivant leur dimension ou codimension : en dimension deux d'espace, pour des maillages formés de triangles, la dimension suffit à cette classification. On distingue alors :
  - Les entités de dimension 0 : les sommets (*vertex* en anglais).
  - Les entités de dimension 1 : les arêtes (segment reliant deux sommets) (*edge* en anglais.)
  - Les entités de dimension 2 : les triangles.
- La **connectivité**. Ceci permet de décrire les relations qui existent entre les différentes entités présentes dans le maillage. Elle s'exprime par exemple sous la forme : *le triangle de numéro t_k est formé des sommets de numéros s_1, s_2, s_3 et partage l'arête numéro a_i avec son second triangle voisin de numéro t_n*. La figure 1 donne un exemple de maillage avec certaines de ces entités.

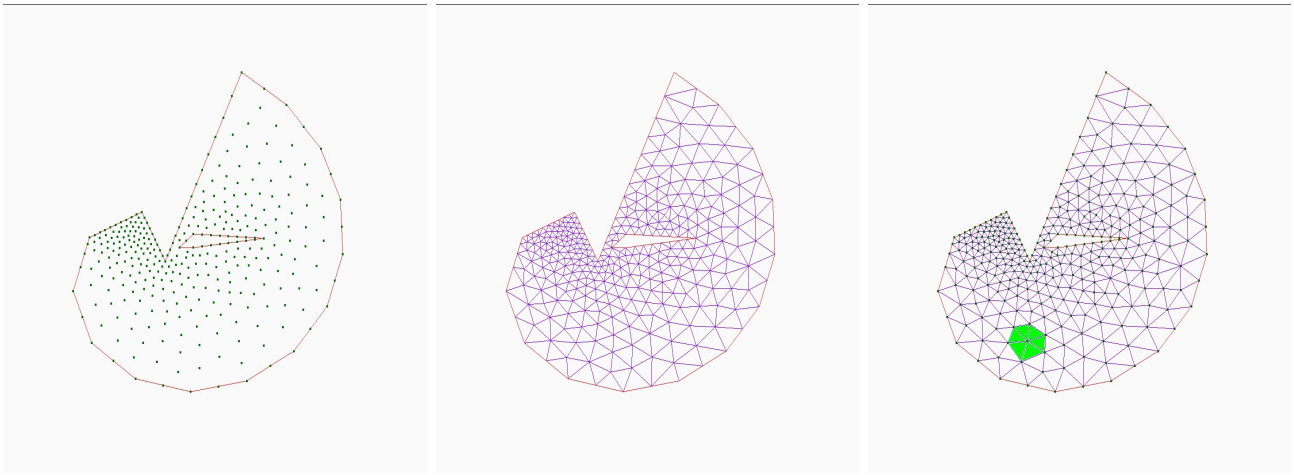

FIGURE 1 – Maillage et entités. De gauche à droite : Sommets, arêtes, triangles avec un sommet et sa boule.

Partie - 2 *Structure de données demi-arête*

Afin de manipuler ces maillages au niveau informatique, il faut être à même de les stocker. Rappelons en effet que ces maillages peuvent contenir des millions de points et de triangles : c'est le cas des images issues de la tomographie, où des reconstructions de surfaces génèrent des milliers de triangles tridimensionnels qui sont d'ailleurs en majorité de bien mauvaise qualité. Il est donc souhaitable de *nettoyer* (lisser) ces maillages d'une manière ou d'une autre bien souvent par des techniques relativement complexes. Sans nous appesantir sur ce point, soulignons que dans tous les cas un stockage optimal du maillage permettant un accès en temps *raisonnable* à toutes les entités qui le composent est nécessaire.

Pour cette fin, une bonne structure de données est nécessaire. La littérature présente plusieurs bien adaptées à gérer les maillages surfaciques. Le but du présent projet est d'examiner l'une d'elles appelée Half-Edge en anglais ou Demi-Arête en français pour ses qualités d'accès en temps constant au maximum d'informations.

