Cours Maillage 2d et 3D C++

Pascal Frey, Frédéric Hecht

2 avril 2004

Table des matières

Chapitre 1

Chaînes et Chaînages

1.1 Introduction

Dans ce chapitre, nous allons décrire de manière formelle les notions de chaînes et de chaînages. Nous présenterons d'abord les choses d'un point de vue mathématique, puis nous montrerons par des exemples comment utiliser cette technique pour écrire des programmes très efficaces.

Rappelons qu'une chaîne est un objet informatique composée d'une suite de maillons. Un maillon, quand il n'est pas le dernier de la chaîne, contient l'information permettant de trouver le maillon suivant. Comme application fondamentale de la notion de chaîne, nous commencerons par donner une méthode efficace de construction de l'image réciproque d'une fonction.

Ensuite, nous utiliserons cette technique pour construire l'ensemble des arêtes d'un maillage, pour trouver l'ensemble des triangles contenant un sommet donné, et enfin pour construire la structure creuse d'une matrice d'éléments finis.

1.2 Construction de l'image réciproque d'une fonction

On va montrer comment construire l'image réciproque d'une fonction F. Pour simplifier l'exposé, nous supposerons que F est une fonction entière de $[0, n]$ dans $[0, m]$ et que ses valeurs sont stockées dans un tableau. Le lecteur pourra changer les bornes du domaine de définition ou de l'image sans grand problème.

Voici une méthode simple et efficace pour construire $F^{-1}(i)$ pour de nombreux i dans $[0, n]$, quand n et m sont des entiers raisonnables. Pour chaque valeur $j \in Im\, F \subset [0, m]$, nous allons construire la liste de ses antécédents. Pour cela nous utiliserons deux tableaux : `int head_F[m]` contenant les "têtes de listes" et `int next_F[n]` contenant la liste des éléments des $F^{-1}(i)$. Plus précisément, si i_1, i_2,...,$i_p \in [0, n]$, avec $p \geq 1$, sont les antécédents de j, `head_F[j]`=i_p, `next_F[`i_p`]`=i_{p-1}, `next_F[`i_{p-1}`]`=i_{p-2}, ..., `next_F[`i_2`]`=i_1 et `next_F[`i_1`]`= -1 (pour terminer la chaîne).

L'algorithme est découpé en deux parties : l'une décrivant la construction des tableaux

next_F et head_F, l'autre décrivant la manière de parcourir la liste des antécédents.

Algorithme 1.1

Construction de l'image réciproque d'un tableau

1. Construction :

```
int Not_In_Im_F = -1;
 for (int j=0;j<m;j++)
  head_F[j]=Not_In_Im_F;        //   initialement, les listes
                               //   des antécédents sont vides
 for (int i=0;i<n;i++)
  j=F[i],next_F[i]=head_F[j],head_F[j]=i; //   chaînage amont
```

2. Parcours de l'image réciproque de j dans $[0, n]$:

```
 for (int i=head_F[j];i!=Not_In_Im_F;i=next_F[i])
  { assert(F(i)==j);       //   j doit être dans l'image de i
                                          //   ... votre code
  }
```

Exercice 1 *Le pourquoi est laissé en exercice.*

1.3 Construction des arêtes d'un maillage

Rappelons qu'un maillage est défini par la donnée d'une liste de points et d'une liste d'éléments (des triangles par exemple). Dans notre cas, le maillage triangulaire est implémenté dans la classe `Mesh` qui a deux membres `nv` et `nt` respectivement le nombre de sommets et le nombre de triangle, et qui a l'opérateur fonction `(int j,int i)` qui retourne le numero de du sommet `i` du triangle `j`. Cette classe `Mesh` pourrait être par exemple :

```
class Mesh { public:
 int nv,nt;                              //   nb de sommet, nb de triangle
 int (* nu)[3];                                    //   connectivité
 int (* c)[3];                                  //   coordonnées de sommet
 int operator()(int i,int j) const { return nu[i][j];}
 Mesh(const char * filename);                   //   lecture d'un maillage

}
```

Dans certaines applications, il peut être utile de construire la liste des *arêtes du maillage*, c'est-à-dire l'ensemble des arêtes de tous les éléments. La difficulté dans ce type de construction réside dans la manière d'éviter – ou d'éliminer – les doublons (le plus souvent une arête appartient à deux triangles).

Nous allons proposer deux algorithmes pour déterminer la liste des arêtes. Dans les deux cas, nous utiliserons le fait que les arêtes sont des segments de droite et sont donc définies complètement par la donnée des numéros de leurs deux sommets. On stockera donc les arêtes dans un tableau `arete[nbex][2]` où `nbex` est un majorant du nombre total d'arêtes. On pourra prendre grossièrement `nbex = 3*nt` ou bien utiliser la formule d'Euler en 2D

$$\texttt{nbe} = \texttt{nt} + \texttt{nv} + \texttt{nb\_de\_trous} - \texttt{nb\_composantes\_connexes}, \tag{1.1}$$

où `nbe` est le nombre d'arêtes (*edges* en anglais), nt le nombre de triangles et nv le nombre de sommets (*vertices* en anglais).

La première méthode est la plus simple : on compare les arêtes de chaque élément du maillage avec la liste de *toutes* les arêtes déjà répertoriées. Si l'arête était déjà connue on l'ignore, sinon on l'ajoute à la liste. Le nombre d'opérations est $nbe * (nbe + 1)/2$.

Avant de donner le première algorithme, indiquons qu'on utilisera souvent une petite routine qui échange deux paramètres :

```
template<class T> inline void Exchange (T& a,T& b) {T c=a;a=b;b=c;}
```

Algorithme 1.2

Construction lente des arêtes d'un maillage $\mathcal{T}_h$

```
ConstructionArete(const Mesh & Th,int  (* arete)[2],int &nbe)
{
  int SommetDesAretes[3][2] = { {0,1},{1,2},{2,0}};
  nbe = 0;                                  //    nombre d'arête;
  for(int t=0;t<Th.nt;t++)
   for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]);
      int  j= Th(t,SommetDesAretes[et][1]);
      if (j < i) Exchange(i,j)        //    on oriente l'arête
      bool existe =false;  //    l'arête n'existe pas a priori
       for (int e=0;e<nbe;e++) //   pour les arêtes existantes
        if (arete[e][0] == i && arete[e][1] == j)
          {existe=true;break;}          //    l'arête est déjà
construite
      if (!existe)                          //    nouvelle arête
        arete[nbe][0]=i,arete[nbe++][1]=j;}
}
```

Cet algorithme trivial est bien trop cher dès que le maillage a plus de 500 sommets (plus de 1.000.000 opérations). Pour le rendre de l'ordre du nombre d'arêtes, on va remplacer la boucle sur l'ensemble des arêtes construites par une boucle sur l'ensemble des arêtes ayant le même plus petit numéro de sommet. Dans un maillage raisonnable, le nombre d'arêtes incidentes sur un sommet est petit, disons de l'ordre de six, le gain est donc important : nous obtiendrons ainsi un algorithme en $3 \times nt$.

Pour mettre cette idée en oeuvre, nous allons utiliser l'algorithme de parcours de l'image réciproque de la fonction qui à une arête associe le plus petit numéro de ses sommets. Autrement dit, avec les notations de la section précédente, l'image par l'application F d'une arête sera le minimum des numéros de ses deux sommets. De plus, la construction et l'utilisation des listes, c'est-à-dire les étapes 1 et 2 de l'algorithme 1.1 seront simultanées.

Construction rapide des arêtes d'un maillage $\mathcal{T}_h$

Algorithme 1.3

```
ConstructionArete(const Mesh & Th,int  (* arete)[2]
                  ,int &nbe,int nbex) {
   int SommetDesAretes[3][2] = { {0,1},{1,2},{2,0}};
   int end_list=-1;
   int * head_minv = new int [Th.nv];
   int * next_edge = new int [nbex];

   for ( int i =0;i<Th.nv;i++)
     head_minv[i]=end_list;                    //   liste vide

   nbe = 0;                                  //    nombre d'arête;

   for(int t=0;t<Th.nt;t++)
    for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]);      //    premier
sommet;
      int  j= Th(t,SommetDesAretes[et][1]);       //    second
sommet;
      if (j < i) Exchange(i,j)         //    on oriente l'arête
      bool existe =false;  //    l'arête n'existe pas a priori
       for (int e=head_minv[i];e!=end_list;e = next_edge[e] )
                  //    on parcourt les arêtes déjà construites
        if ( arete[e][1] == j)           //    l'arête est déjà
construite
          {existe=true;break;}                     //    stop
      if (!existe) {                       //    nouvelle arête
        assert(nbe < nbex);
        arete[nbe][0]=i,arete[nbe][1]=j;
                                  //    génération des chaînages
        next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;}
      }
  delete []  head_minv;
  delete []  next_edge;
}
```

Preuve : la boucle `for(int e=head_minv[i];e!=end_list;e=next_edge[e])` permet de parcourir toutes des arêtes (i, j) orientées $(i < j)$ ayant même i, et la ligne :

```
next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;
```

permet de chaîner en tête de liste des nouvelles arêtes. Le `nbe++` incrémente pour finir le nombre d'arêtes. ∎

Exercice 2 *Il est possible de modifier l'algorithme précédent en supprimant le tableau* `next_edge` *et en stockant les chaînages dans* `arete[i][0]`*, mais à la fin, il faut faire une boucle de plus sur les sommets pour reconstruire* `arete[.][0]`*.*

Exercice 3 *Construire le tableau* `adj` *d'entier de taille* `3 nt` *qui donne pour l'arete* `i` *du triangle* `k`*.*

- *si cette arete est interne alors* `adj[i+3k]=ii+3kk` *où est l'arete* `ii` *du triangle* `kk`*, remarquons :* `ii=adj[i+3k]%3, kk=adj[i+3k]/3`*.*
- *sinon* `adj[i+3k]=-1`*.*

1.4 Construction des triangles contenant un sommet donné

La structure de données classique d'un maillage permet de connaître directement tous les sommets d'un triangle. En revanche, déterminer tous les triangles contenant un sommet n'est pas immédiat. Nous allons pour cela proposer un algorithme qui exploite à nouveau la notion de liste chaînée.