Listons quelques unes des informations couramment recherchées, qui sont pour la plupart des informations de connectivité :

— La connectivité dite $(d) - (d)$. Pour toutes entités de dimension d, on cherche les entités de la même dimension qui lui sont rattachées (on dit aussi qui lui sont incidentes).
— La connectivité dite $(d) - (k)$. Pour toutes les entités de dimension d on cherche les entités de dimension (ou de codimension) k qui lui sont rattachées. Cette opération est symétrique au sens où elle s'accompagne de la recherche de la connectivité $(k) - (d)$.

Des exemples concrets sont : la recherche de tous les triangles contenant un sommet (connectivité 0-2) ou la recherche de toutes les arêtes contenant un point (connectivité 0-1), ou toutes les arêtes d'un triangle (connectivité 3-1) etc.

La structure de donnée Half-Edge semble requérir un minimum d'information à stocker pour accéder en temps constant au maximum des connectivités. Elle va du principe selon lequel, si l'on dédouble les arêtes du maillage de sorte à en faire deux demi-arêtes jumelles orientées en sens opposés, alors en introduisant les notions de sommet origine, de demi-arête suivante et de face (ou facette ou triangle) à gauche de la demi-arête, on peut réduire au mieux la quantité d'information à stocker pour décrire complètement le maillage, tout en rendant accessible en temps constant les informations de connectivité.

La suite du document décrit la mise en oeuvre informatique de cette structure de données, sous forme d'exercice et la dernière partie décrit les moyens de validation des implémentations mises en place.

Partie - 3 *Implémentation*

Q-3-1 : Fournir les types abstraits suivants :

— Une structure **DemiArete**. Pour gérer les arêtes du maillage.

Listing 1 – DemiArete

```
typedef struct DemiArete
{
  struct Sommet* m_origine;     /*< Sommet origine de la demi-arete           */
  struct DemiArete* m_suivant;  /*< Demi-arete suivante a la demi-arete       */
  struct DemiArete* m_jumeau;   /*< Demi-arete jumelle de la demi-arete       */
  struct Facette* m_face;       /*< Facette (triangle) contenant la demi-arete */
  int m_label;                  /*< Couleur associee a la demi-arete          */
}*pDemiArete;
```

— Une structure **Sommet** pour gérer les sommets du maillage.

Listing 2 – Sommet

```
typedef struct Sommet
```

```
{
 double m_x;          /* Abscisse du sommet             */
 double m_y;          /* Ordonnee du sommet             */
int m_label;          /* Couleur  du sommet             */
pDemiArete m_arete;   /* Demi-arete contenant le sommet */
}*pSommet;
```

— Une structure **Facette**. Pour gérer les faces (dans le présent cas les triangles) du maillage.

Listing 3 – Facette

```
typedef struct Facette
{
 pDemiArete m_arete; /* Une Demi-arete bordant la facette */
 int m_label;        /* Couleur de la facette             */
 }*pFacette;
```

— Une structure **Maillage**. Permettant de stocker l'ensemble des sommets, des arêtes et des triangles.

Listing 4 – Maillage

```
typedef
struct Maillage{
  pSommet m_sommets;   /*  Tableau des sommets              */
  pDemiArete m_aretes; /*  Tableau des demi-aretes          */
  pFacette m_faces;    /*  Tableau des facetes (triangles)  */
  int m_ns;            /*  Nombre de sommets                */
  int m_nf;            /*  Nombre de faces                  */
  int m_na;            /*  Nombre de demi-aretes            */
}*pMaillage;
```

Q-3-2 : Afin de faciliter la création de ces entités, fournir des fonctions d'allocation mémoire et de libération mémoire pour chacun de ces types représentés, de sorte que le code suivant permette d'allouer un tableau de ns sommets et de le libérer

Listing 5 – Acquisition et libération de ressources pour un Sommet

```
pSommet sommets = callocSommet(ns); /* creation d'un tableau de ns sommets      */
freeSommet(&sommets);               /* liberation des ressources des ressources */
```

Afin d'accéder rapidement aux différentes connectivités, il est parfois courant d'introduire la notion d'itérateur. Qui est une structure de données abstraite permettant de parcourir les éléments d'un en semble, en retournant l'élément courant si demandé.

Q-3-3 : Fournir les différents itérateurs suivants :

— **FaceIncidentesDeSommet**, pour parcourir toutes les faces contenant un sommet donné.
— **AretesIncidentesDeSomment**, pour parcourir toutes les demi-arêtes contenant un sommet donné.
— **AretesIncidentesDeFacette**, pour parcourir toutes les demi-arêtes décrivant une facette (triangle) donnée.
— **SommetsIncidentsDeSommet**, pour parcourir tous les sommets en contact (par le biais d'une demi-arête) avec un sommet donné.
— **SommetsIncidentsDeFacette**, pour parcourir tous les sommets décrivant une facette (triangle) donnée.

Voici un exemple de code permettant de parcourir toutes les arêtes incidentes à une facette.

Listing 6 – Illustration de l'utilisation de l' itérateur sur faces d'un sommet

```
Sommet e;
FaceIncidentesDeSommet iter;
debutFaceIncidentesDeSommet(&iter,&e);
while( iter.valeur ){
 /* on travaille avec iter.valeur */
suivantFaceIncidentesDeSommet(&iter);  /* iter.valeur passe au suivant */
}
```

Q-3-4 : Fournir des fonctions de lecture et d'écriture du maillage au format **.msh** de FreeFem++, voir annexe pour la description de ce format. On pourra utiliser le prototype suivant

Listing 7 – Lecture et écriture du maillage

```
void lireMaillage(pMaillage mesh, const char* nomFichier);
```

```
void ecrireMaillage(pMaillage mesh, const char* nomFichier);
```

On pourra aussi fournir des fonctions de lecture et d'écriture du maillage sous d'autres formats (**.off**, **.stl**, ...).

Partie - 4 *Validations*

Pour valider ces développements, on va considérer quelques problèmes courants.

Q-4-1 **: Recherche du triangle contenant un point.**

Fournir une fonction qui pour un point donné par ses coordonnées, détermine le triangle qui le contient. Calculer la complexité de l'algorithme implanté et représenter graphiquement à l'aide de l'outil graphique fourni les triangles visités pour accéder à ce point à partir du triangle de départ considéré.

Q-4-2 **: Analyse de convexité de la boule d'un point.** Fournir une fonction qui pour un sommet donné, retourne 1 si le polygone définit par sa boule de point est convexe, et 0 sinon.

Q-4-3 **: Qualité et amélioration de maillage**.

On appelle **qualité d'un triangle** la quantité définie par

$$Q_T = \alpha \frac{L_m P}{A}$$

où L_m est la longueur de la plus longue arête du triangle, P son demi-périmètre et A son aire. Le coefficient α est choisi de sorte que Q_T soit 1 pour un triangle équilatéral. Il en découle que la qualité d'un bon triangle est proche de 1 et celle d'un mauvais triangle tend vers l'infini. Ainsi pour éviter des erreurs d'arrondis, on préfère plutôt manipuler l'inverse de la qualité dont la valeur reste confinée en entre 0 et 1 : c'est-à-dire $0 \leq \frac{1}{Q_T} \leq 1$.

A partir de cette définition on déduit celle de la qualité d'un maillage, comme étant le maximum des qualités de ses triangles.

- Fournir une fonction qui calcule et retourne la qualité d'un triangle donné.
- En vue d'améliorer la qualité d'un maillage, on a quelque-fois recours à ce que l'on appelle **bougé de points**. Il consiste à déplacer un point (Sommet) dans sa boule de point de sorte à améliorer la qualité des triangles de cette boule. On le fait de manière itérative sur tous les sommets (**non frontières**) du maillage. Un exemple de bougé de point est donné par :

$$S^* = \frac{\sum_{T \in \text{boule}(S)} A_T \times G_T}{\sum_{T \in \text{boule}(S)} A_T}$$

  où S^* est la nouvelle position du sommet S, donné comme barycentre des centres de gravité (G_T) des triangles de sa boule de point, pondéré par l'aire (A_T) du triangle considéré.

Q-4-4 **: Vérification du caractère Delaunay du maillage**. Un maillage est dit de Delaunay lorsque le cercle circonscrit à chaque triangle de ce maillage ne contient aucun autre sommet du maillage. On caractérise aussi cette propriété en disant qu'on respecte le **critère de la sphère vide** (voir FIGURE 2).