Rappelons que si `Th` est une instance de la class `Mesh` (voir 1.3), `i=Th(k,j)` est le numéro global du sommet `j`$\in [0,3[$ de l'élément `k`. L'application F qu'on va considérer associe à un couple (k,j) la valeur `i=Th(k,j)`. Ainsi, l'ensemble des numéros des triangles contenant un sommet `i` sera donné par les premières composantes des antécédents de `i`.

On va utiliser à nouveau l'algorithme 1.1, mais il y a une petite difficulté par rapport à la section précédente : les éléments du domaine de définition de F sont des *couples* et non plus simplement des entiers. Pour résoudre ce problème, remarquons qu'on peut associer de manière unique au couple `(k,j)`, où $j \in [0,m[$, l'entier `p(k,j)=k*m+j` [1]. Pour retrouver le couple `(k,j)` à partir de l'entier `p`, il suffit d'écrire que `k` et `j` sont respectivement le quotient et le reste de la division euclidienne de `p` par `m`, autrement dit :

$$\mathtt{p} \longrightarrow (\mathtt{k},\mathtt{j}) = (\mathtt{k} = \mathtt{p}/\mathtt{m}, \mathtt{j} = \mathtt{p}\%\mathtt{m}). \tag{1.2}$$

Voici donc l'algorithme pour construire l'ensemble des triangles ayant un sommet en commun :

Algorithme 1.4

Construction de l'ensemble des triangles ayant un sommet commun
Préparation :

```
int end_list=-1,
int *head_s = new int[Th.nv];
int *next_p = new int[Th.nt*3];
int i,j,k,p;
for (i=0;i<Th.nv;i++)
  head_s[i] = end_list;
for (k=0;k<Th.nt;k++)                    //    forall triangles
  for(j=0;j<3;j++) {
    p = 3*k+j;
    i = Th(k,j);
    next_p[p]=head_s[i];
    head_s[i]= p;}
```

Utilisation : parcours de tous les triangles ayant le sommet numéro `i`

```
  for (int p=head_s[i]; p!=end_list; p=next_p[p])
  { int k=p/3,j = p % 3;
    assert( i == Th(k,j));
                                                 //    votre code
  }
```

Exercice 4 *Optimiser le code en initialisant* `p =-1` *et en remplaçant* `p = 3*j+k` *par* `p++`.

[1] Noter au passage que c'est ainsi que `C++` traite les tableaux à double entrée : un tableau `T[n][m]` est stocké comme un tableau à simple entrée de taille `n*m` dans lequel l'élément `T[k][j]` est repéré par l'indice `p(k,j) = k*m+j`.

1.5 Construction de la structure d'une matrice morse

Il est bien connu que la méthode des éléments finis conduit à des systèmes linéaires associés à des matrices très *creuses*, c'est-à-dire contenant un grand nombre de termes nuls. Dès que le maillage est donné, on peut construire le graphe des coefficients *a priori* non nuls de la matrice. En ne stockant que ces termes, on pourra réduire au maximum l'occupation en mémoire et optimiser les produits matrices/vecteurs.

1.5.1 Description de la structure morse

La structure de données que nous allons utiliser pour décrire la matrice creuse est souvent appelée "matrice morse" (en particulier dans la bibliothèque MODULEF), dans la littérature anglo-saxonne on trouve parfois l'expression "Compressed Row Sparse matrix" (cf. SIAM book...). Notons `n` le nombre de lignes et de colonnes de la matrice, et `nbcoef` le nombre de coefficients non nuls *a priori*. Trois tableaux sont utilisés : `a[k]` qui contient la valeur du k-ième coefficient non nul avec $k \in [0, nbcoef[$, `ligne[i]` qui contient l'indice dans `a` du premier terme de la ligne `i+1` de la matrice avec $i \in [-1, n[$ et enfin `colonne[k]` qui contient l'indice de la colonne du coefficient `k`$\in$`[0 :nbcoef[`. On va de plus supposer ici que la matrice est symétrique, on ne stockera donc que sa partie triangulaire inférieure. En résumé, on a :

$$\texttt{a[k]} = a_{\texttt{ij}} \quad \text{pour } \texttt{k} \in [\texttt{ligne[i-1]}+\texttt{1}, \texttt{ligne[i]}] \quad \text{et } \texttt{j} = \texttt{colonne[k]} \quad \text{si } \texttt{i} \leq \texttt{j}$$

et s'il n'existe pas de k pour un couple `(i,j)` ou si `i` $\leq$ `j` alors $a_{\texttt{ij}} = 0$.

La classe décrivant une telle structure est :

```
 class MatriceMorseSymetrique {
  int n,nbcoef;    //    dimension de la matrice et nombre de coefficients non
nuls
  int *ligne,* colonne;
  double *a;
  MatriceMorseSymetrique(Maillage & Th);                    //    constructeur
}
```

Exemple : on considère la partie triangulaire inférieure de la matrice d'ordre 10 suivante (les valeurs sont les rangs dans le stockage et non les coefficients de la matrice) :

$$\begin{pmatrix}
0 & . & . & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . \\
. & 2 & 3 & . & . & . & . & . & . & . \\
. & 4 & 5 & 6 & . & . & . & . & . & . \\
. & . & 7 & . & 8 & . & . & . & . & . \\
. & . & . & 9 & 10 & 11 & . & . & . & . \\
. & . & 12 & . & 13 & . & 14 & . & . & . \\
. & . & . & . & . & 15 & 16 & 17 & . & . \\
. & . & . & . & 18 & . & . & . & 19 & . \\
. & . & . & . & . & . & . & . & . & 20
\end{pmatrix}$$

On numérote les lignes et les colonnes de `[0..9]`. On a alors :

```
n=10,nbcoef=20,

ligne[-1:9] = {-1,0,1,3,6,8,11,14,17,19,20};

colonne[21] ={0, 1,  1,2,  1,2,3,  2,4,  3,4,5,  2,4,6,

              5,6,7,  4,8,  9};

a[21]                      //    ... valeurs des 21 coefficients de la matrice
```

1.5.2 Construction de la structure morse par coloriage

Nous allons maintenant construire la structure morse d'une matrice symétrique à partir de la donnée d'un maillage d'éléments finis P_1. Pour construire la ligne i de la matrice, il faut trouver tous les sommets j tels que i, j appartiennent à un même triangle. Ainsi, pour un noeud donné i, il s'agit de lister les sommets appartenant aux triangles contenant i. Le premier ingrédient de la méthode sera donc d'utiliser l'algorithme 1.4 pour parcourir l'ensemble des triangles contenant i. Mais il reste une difficulté : il faut éviter les doublons. Nous allons pour cela utiliser une autre technique classique de programmation qui consiste à "colorier" les coefficients déjà répertoriés : pour chaque sommet i (boucle externe), on effectue une boucle interne sur les triangles contenant i puis on balaie les sommets j de ces triangles en les coloriant pour éviter de compter plusieurs fois les coefficients a_{ij} correspondant. Si on n'utilise qu'une couleur, on doit "démarquer" les sommets avant de passer à un autre i. Pour éviter cela, on va utiliser plusieurs couleurs, et on changera de couleur de marquage à chaque fois qu'on changera de sommet i dans la boucle externe.

Construction de la structure d'une matrice morse

Algorithme 1.5

```
#include "MatMorse.hpp"
MatriceMorseSymetrique::MatriceMorseSymetrique(const  Mesh &
Th)
{
  int color=0, * mark;
  int i,j,jt,k,p,t;
  n = m = Th.nv;
  mark = new int [n];
         //    construction optimisée de l'image réciproque de
Th(k,j)
  int end_list=-1,*head_s,*next_p;
  head_s = new int [Th.nv];
  next_p = new int [Th.nt*3];
  p=0;
  for (i=0;i<Th.nv;i++)
    head_s[i] = end_list;
  for (k=0;k<Th.nt;k++)
    for(j=0;j<3;j++)
      next_p[p]=head_s[i=Th(k,j)], head_s[i]=p++;

                    //    initialisation du tableau de couleur
  for(j=0;j<Th.nv;j++)
    mark[j]=color;
  color++;

     //   1) calcul du nombre de coefficients a priori non-nuls
de la matrice
  nbcoef = 0;
  for(i=0; i<n; i++,color++,nbcoef++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
      for(jt=0; jt< 3; jt++ )
if ( i <= (j=Th(t,jt) )  && (mark[j]!= color))
  mark[j]=color,nbcoef++;          //    nouveau coefficient =>
marquage + ajout
                                  //    2) allocations mémoires
  ligne.set(new int [n+1],n+1);

  colonne.set( new int [ nbcoef],nbcoef);
  a.set(new double [nbcoef],nbcoef);
     //    3) constructions des deux tableaux ligne et colonne
  ligne[0] = -1;
  nbcoef =0;
  for(i=0; i<n; ligne[++i]=nbcoef, color++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
      for(jt=0; jt< 3; jt++ )
        if ( i <= (j=Th(t,jt))  && (mark[j]!= color))
  mark[j]=color, colonne[nbcoef++]=j;            //    nouveau
coefficient => marquage + ajout
                  //    4) tri des lignes par index de colonne
  for(i=0; i<n; i++)
    HeapSort(colonne + ligne[i] + 1 ,ligne[i+1] - ligne[i]);
                                                 //    nettoyage
  delete  [] head_s;
  delete  [] next_p;
}
```