Cette propriété joue un rôle fondamental dans certaines méthodes numériques de résolution d'équations aux dérivées partielles ou dans la théorie des classifications (nous ne rentrons pas dans les détails ici). En dimension 2, la théorie nous garantit que tout maillage formé de triangle peut-être rendu de Delaunay par une procédure appelée *edge swapping* en anglais ou bascule d'arête en français. Elle consiste à parcourir les triangles du maillage, et pour tout triangle qui viole le critère de Delaunay, on permute les diagonales du quadrilatère obtenu en joignant le triangle et le triangle voisin concerné.

- Fournir une fonction **swap22** qui permute les diagonales de deux triangles contigus ne respectant pas le critère de Delaunay pour générer deux nouveaux qui respectent le critère de Delaunay.
- Fournir une fonction **rendreDelaunay** qui prend un maillage, vérifie s'il est de Delaunay. Et s'il ne l'est pas y effectue des bascules d'arêtes pour qu'il le soit.

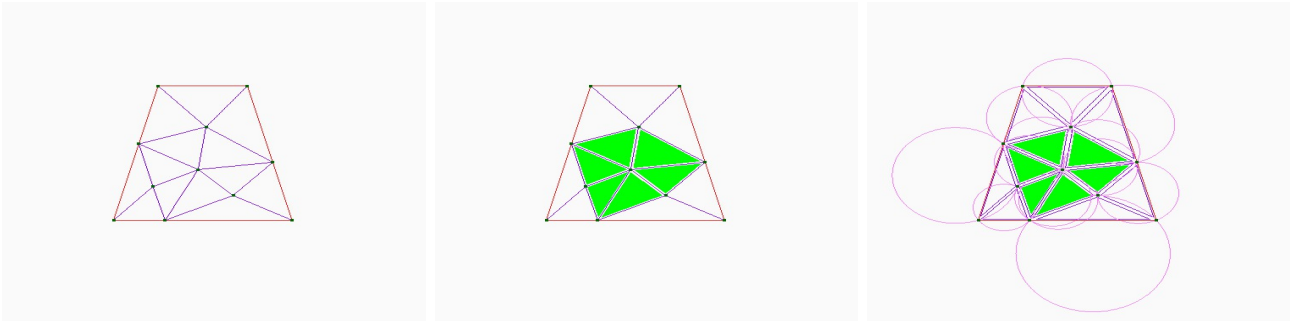

FIGURE 2 – De gauche à droite : maillage simple, boule d'un point, vérification du critère de Delaunay.

Partie - 5 *Utilitaires*

Ci-dessous figurent quelques utilitaires nécessaires pour simplifier la mise en oeuvre des algorithmes sus-mentionnés.