Au passage, nous avons utilisé la fonction `HeapSort` qui implémente un petit algorithme de tri, présenté dans [Knuth-1975], qui a la propriété d'ête toujours en $nlog_2n$ (*cf.* code ci-dessous). Noter que l'étape de tri n'est pas absolument nécessaire, mais le fait d'avoir des lignes triées par indice de colonne permet d'optimiser l'accés à un coefficient de la matrice dans la structure creuse.

```
template<class T>
void  HeapSort(T *c,long n) {
  c-;             //    because fortran version aray begin at 1 in the routine
  register long m,j,r,i;
  register T crit;
  if( n <= 1) return;
  m = n/2 + 1;
  r = n;
  while (1) {
    if(m <= 1 ) {
      crit = c[r];
      c[r-] = c[1];
    if ( r == 1 ) { c[1]=crit; return;}
    } else  crit = c[-m];
    j=m;
    while (1) {
      i=j;
      j=2*j;
      if  (j>r) {c[i]=crit;break;}
      if ((j<r) && c[j] < c[j+1])) j++;
      if (crit < c[j]) c[i]=c[j];
      else {c[i]=crit;break;}
    }
  }
}
```

Remarque : Si vous avez tout compris dans ces algorithmes, vous pouvez vous attaquer à la plupart des problèmes de programmation.

Chapitre 2

Construction d'un maillage bidimensionnel

Nous nous proposons de présenter dans ce chapitre les notions théoriques et pratiques nécessaires pour écrire un générateur de maillage (*mailleur*) bidimensionnel de type Delaunay-Voronoï, simple et rapide.

2.1 Bases théoriques

2.1.1 Notations

1. Le segment fermé (respectivement ouvert) d'extrémités a, b de $\mathbb{R}^d$ est noté $[a, b]$ (respectivement $]a, b[$).
2. Un ensemble convexe C est tel que $\forall (a, b) \in \mathbb{C}^2, [a, b] \subset C$.
3. Le convexifié d'un ensemble S de points de $\mathbb{R}^d$ est noté $\mathcal{C}(S)$ est le plus petit convexe contenant S et si l'ensemble est fini (*i.e.* $S = \{x_i, i = 1, .., n\}$) alors nous avons :

$$\mathcal{C}(S) = \left\{ \sum_{i=1}^{n} \lambda_i x_i : \quad \forall (\lambda_i)_{i=1,..,n} \in \mathbb{R}_+^n, \text{ tel que } \sum_{i=1}^{n} \lambda_i \leq 1 \right\}$$

   .
4. Un ouvert Ω est polygonal si le bord $\partial\Omega$ de cet ouvert est formé d'un nombre fini de segments.
5. L'adhérence de l'ensemble O est notée $\overline{O}$.
6. L'intérieur de l'ensemble F est noté $\overset{\circ}{F}$.
7. un n-simplex $(x_0, ..., x_n)$ est le convexifié des $n + 1$ points de $\mathbb{R}^d$ affine independant (donc $n \leq d$) ,
   - une arête ou un segment est un 1-simpex,
   - un triangle est un 2-simplex,
   - un trétraédre est un 3-simplex ;

   les $p - simplex$ d'un $k - simplex$ sont formés avec $p + 1$ points de $(x_0, ..., x_d)$. Les ensembles de $k - simplex$
   - des sommets sera l'ensemble des $k + 1$ points (0-simplex),
   - des arêtes sera l'ensemble des $\frac{(k+1)\times k}{2}$ 1-simplex ,

- des triangles sera l'ensemble des $\frac{(k+1)\times k}{2}$ 2-simplex ,
- des faces sera l'ensemble des $\frac{(k+1)\times k}{2}$ (d-1)-simplex ,
-

8. le graphe d'une fonction $f : E \longleftarrow F$ est l'ensemble les points de $(x, f(x)) \in E \times F$.

2.1.2 Introduction

Commençons par définir la notion de maillage simplexial.

Définition 2.1 *Un maillage simplexial $\mathcal{T}_{d,h}$ d'un ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^d$ est un ensemble de $d-$ simplex K^k de $\mathbb{R}^d$ pour $k = 1, N_t$ (triangle si $d = 2$ et tétraèdre si $d = 3$), tel que l'intersection de deux d-simplex distincts $\overline{K}^i, \overline{K}^j$ de $\mathcal{T}_{d,h}$ soit :*
- *l'ensemble vide,*
- *ou p-simplex commun à K et K' avec $p \leq d$*

Le maillage $\mathcal{T}_{d,h}$ couvre l'ouvert défini par :

$$\mathcal{O}_h \stackrel{def}{=} \overline{\bigcup_{K\in\mathcal{T}_{d,h}}^{\circ} K} \tag{2.1}$$

De plus, $\mathcal{T}_{0,h}$ désignera l'ensemble des sommets de $\mathcal{T}_{d,h}$ et $\mathcal{T}_{1,h}$ l'ensemble des arêtes de $\mathcal{T}_{d,h}$ et l'ensemble de faces serat $\mathcal{T}_{d-1,h}$. Le bord $\partial\mathcal{T}_{d,h}$ du maillage $\mathcal{T}_{d,h}$ est défini comme l'ensemble des faces qui ont la propriété d'appartenir à un unique d-simplex de $\mathcal{T}_{d,h}$. Par conséquent, $\partial\mathcal{T}_{d,h}$ est un maillage du bord $\partial\mathcal{O}_h$ de $\mathcal{O}_h$. Par abus de langage, nous confondrons une arête d'extrémités (a, b) et le segment ouvert $]a, b[$, ou fermé $[a, b]$.

Remarque 2.1 *Les triangles sont les composantes connexes de*

$$\mathcal{O}_h \setminus \bigcup_{(a,b)\in\mathcal{T}_{1,h}} [a, b]. \tag{2.2}$$

Commençons par donner un théorème fondamental en dimension $d = 2$.

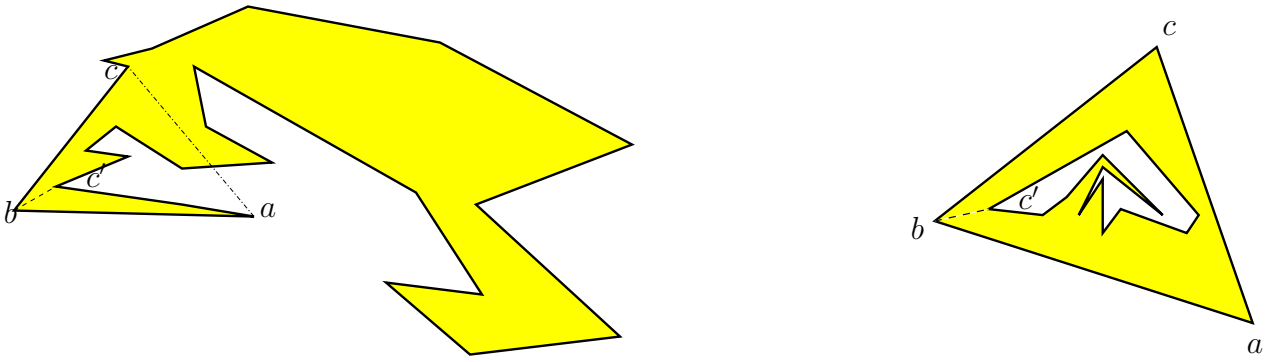
Théorème 2.1 *Pour tout ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^2$, il existe un maillage de cet ouvert sans sommet interne.*

Les sommets de ce maillage sont les points anguleux du bord $\partial\mathcal{O}_h$.

La démonstration de ce théorème utilise le lemme suivant :

Lemme 2.1 *Dans un ouvert polygonal $\mathcal{O}$ connexe qui n'est pas un triangle, il existe deux points anguleux α, β tels que $]\alpha, \beta[\subset \mathcal{O}$.*

Preuve : il existe trois points anguleux a, b, c consécutifs du bord $\partial\mathcal{O}$ tel que l'ouvert soit localement du côté droite de $]a, b[$ et tel que l'angle $\widehat{abc} < \pi$.



Il y a deux cas :
– si $]a, c[\subset \mathcal{O}_h$, il suffit de prendre $]a, c[=]\alpha, \beta[$;
– si $]a, b[$ n'est pas inclus dans $\mathcal{O}_h$, donc l'intersection du bord $\partial\mathcal{O}_h$ avec le triangle ouvert abc n'est pas vide car $\mathcal{O}_h$ n'est pas un triangle. Soit $\mathcal{C}$ le convexifié de cette intersection. Par construction, ce convexifié $\mathcal{C}$ ne touche pas $]a, b[$ et $]b, c[$ et il est inclus dans le triangle ouvert abc. Soit le point anguleux c' du bord du convexifié le plus proche de b ; il sera tel que $]b, c'[\cup \mathcal{C} = \emptyset$ et $]b, c'[\subset \mathcal{O}_h$. Il suffit de prendre $]b, c'[=]\alpha, \beta[$.

■

Preuve du théorème 3.1:

Construisons par récurrence une suite d'ouverts $\mathcal{O}^i, i = 0, ..., k$, avec $\mathcal{O}^0 \stackrel{def}{=} \mathcal{O}$.

Retirons à l'ouvert $\mathcal{O}^i$ un segment $]a_i, b_i[$ joignant deux sommets a_i, b_i et tel que $]a_i, b_i[\subset \mathcal{O}^i$, tant qu'il existe un tel segment.

$$\mathcal{O}^{i+1} \stackrel{def}{=} \mathcal{O}^i \setminus]a_i, b_i[\tag{2.3}$$

Soit N_c le nombre de sommets ; le nombre total de segments joignant ces sommets étant majoré par $N_c \times (N_c - 1)/2$, la suite est donc finie en $k < N_c \times (N_c - 1)/2$.

Pour finir, chaque composante connexe de l'ouvert $\mathcal{O}^k$ est un triangle (sinon le lemme nous permettrait de continuer) et le domaine est découpé en triangles. ■

Remarque 2.2 *Malheureusement ce théorème n'est plus vrai en dimension plus grande que 2, car il existe des configurations d'ouvert polyédrique non-convexe qu'il est impossible de mailler sans point interne.*

2.1.3 Maillage de Delaunay-Voronoï

Pour construire un maillage, nous avons besoin de connaître :
- un ensemble de points

$$\mathcal{S} \stackrel{def}{=} \{x^i \in \mathbb{R}^2 / i \in \{1, ..., N_p\}\} \tag{2.4}$$

- un ensemble d'arêtes (couples de numéros de points) définissant le maillage de la frontière Γ_h des sous-domaines.

$$\mathcal{A} \stackrel{def}{=} \{(sa_1^j, sa_2^j) \in 1, ..., {N_p}^2 / j \in \{1, ..., N_a\}\} \tag{2.5}$$

- un ensemble de sous-domaines (composantes connexes de $\mathbb{R}^2 \setminus \Gamma_h$) à mailler, avec l'option par défaut suivante : mailler tous les sous-domaines bornés. Les sous-domaines peuvent être définis par une arête frontière et un sens (le sous domaine est à droite (-1) ou à gauche (+1) de l'arête orientée). Formellement, nous disposons donc de l'ensemble

$$\mathcal{SD} \stackrel{def}{=} \{(a^i, sens^i) \in \{1, ..., N_a\} \times \{-1, 1\} / i = 1, N_{sd}\} \tag{2.6}$$

qui peut être vide (cas par défaut).

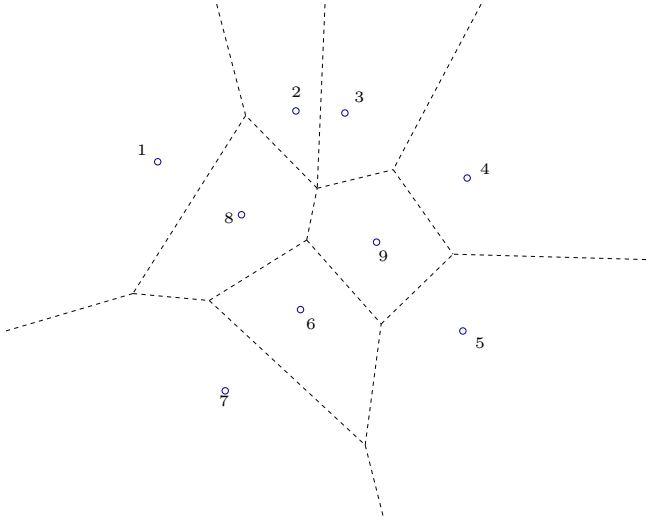


Figure 2.1 – Diagramme de Voronoï : les ensembles des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j.

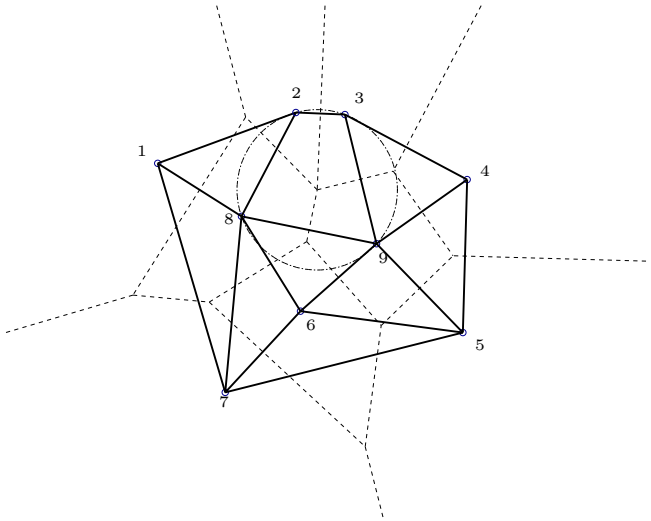


Figure 2.2 – Maillage de Delaunay : nous relions deux points x^i et x^j si les diagrammes V^i et V^j ont un segment en commun.

La méthode est basée sur les diagrammes de Voronoï :

Définition 2.2 *Les diagrammes de Voronoï sont les polygones convexes $V^i, i = 1, N_p$ formés par l'ensemble des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j (voir figure 3.1).*

On peut donc écrire formellement :

$$V^i \stackrel{def}{=} \{x \in \mathbb{R}^2 / ||x - x^i|| \leq ||x - x^j||, \forall j \in \{1, ..., N_p\}\}. \tag{2.7}$$

Ces polygones, qui sont des intersections finies de demi-espaces, sont convexes. De plus, les sommets v^k de ces polygones sont à égale distance des points $\{x^{i_j^k} / j = 1, ..., n_k\}$ de $\mathcal{S}$, où le nombre n_k est généralement égal (le cas standard) ou supérieur à 3. À chacun de ces sommets v^k, nous pouvons associer le polygone convexe construit avec les points $\{x^{i_j^k}, j = 1, ..., n_k\}$ en tournant dans le sens trigonométrique. Ce maillage est généralement formé de triangles, sauf si il y a des points cocycliques (voir figure 3.2 où $n_k > 3$).

Définition 2.3 *Nous appelons maillage de Delaunay strict, le maillage dual des diagrammes de Voronoï, construit en reliant deux points x^i et x^j, si les diagrammes V^i et V^j ont un segment en commun.*
Pour rendre le maillage triangulaire, il suffit de découper les polygones qui ne sont pas des triangles en triangles. Nous appelons ces maillages des maillages de Delaunay de l'ensemble $\mathcal{S}$.

Remarque 2.3 *Le domaine d'un maillage de Delaunay d'un ensemble de points $\mathcal{S}$ est l'intérieur du convexifié $\mathcal{C}(\mathcal{S})$ de l'ensemble de points $\mathcal{S}$.*

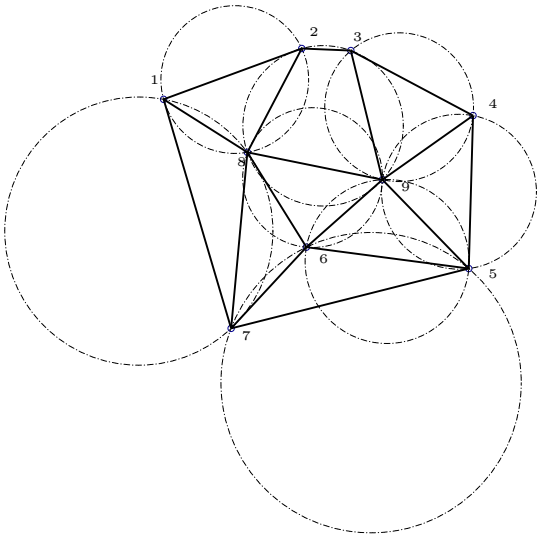


Figure 2.3 – Propriété de la boule vide : le maillage de Delaunay et les cercles circonscrits aux triangles.

Nous avons le théorème suivant qui caractérise les maillages de Delaunay :

Théorème 2.2

Un maillage $\mathcal{T}_{d,h}$ de Delaunay est tel que : pour tout triangle T du maillage, le disque ouvert $D(T)$ correspondant au cercle circonscrit à T ne contient aucun sommet (propriété de la boule vide). Soit, formellement :

$$D(T) \cap \mathcal{T}_{0,h} = \emptyset. \tag{2.8}$$

Réciproquement, si le maillage $\mathcal{T}_{d,h}$ d'un domaine convexe vérifie la propriété de la boule vide, alors il est de Delaunay.

Preuve : il est clair que si le maillage est de Delaunay, il vérifie la propriété de la boule vide.

Réciproquement : soit un maillage vérifiant (3.8).

Commençons par montrer la propriété (a) suivante : toutes les arêtes (x_i, x_j) du maillage sont telles que l'intersection $V^i \cap V^j$ contient les centres des cercles circonscrits aux triangles contenant $]x_i, x_j[$.

Soit une arête (x_i, x_j) ; cette arête appartient au moins à un triangle T. Notons c le centre du cercle circonscrit à T et montrons par l'absurde que c appartient à V^i et à V^j.

Si c n'est pas dans V^i, il existe un x^k tel que $||c - x^k|| < ||c - x^i||$ d'après (3.7), ce qui implique que x^k est dans le cercle circonscrit à T, d'où la contradiction avec l'hypothèse. Donc, c est dans V^i et il y en va de même pour V^j, ce qui démontre la propriété (a).

Il reste deux cas à étudier : l'arête est frontière ou est interne.
- si l'arête (x_i, x_j) est frontière, comme le domaine est convexe, il existe un point c' sur la médiatrice de x^i et x^j suffisamment loin du domaine dans l'intersection de V^i et V^j et tel que c' ne soit pas un centre de cercle circonscrit de triangle,
- si l'arête (x_i, x_j) est interne, elle est contenue dans un autre triangle T' et $V^i \cap V^j$ contient aussi c', le centre du cercle circonscrit à T'.