Listing 8 – Fonction lireMaillage

```
/* lireMaillage   Copyright (C) APOUNG KAMGA Jean-Baptiste  2013   */
/* */
pMaillage lireMaillage(const char* file){
  FILE* fid = fopen(file,"r");
  if(fid == NULL){
    printf("Erreur ouverture de fichier\n");
    exit(1);
  }

  int ns,nf,nba;
  fscanf(fid,"%d %d %d\n",&ns,&nf,&nba);
  printf("%d\t%d \t%d\n", ns,nf,nba);

  pMaillage p  = callocMaillage(ns,nf);
  int i;
  double x,y;
  int lab;
  int i0,i1,i2;
  int nfacei =0;
  int numf1,numf2,numf3;
  int hkey, num_jumeau;
  int hsize ;

  int* HashTable= createHTableKey(ns);
  hsize = ns;

  for(i = 0; i < ns; i++){
    fscanf(fid,"%lf %lf %d",&x,&y,&lab);
    fprintf(stdout,"%lf \t %lf \t %d\n",x,y,lab);
    p->m_sommets[i].m_x = x;
    p->m_sommets[i].m_y = y;
    p->m_sommets[i].m_label =lab;
  }

  for(i = 0; i < nf; i++){
   fscanf(fid,"%d %d %d %d",&i0,&i1,&i2,&lab);
   --i0; --i1;--i2;
   fprintf(stdout,"%d \t %d \t %d \t  %d\n",i0+1,i1+1,i2+1,lab);
   p->m_facettes[i].m_label = lab;

   numf1 = nfacei;
   numf2 = numf1+1;
   numf3 = numf2+1;

   p->m_aretes[numf1].m_origine = &p->m_sommets[i0];
   p->m_sommets[i0].m_arete = & p->m_aretes[numf1];
   p->m_aretes[nfacei].m_jumeau = NULL;
   p->m_aretes[numf1].m_face = &p->m_facettes[i];
   p->m_aretes[numf1].m_suivant = &p->m_aretes[numf2];
   p->m_aretes[numf1].m_label = 0;
   p->m_facettes[i].m_arete = &p->m_aretes[numf1];

    hkey = hashTableKeyFn(i0,i1,hsize);
   if(HashTable[hkey]!=-1) {
     num_jumeau = HashTable[hkey];
     p->m_aretes[num_jumeau].m_jumeau = &p->m_aretes[numf1];
     p->m_aretes[numf1].m_jumeau = &p->m_aretes[num_jumeau];
```

```
   }else{
     HashTable[hkey] =  numf1;
  }

   p->m_aretes[numf2].m_origine = &p->m_sommets[i1];
   p->m_sommets[i1].m_arete = & p->m_aretes[numf2];
   p->m_aretes[numf2].m_jumeau = NULL;
   p->m_aretes[numf2].m_face = &p->m_facettes[i];
   p->m_aretes[numf2].m_suivant = &p->m_aretes[numf3];
   p->m_aretes[numf2].m_label = 0;

   hkey = hashTableKeyFn(i1,i2,hsize);
   if(HashTable[hkey] != -1) {
     num_jumeau = HashTable[hkey];
     p->m_aretes[num_jumeau].m_jumeau = &p->m_aretes[numf2];
     p->m_aretes[numf2].m_jumeau = &p->m_aretes[num_jumeau];
   }else{
     HashTable[hkey] =  numf2;
  }
   p->m_aretes[numf3].m_origine = &p->m_sommets[i2];
   p->m_sommets[i2].m_arete = & p->m_aretes[numf3];
   p->m_aretes[numf3].m_jumeau = NULL;
   p->m_aretes[numf3].m_face = &p->m_facettes[i];
   p->m_aretes[numf3].m_suivant = &p->m_aretes[numf1];
   p->m_aretes[numf3].m_label = 0;

   hkey = hashTableKeyFn(i2,i0,hsize);
   if(HashTable[hkey] != -1) {
     num_jumeau = HashTable[hkey];
     p->m_aretes[num_jumeau].m_jumeau = &p->m_aretes[numf3];
     p->m_aretes[numf3].m_jumeau = &p->m_aretes[num_jumeau];
   }else{
     HashTable[hkey] =  numf3;
  }
   nfacei = numf3 + 1;
  }
  for(i = 0; i < nba; i++){
   fscanf(fid,"%d %d %d",&i0,&i1,&lab);
   --i0; --i1;
   fprintf(stdout,"%d \t %d \t %d\n",i0 + 1,i1 + 1,lab);
   // rechere de l'arete reposant sur i0,i1
   hkey = hashTableKeyFn(i0,i1,hsize);
    assert(hkey != -1);
    assert( p->m_aretes[HashTable[hkey]].m_jumeau == NULL);
   p->m_aretes[HashTable[hkey]].m_label = lab;
  }
  free(HashTable);
  fclose(fid);
  fprintf(stdout,"nombres aretes stockees %d\n",nfacei);
  return p;
}
```