Dans tous les cas, c et c' sont dans $V^i \cap V^j$ et comme V^i et V^j sont convexes, l'intersection $V^i \cap V^j$ est aussi convexe. Donc le segment $[c, c']$ est inclus dans $V^i \cap V^j$.

Maintenant, il faut étudier les deux cas : $c = c'$ ou $c \neq c'$.
- si le segment $[c, c']$ n'est pas réduit à un point, alors l'arête (x_i, x_j) est dans le maillage Delaunay ;
- si le segment $[c, c']$ est réduit à un point, alors nous sommes dans cas où l'arête (x_i, x_j) est interne et c' est le centre de $D(T')$. Les deux triangles T, T' contenant l'arête (x_i, x_j) sont cocycliques et l'arête n'existe pas dans le maillage de Delaunay strict.

Pour finir, les arêtes qui ne sont pas dans le maillage de Delaunay sont entre des triangles cocycliques. Il suffit de remarquer que les classes d'équivalence des triangles cocycliques d'un même cercle forment un maillage triangulaire de polygones du maillage de Delaunay strict.

■

Cette démonstration est encore valide en dimension d quelconque, en remplaçant les arêtes par des hyperfaces qui sont de codimension 1.

Remarque 2.4

Il est possible d'obtenir un maillage de Delaunay strict en changeant la propriété de la boule vide définie en (3.8) par la propriété stricte de la boule vide, définie comme suit :

$$\overline{D(T)} \cap \mathcal{T}_{0,h} = \overline{T} \cap \mathcal{T}_{0,h}, \tag{2.9}$$

où $\overline{D(T)}$ est le disque fermé correspondant au cercle circonscrit à un triangle T et $\mathcal{T}_{0,h}$ est l'ensemble de sommets du maillage. La différence entre les deux propriétés (3.9) et (3.8) est qu'il peut exister dans (3.8) d'autres points de $\mathcal{T}_{0,h}$ sur le cercle circonscrit $C(T) = \overline{D(T)} \setminus D(T)$.

B. Delaunay a montré que l'on pouvait réduire cette propriété au seul motif formé par deux triangles adjacents.

Lemme 2.2

[Delaunay] Si le maillage $\mathcal{T}_{d,h}$ d'un domaine convexe est tel que tout sous-maillage formé de deux triangles adjacents par une arête vérifie la propriété de la boule, alors le maillage $\mathcal{T}_{d,h}$ vérifie la propriété globale de la boule vide et il est de Delaunay.

La démonstration de ce lemme est basée sur

Alternative 1 *Si deux cercles C_1 et C_2 s'intersectent sur la droite D séparant le plan des deux demi-plans P^+ et P^-, alors on a l'alternative suivante :*

$$D_1 \cap P^+ \subset D_2 \text{ et } D_2 \cap P^- \subset D_1,$$

ou

$$D_2 \cap P^+ \subset D_1 \text{ et } D_1 \cap P^- \subset D_2,$$

où D_i est le disque associé au cercle C_i, pour $i = 1, 2$.

Exercice 2.1

La démonstration de l'alternative est laissée en exercice au lecteur.

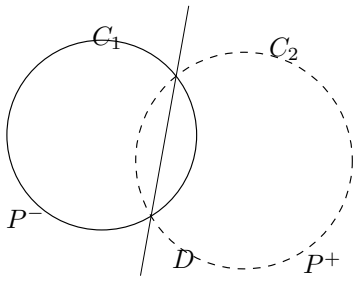


Figure 2.4 – Représentation graphique de l'alternative 1.

Preuve du lemme de Delaunay: nous faisons une démonstration par l'absurde qui est quelque peu technique.

Supposons que le maillage ne vérifie pas la propriété de la boule vide. Alors il existe un triangle $T \in \mathcal{T}_{d,h}$ et un point $x^i \in \mathcal{T}_{0,h}$, tel que $x^i \in D(T)$.

Soit a un point interne au triangle T tel que $\overline{T_a^j} \cap]a, x^i[= \emptyset$ (il suffit de déplacer un petit peu le point a si ce n'est pas le cas). Le segment $]a, x^i[$ est inclus dans le domaine car il est convexe. Nous allons lui associer l'ensemble des triangles $T_a^j, j = 0, .., k_a$ qui intersectent le segment$]a, x^i[$. Cette ensemble est une une chaîne de triangles T_a^j pour $j = 0, .., k_a$ c'est à dire que les triangles T_a^j et T_a^{j+1} sont adjacents par une arête à cause de l'hypothèse $\overline{T_a^j} \cap]a, x^i[= \emptyset$.

Nous pouvons toujours choisir le couple (T, x^i) tel que $x^i \in D(T)$ et $x^i \notin T$ tel que le cardinal k de la chaîne soit minimal. Le lemme se résume à montrer que $k = 1$.

Soit x^{i_1} (resp. x^{i_0}) le sommet de T^1 (resp. T^0) opposé à l'arête $T^0 \cup T^1$.

- Si x^{i_0} est dans $D(T^1)$ alors $k = 1$ (la chaîne est T^1, T^0).
- Si x^{i_1} est dans $D(T^0)$ alors $k = 1$ (la chaîne est T^0, T^1)
- Sinon, les deux points x^{i_0} et x^{i_1} sont de part et d'autre de la droite D définie par l'intersection des deux triangles T^0 et T^1, qui est aussi la droite d'intersection des deux cercles $C(T^0)$ et $C(T^1)$. Cette droite D définit deux demi-plans $\mathcal{P}^0$ et $\mathcal{P}^1$ qui contiennent, respectivement, les points x^{i_0} et x^{i_1}. Pour finir, il suffit d'utiliser l'alternative 1 avec les cercles $C(T^0)$ et $C(T^1)$. Comme x^{i_0} n'est dans $D(T^1)$, alors $D(T^0)$ est inclus dans $D(T^1) \cap \mathcal{P}^0$. Mais, $x^i \in C(T^0)$ par hypothèse, et comme x^i n'est pas dans le demi-plan $\mathcal{P}^1$ car le segment $]a, x^i[$ coupe la droite D, on a $x^i \in C(T^0) \subset C(T^1)$, ce qui impliquerait que le cardinal de la nouvelle chaîne est $k - 1$ et non k d'où la contradiction avec l'hypothèse de k minimal.

■

En fait, nous avons montré que nous pouvons réduire cette propriété au seul motif formé par deux triangles adjacents. De plus, comme un quadrilatère non-convexe maillé en deux triangles vérifie la propriété de la boule vide, il suffit que cette propriété soit vérifiée pour toutes les paires de triangles adjacents formant un quadrilatère convexe.

Remarque 2.5

La démonstration du lemme de Delaunay est encore valide en dimension n ; il suffit de remplacer cercle par sphère, droite par hyperplan et arête par hyperface.
Mais, attention, en dimension 3, il existe des configurations formées de deux tétraèdres adjacentes par une face non-convexe ne vérifiant pas la propriété de la boule vide .

Nous ferons un échange de diagonale $[s^a, s^b]$ dans un quadrilatère convexe de coordonnées s^1, s^a, s^2, s^b (tournant dans le sens trigonométrique) si le critère de la boule vide n'est pas vérifié comme dans la figure 3.5.

Lemme 2.3

Le critère de la boule vide dans un quadrilatère convexe s^1, s^a, s^2, s^b en $[s^1, s^2]$ est équivalent à l'inégalité angulaire (propriété des angles inscrits dans un cercle) :

$$\widehat{s^1 s^a s^b} < \widehat{s^1 s^2 s^b}.$$

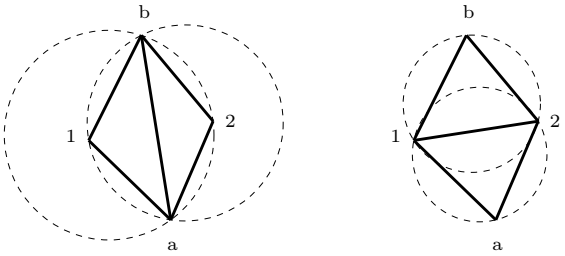


Figure 2.5 – ,Échange de diagonale d'un quadrilatère convexe selon le critère de la boule vide.

Preuve : comme la cotangente est une fonction strictement décroissante entre $]0, \pi[$, il suffit de vérifier :

$$cotg(\widehat{s^1 s^a s^b}) = \frac{(s^1 - s^a, s^b - s^a)}{det(s^1 - s^a, s^b - s^a)} \quad > \quad \frac{(s^1 - s^2, s^b - s^2)}{det(s^1 - s^2, s^b - s^2)} = cotg(\widehat{s^1 s^2 s^b}),$$

où $(.,.)$ est le produit scalaire de $\mathbb{R}^2$ et $det(.,.)$ est le déterminant de la matrice formée avec les deux vecteurs de $\mathbb{R}^2$.

Ou encore, si l'on veut supprimer les divisions, on peut utiliser les aires des triangles $aire^{1ab}$ et $aire^{12b}$. Comme

$$det(s^1 - s^a, s^b - s^a) = 2 \times aire^{1ab} \quad \text{et} \quad det(s^1 - s^2, s^b - s^2) = 2 \times aire^{12b},$$

le critère d'échange de diagonale optimisé est

$$aire^{12b} \quad (s^1 - s^a, s^b - s^a) \quad > \quad aire^{1ab} \quad (s^1 - s^2, s^b - s^2). \tag{2.10}$$

■

Maintenant, nous avons théoriquement les moyens de construire un maillage passant par les points donnés, mais généralement nous ne disposons que des points de la frontière ; il va falloir donc générer ultérieurement les points internes du maillage.

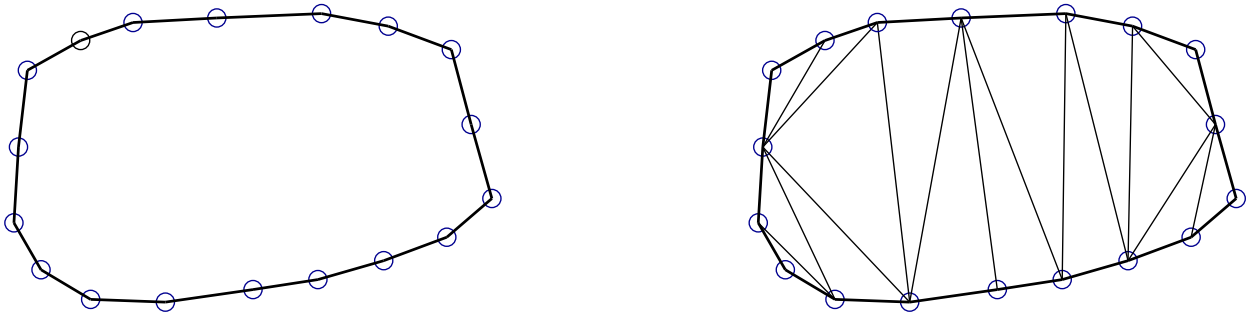


Figure 2.6 – Exemple de maillage d'un polygone sans point interne.

Par construction, le maillage de Delaunay n'impose rien sur les arêtes. Il peut donc arriver que ce maillage ne respecte pas la discrétisation de la frontière, comme nous pouvons le remarquer sur la figure 3.7. Pour éviter ce type de maillage, nous pouvons suivre deux pistes :

- modifier le maillage afin qu'il respecte la frontière ;
- ou bien, modifier la discrétisation de la frontière afin qu'elle soit contenue dans le maillage de Delaunay.

Pour des raisons de compatibilité avec d'autres méthodes nous ne modifierons pas le maillage de la frontière.

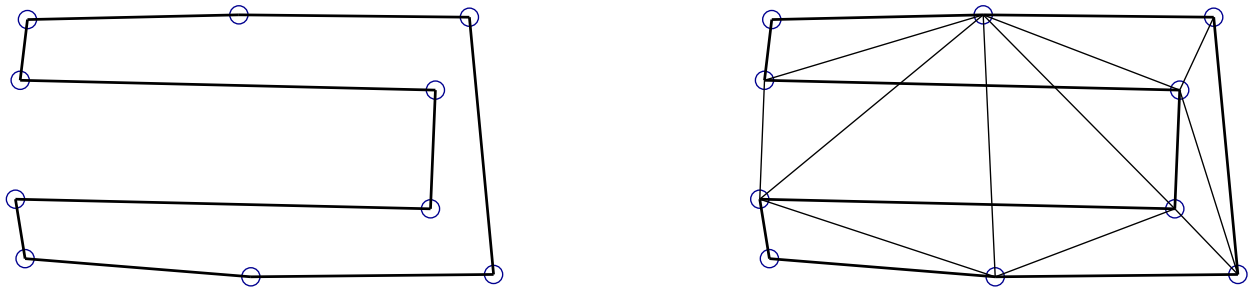


Figure 2.7 – Exemple de maillage de Delaunay ne respectant pas la frontière.

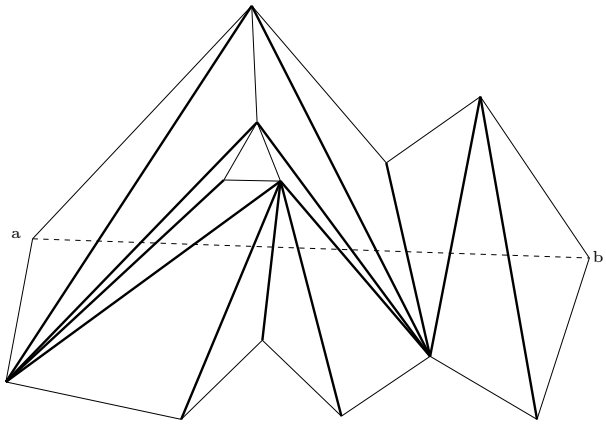


Figure 2.8 – Exemple d'arête $]a, b[$ manquante dans un maillage compliqué.

2.1.4 Forçage de la frontière

Comme nous l'avons déjà vu, les arêtes de la frontière ne sont pas toujours dans le maillage de Delaunay construit à partir des points frontières (voir figures 3.7 et 3.8).

Nous dirons qu'une arête (a, b) coupe une autre arête (a', b') si $]a, b[\cap]a', b'[= p \neq \emptyset$.

Théorème 2.3 *Soient $\mathcal{T}_{d,h}$ une triangulation et a, b deux sommets différents de $\mathcal{T}_{d,h}$ (donc dans $\mathcal{T}_{0,h}$) tels que*

$$]a,b[\cap\mathcal{T}_{0,h}=\emptyset \quad \text{et} \quad]a,b[\subset \mathcal{O}_h. \tag{2.11}$$

Alors, il existe une suite finie d'échanges de diagonale de quadrilatère convexe, qui permet d'obtenir un nouveau maillage $\mathcal{T}_{d,h}^{ab}$ contenant l'arête (a,b).
Nous avons de plus la propriété de localité optimale suivante : toute arête du maillage $\mathcal{T}_{d,h}$ ne coupant pas $]a,b[$ est encore une arête du nouveau maillage $\mathcal{T}_{d,h}^{ab}$.

Preuve : nous allons faire un démonstration par récurrence sur le nombre $m_{ab}(\mathcal{T}_{d,h})$ d'arêtes du maillage $\mathcal{T}_{d,h}$ coupant l'arête (a,b).

Soit T^i, pour $i=0,...,m_{ab}(\mathcal{T}_{d,h})$, la liste des triangles coupant $]a,b[$ tel que les traces des T^i sur $]a,b[$ aillent de a à b pour $i=0,...,n$.

Comme $]a,b[\cap\mathcal{T}_{0,h}=\emptyset$, l'intersection de $\overline{T}^{i-1}$ et $\overline{T}^{i}$ est une arête notée $[\alpha_i,\beta_j]$ qui vérifie

$$[\alpha_i,\beta_j] \quad \stackrel{def}{=} \quad \overline{T}^{i-1}\cap\overline{T}^{i}, \qquad \text{avec} \qquad \alpha_i\in P_{ab}^{+} \quad \text{et} \quad \beta_i\in P_{ab}^{-}, \tag{2.12}$$

où P_{ab}^{+} et P_{ab}^{-} sont les deux demi-plans ouverts, définis par la droite passant par a et b.

Nous nous placerons dans le maillage restreint $\mathcal{T}_{d,h}^{r_{a,b}}$ formé seulement de triangles T^i pour $i=0,...,m_{ab}(\mathcal{T}_{d,h})=m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})$ pour assurer le propriété de localité. De plus, le nombre de triangles N_t^{ab} de $\mathcal{T}_{d,h}^{a,b}$ est égal à $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})+1$.

- Si $m_{ab}(\mathcal{T}_{d,h}^{r_{ab}})=1$, on fait l'observation que le quadrilatère formé par les deux triangles contenant l'unique arête coupant (a,b) est convexe, et donc il suffit d'échanger les arêtes du quadrilatère.
- Sinon, supposons vraie la propriété pour toutes les arêtes (a,b) vérifiant (3.11) et telles que $m_{ab}(\mathcal{T}_{d,h}^{r_{ab}})<n$ et ceci pour tous les maillages possibles.

Soit une arête (a,b) vérifiant (3.11) et telle que $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})=n$. Soit α_{i^+} le sommet α_i pour $i=1,...,n$ le plus proche du segment $[a,b]$. Nous remarquerons que les deux inclusions suivantes sont satisfaites :

$$]a,\alpha_{i^+}[\subset \overbrace{\bigcup_{i=0}^{i^+-1}\overline{T}^{i}}^{\circ} \quad \text{et} \quad]\alpha_{i^+},b[\subset \overbrace{\bigcup_{i=i^+}^{n}\overline{T}^{i}}^{\circ}. \tag{2.13}$$

Les deux arêtes $]a,\alpha_{i^+}[$ et $]\alpha_{i^+},b[$ vérifient les hypothèses de récurrence, donc nous pouvons les forcer par échange de diagonales, car elles ont des supports disjoints. Nommons $\mathcal{T}_{d,h}^{r_{a,b+}}$ le maillage obtenu après forçage de ces deux arêtes. Le nombre de triangles de $\mathcal{T}_{d,h}^{r_{a,b+}}$ est égal à $n+1$ et, par conséquent, $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b+}})\leq n$.

Il nous reste à analyser les cas suivants :
- si $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b+}})<n$, nous appliquons l'hypothèse de récurrence et la démonstration est finie ;

- si $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b+}}) = n$, nous allons forcer (a, b) dans le maillage $\mathcal{T}_{d,h}^{r_{a,b+}}$ et utiliser la même méthode. Les T^i seront maintenant les triangles de $\mathcal{T}_{d,h}^{r_{a,b+}}$ et les α^+ en β^+ seront définis par (3.12). Nous avons traité le demi-plan supérieur, traitons maintenant la partie inférieure. Soit i^- l'indice tel que le sommet $\beta_{i^+}^+$ soit le plus proche du segment $]a, b[$ des sommets β_i^+. Nous forçons les deux arêtes $]a, \beta_{i^-}[$ et $]\beta_{i^-}, b[$ en utilisant les mêmes arguments que précédemment.