Partie - 6 *Quelques illustrations*

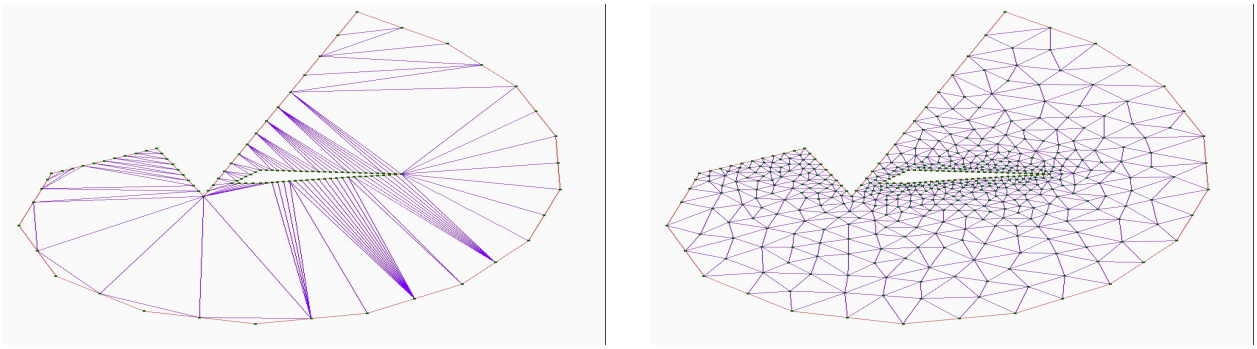

FIGURE 3 – A gauche : maillage vide (sans point intérieur). A droite : insertion de points sans optimisation.

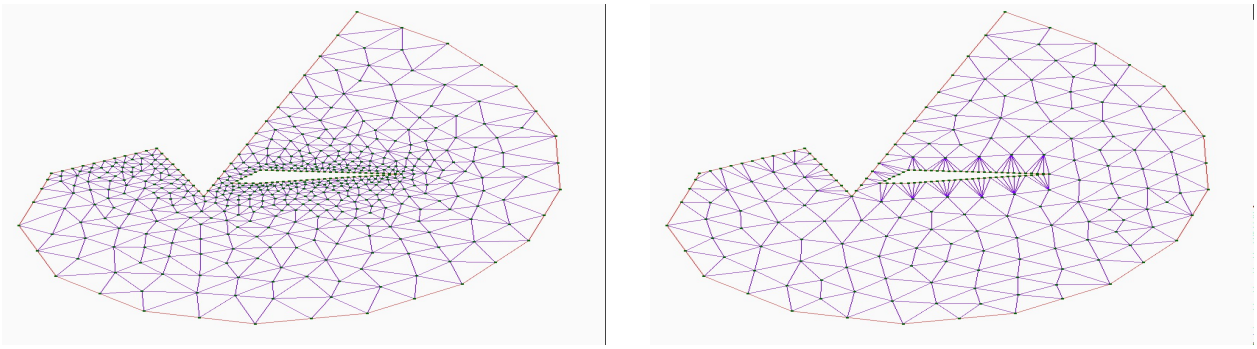

FIGURE 4 – A gauche : optimisation par bougé de points. A droite : maillage obtenu par décimation de points avec une procédure de fusion d'arêtes selon un critère de taille.

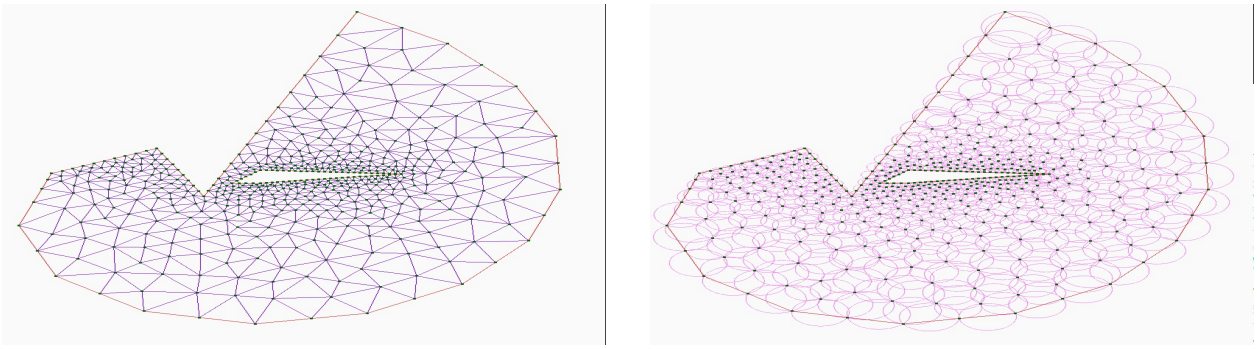

FIGURE 5 – A gauche : nouvelle insertion de points suivant un critère de qualité. A droite : vérification du critère de Delaunay.