Nous avons donc un maillage local du quadrilatère $a, \alpha_{i^+}, b, \beta_{i^-}^+$ qui contient l'arête $]a, b[$ et qui ne contient aucun autre point du maillage. Il est donc formé de deux triangles T, T' tels que $]a, b[\subset \overline{T} \cup \overline{T}'$, ce qui nous permet d'utiliser une dernière fois l'hypothèse de récurrence $(n = 1)$ pour finir la démonstration. ∎

Remarque 2.6 *On en déduit facilement une autre démonstration du théorème 3.1 : il suffit de prendre un maillage de Delaunay de l'ensemble des sommets de l'ouvert, de forcer tous les segments frontières de l'ouvert et de retirer les triangles qui ne sont pas dans l'ouvert.*

Du théorème 3.3, il découle :

Théorème 2.4 *Soit deux maillages $\mathcal{T}_{d,h}$ et $\mathcal{T}'_{d,h}$ ayant les mêmes sommets ($\mathcal{T}_{0,h} = \mathcal{T}'_{0,h}$) et le même maillage du bord $\partial\mathcal{T}_{d,h} = \partial\mathcal{T}'_{d,h}$. Alors, il existe une suite d'échanges de diagonales de quadrilatères convexes qui permet de passer du maillage $\mathcal{T}_{d,h}$ au maillage $\mathcal{T}'_{d,h}$.*

Preuve : il suffit de forcer toutes les arêtes du maillage $\mathcal{T}_{d,h}$ dans $\mathcal{T}'_{d,h}$. ∎

Pour finir cette section, donnons un algorithme de forçage d'arête très simple (il est dû à Borouchaki [George,Borouchaki-1997, page 99]).

Algorithme 2.1 *Si l'arête (s^a, s^b) n'est pas une arête du maillage de Delaunay, nous retournons les diagonales (s^α, s^β) des quadrangles convexes $s^\alpha, s^1, s^\beta, s^2$ formés de deux triangles dont la diagonale $]s^\alpha, s^\beta[$ coupe $]s^a, s^b[$ en utilisant les critères suivants :*
- *si l'arête $]s^1, s^2[$ ne coupe pas $]s^a, s^b[$, alors on fait l'échange de diagonale ;*
- *si l'arête $]s^1, s^2[$ coupe $]s^a, s^b[$, on fait l'échange de diagonale de manière aléatoire.*

Comme il existe une solution au problème, le fait de faire des échanges de diagonales de manière aléatoire va permettre de converger, car, statistiquement, tous les maillages possibles sont parcourus et ils sont en nombre fini.

2.1.5 Recherche de sous-domaines

L'idée est de repérer les parties qui sont les composantes connexes de $\mathcal{O}_h \setminus \cup_{j=1}^{N_a} [x^{sa_1^j}, x^{sa_2^j}]$, ce qui revient à définir les composantes connexes du graphe des triangles adjacents où l'on a supprimé les connexions avec les arêtes de $\mathcal{A}$. Pour cela, nous utilisons l'algorithme de coloriage qui recherche la fermeture transitive d'un graphe.

Algorithme 2.2

Coloriage de sous-domaines
`Coloriage(T)`
Si `T` *n'est pas colorié*
Pour tous les triangles `T'` *adjacents à* `T` *par une arête non-marquée*
`Coloriage(T')`
marquer toutes les arêtes $\mathcal{T}_{d,h}$ *qui sont dans* $\mathcal{A}$
Pour tous les Triangles `T` *non-coloriés*
Changer de couleur
`Coloriage(T)`

Observons qu'à chaque couleur correspond une composante connexe de $\mathcal{O}_h \setminus \cup_{j=1}^{N_a}[x^{sa_1^j}, x^{sa_2^j}]$. La complexité de l'algorithme est en $3 \times N_t$, où N_t est le nombre de triangles.

Attention, si l'on utilise simplement la récursivité du langage, nous risquons de gros problèmes car la profondeur de la pile de stockage mémoire est généralement de l'ordre du nombre d'éléments du maillage.

Exercice 2.2 Récrire l'algorithme 3.2 sans utiliser la récursivité.

2.1.6 Génération de points internes

Pour compléter l'algorithme de construction du maillage, il faut savoir générer les points internes (voir figure 3.3). Le critère le plus naturel pour distribuer les points internes est d'imposer en tout point $\mathbf{x}$ de $\mathbb{R}^2$, le pas de maillage $h(\mathbf{x})$.

La difficulté est que, généralement, cette information manque et il faut construire la fonction $h(\mathbf{x})$ à partir des données du maillage de la frontière. Pour ce faire, nous pouvons utiliser, par exemple, la méthode suivante :

1. À chaque sommet s de $\mathcal{T}_{0,h}$ on associe une taille de maillage qui est la moyenne des longueurs des arêtes ayant comme sommet s. Si un sommet de $\mathcal{T}_{0,h}$ n'est contenu dans aucune arête, alors on lui affecte une valeur par défaut, par exemple le pas de maillage moyen.
2. On construit le maillage de Delaunay de l'ensemble des points.
3. Pour finir, on calcule l'interpolé P^1 (voir §**??**, définition **??**) de la fonction $h(\mathbf{x})$ dans tous les triangles de Delaunay.

Dans la pratique, nous préférons disposer d'une fonction $N(a,b)$ de $\mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ qui donne le nombre de mailles entre les points a et b. Nous pouvons construire différents types de fonctions N à partir de $h(\mathbf{x})$. Par exemple, une expression utilisant la moyenne simple $|ab|^{-1}\int_a^b h(t)dt$ serait :

$$N_1(a,b) \quad \stackrel{def}{=} \quad |ab|(\frac{\int_a^b h(t)dt}{|ab|})^{-1} = (\int_a^b h(t)dt)^{-1}. \tag{2.14}$$

Nous pouvons aussi utiliser quelques résultats de la géométrie différentielle. La longueur l d'une courbe γ paramétrée par $t \in [0,1]$ étant définie par

$$l \quad \stackrel{def}{=} \quad \int_{t=0}^{1} \sqrt{(\gamma'(t), \gamma'(t))}dt, \tag{2.15}$$

si nous voulons que la longueur dans le plan tangent en $\mathbf{x}$ d'un segment donne le nombre de pas, il suffit d'introduire la longueur *remmanienne* suivante (on divise par $h(\mathbf{x})$) :

$$l^h \stackrel{def}{=} \int_{t=0}^{1} \frac{\sqrt{(\gamma'(t), \gamma'(t))}}{h(\gamma(t))} dt \tag{2.16}$$

Nous obtenons une autre définition de la fonction $N(a, b)$:

$$N_2(a, b) \stackrel{def}{=} \int_a^b h(t)^{-1} dt. \tag{2.17}$$

Les deux méthodes de construction de $N(a, b)$ sont équivalentes dans le cas où $h(\mathbf{x})$ est indépendant de $\mathbf{x}$. Regardons ce qui change dans le cas d'une fonction h affine sur le segment $[0, l]$. Si

$$h(t) = h_0 + (h_l - h_0)t, \quad t \in [0, 1],$$

nous obtenons

$$N_1(0, t) = \left(\int_0^t [h_0 + (h_l - h_0)x] dx \right)^{-1} = (h_0 t + \frac{(h_l - h_0)}{2} t^2)^{-1}, \tag{2.18}$$

$$N_2(0, t) = \frac{1}{h_l - h_0} \ln \left(1 + \frac{h_l - h_0}{h_0} t \right). \tag{2.19}$$

Par construction, N_2 vérifie la relation de Chasle :

$$N_2(a, c) = N_2(a, b) + N_2(b, c),$$

où b est un point entre a et b, alors que N_1 ne vérifie clairement pas cette relation. C'est la raison pour laquelle nous préférons l'expression (3.19) pour la fonction N. De plus, si nous voulons avoir un nombre optimal de points tels que $N_2(0, t) = i, \forall i \in 1, 2, ..$ dans (3.19), ces points seront alors distribués suivant une suite géométrique de raison $\frac{\ln(h_l - h_0)}{h_l - h_0}$.

2.2 Algorithme de construction du maillage

Pour construire un maillage de Delaunay à partir d'un ensemble donné de points, nous allons suivre les étapes suivantes :

1. Prendre pour origine le point de coordonnée minimale $O = (\min_{i=1,N_p} x^i, \min_{i=1,N_p} y^i)$, où les (x^i, y^i) sont les coordonnées des points $\mathbf{x}^i$.
2. Trier les $\mathbf{x}^i$ en ordre lexicographique par rapport à la norme $||\mathbf{x}^i - O||$ puis par rapport à y^i. Cet ordre est tel que le point courant ne soit jamais dans le convexifié des points précédents.
3. Ajouter les points un à un suivant l'ordre prédéfini à l'étape 2. Les points ajoutés sont toujours à l'extérieur du maillage courant. Donc, on peut créer un nouveau maillage en reliant toutes les arêtes frontières du maillage précédent qui voit les points du bon côté.

4. Pour que le maillage soit de Delaunay, il suffit d'appliquer la méthode **d'optimisation locale** du maillage suivant : autour d'un sommet rendre de Delaunay tous les motifs formés de deux triangles convexes contenant ce sommet, en utilisant la formule (3.10). Puis, appliquer cette optimisation à chaque nouveau point ajouté pour que le maillage soit toujours de Delaunay.
5. Forcer la frontière dans le maillage de Delaunay en utilisant l'algorithme 3.1.
6. Retirer les parties externes du domaine en utilisant l'algorithme 3.2.
7. Générer des points internes. S'il y a des points internes nous recommençons avec l'étape 1, avec l'ensemble de points auquel nous avons ajouté les points internes.
8. Régularisation, optimisation du maillage.

2.3 Programme C++

La programmation d'un générateur de maillage n'est pas simple, les problèmes informatiques qu'il faut résoudre sont : choix des structures de données, problème d'erreurs d'arrondi, problèmes de localisation, etc. Les choix fait ici correspondent à des développements fait dans les logiciels de l'INRIA : `emc2` [1] qui est inclus dans la bibliothèque éléments finis `MODULEF` [2], `ghs3d`[3], Cet exemple est directement extrait du logiciel `Bamg`. [4]

Les idées utilisées pour ce développement sont décrites dans [Frey, George-1999]. Pour les curieux, une version en Fortran du générateur de maillage du logiciel `emc2` est disponible dans le fichier *Mesh/mshptg.f*.

Dans un premier temps, nous commençons par modéliser le plan $\mathbb{R}^2$ avec le même type de classes que dans la section §**??**. Mais, ici nous avons à faire des calculs d'aire de triangles sans erreurs d'arrondi. Pour faire ceci, nous allons construire différents types de coordonnées de $\mathbb{R}^2$: des coordonnées entières pour un calcul exact et des coordonnée flottantes. En conséquence, la classe `P2` qui modélise $\mathbb{R}^2$ dépend de deux types : le type (`R`) des coordonnées et le type (`RR`) du résultat d'un produit (produit scalaire, déterminant, etc.).

Listing 2.1 *(R2.hpp)*

```
#include <cstdio>
#include <cstring>
#include <cmath>

#include <iostream>

namespace bamg {
using namespace std;
```

[1] `http://www-rocq.inria.fr/gamma/cdrom/www/emc2/fra.htm`
[2] `http://www-rocq.inria.fr/modulef/`
[3] `http://www-rocq.inria.fr/gamma/ghs3d/ghs.html`
[4] `http://www-rocq.inria.fr/gamma/cdrom/www/bamg/fra.htm`

```
template <class R,class RR>
class P2 {

public:
  R x,y;
  P2 () :x(0),y(0) {};
  P2 (R a,R b)  :x(a),y(b)  {}
  P2<R,RR>   operator+(const P2<R,RR> & cc) const
                    {return P2<R,RR>(x+cc.x,y+cc.y);}
  P2<R,RR>   operator-(const P2<R,RR> & cc) const
                    {return P2<R,RR>(x-cc.x,y-cc.y);}
  P2<R,RR>   operator-()  const{return P2<R,RR>(-x,-y);}
  RR   operator,(const P2<R,RR> & cc) const            //    produit scalaire
                     {return  (RR) x* (RR) cc.x+(RR) y* (RR) cc.y;}
  P2<R,RR>  operator*(R  cc) const
                     {return P2<R,RR>(x*cc,y*cc);}
  P2<R,RR>  operator/(R  cc) const
                      {return P2<R,RR>(x/cc,y/cc);}
  P2<R,RR>  operator+=(const  P2<R,RR> & cc)
                      {x += cc.x;y += cc.y;return *this;}
  P2<R,RR>  operator/=(const  R r)
                      {x /= r;y /= r;return *this;}
  P2<R,RR>  operator*=(const  R r)
                      {x *= r;y *= r;return *this;}
  P2<R,RR>  operator-=(const  P2<R,RR> & cc)
                      {x -= cc.x;y -= cc.y;return *this;}
 };

template  <class R,class RR>
inline RR Det(const P2<R,RR> x,const P2<R,RR> y) {
  return (RR) x.x * (RR) y.y - (RR) x.y * (RR) y.x;}

template  <class R,class RR>
inline RR Area2 (const P2<R,RR> a,const P2<R,RR> b,const P2<R,RR> c) {
  return Det(b-a,c-a);}

template <class R,class RR>
inline  ostream& operator «(ostream& f, const P2<R,RR> & c)
  { f «  c.x « "   " « c.y « ' '; return f; }

}
```

Il nous faut choisir les classes que nous allons utiliser pour programmer le générateur de maillage. Premièrement, il faut décider comment définir un maillage. Il y a clairement deux possibilités : une liste de triangles ou une liste d'arêtes. Les deux choix sont valables, mais ici nous ne traiterons que le cas d'une liste de triangles.

Ensuite, pour construire le maillage, la partie la plus délicate est de définir les objets informatiques qui nous permettrons d'implémenter les algorithmes théoriques de manière simple et efficace.

Nous choisissons de définir :

- un sommet par :

  - les coordonnées réelles et entières (pour éviter les erreurs d'arrondi),
  - la taille de la maille associée h,
  - le numéro de label associé,
  - plus un pointeur t sur un triangle contenant ce sommet et le numéro du sommet dans ce triangle t, afin de retrouver rapidement des triangles quand nous forcerons les arêtes.
- un triangle par :
  - les trois pointeurs sur les trois sommets, tournant dans le sens trigonométrique,
  - les trois triangles adjacents d'un triangle, (pour faire les échanges de diagonale d'un quadrilatère (voir figure 3.5) et pour nous promener dans le maillage *via* les triangles adjacents).
    Il sera aussi utile de connaître pour chaque arête d'un triangle le numéro de l'arête dans le triangle adjacent correspondant (pour éviter des recherches intempestives et simplifier la programmation).
  - le déterminant entier du triangle (deux fois l'aire du triangle).
- Les arêtes du bord seront considérées comme un triangle dégénéré (avec l'un des pointeur sur les sommets nul). Nous pourrons utiliser les triangles adjacents pour parcourir la frontière. Géométriquement, cela revient à ajouter un point à l'infini et de travailler sur une surface topologiquement équivalente à une sphère. Remarquons que le nombre de triangles sera égal à $2 * (nbv - 1)$, car il y a pas d'arêtes frontières (nbv étant le nombre de sommets du maillage).
- Les arêtes du maillage seront représentées comme le couple (triangle, numéro d'arête) et non pas comme un couple de sommets. Elle seront stockées dans la classe `TriangleAdjacent`. Comme nous allons forcer des arêtes dans le maillage, nous allons introduire un système de marquage des arêtes (voir les fonctions `Locked` et la classe `Triangle` du fichier *Mesh.hpp*).
- Le maillage sera modélisé par la classe `Triangles` qui contiendra un tableau de sommets, un tableau de triangles, le nombre maximal de triangles et de sommets du maillage, afin de ne pas faire d'allocations mémoire en cours de génération du maillage. Nous rajoutons aussi quelques petites fonctions utiles (`SetIntCoor`, etc.).

Listing 2.2 *(Mesh.hpp)*

```
#include <cassert>
#include <cstdlib>
#include <cmath>
#include <climits>
#include <ctime>

#include "R2.hpp"

namespace bamg {
using namespace std;

  template<class T> inline T Min (const T &a,const T &b)
            {return a < b? a : b;}
  template<class T> inline T Max (const T &a,const T & b)
            {return a > b? a : b;}
  template<class T> inline T Abs (const T &a){return a <0? -a : a;}
  template<class T> inline double Norme (const T &a){return sqrt(a*a);}
```

```
  template<class T> inline void Exchange (T& a,T& b) {T c=a;a=b;b=c;}

  //    définition des types pour les coordonnées
  typedef double R;
  typedef int  Icoor1;                    //    type d'une coordonnée entière
  typedef double   Icoor2;      //    type des produits de coordonnées entière
  const Icoor1 MaxICoor = 8388608;   //    max de coordonnées entières 2^23 pour
                    //    ne pas avoir d'erreur d'arrondi
  const  Icoor2 MaxICoor22   //    MaxICoor22 plus gros produit de coordonnées
              = Icoor2(2)*Icoor2(MaxICoor) * Icoor2(MaxICoor);            //
entières
  typedef P2<Icoor1,Icoor2> I2;           //    points à coordonnées entières
  typedef P2<R,R> R2;                      //    points à coordonnées réelles

                  //                    Gestion des erreurs

  inline void MeshError(int Err){
    cerr « " Fatal error in the meshgenerator " « Err « endl;
    exit(1);  }


  inline int BinaryRand(){                                //    OuiNon aléatoire
    const long HalfRandMax = RAND_MAX/2;
    return rand() <HalfRandMax;
  }

  //    déterminant entier du triangle a, b, c
  Icoor2 inline det(const I2 &a,const I2 & b,const I2 &c)
  {
     Icoor2 bax = b.x - a.x ,bay = b.y - a.y;
     Icoor2 cax = c.x - a.x ,cay = c.y - a.y;
     return  bax*cay - bay*cax;}


  //    définition des numérotations dans un triangle
  static const short VerticesOfTriangularEdge[3][2] = {{1,2},{2,0},{0,1}};
  static const short EdgesVertexTriangle[3][2] = {{1,2},{2,0},{0,1}};
  static const short OppositeVertex[3] = {0,1,2};
  static const short OppositeEdge[3] =  {0,1,2};
  static const short NextEdge[3] = {1,2,0};
  static const short PreviousEdge[3] = {2,0,1};
  static const short NextVertex[3] = {1,2,0};
  static const short PreviousVertex[3] = {2,0,1};


  class Triangles;                          //    Le maillages (les triangles)
  class Triangle;                                        //    un triangle

//   ////////////////////////////////////////////////////////////////////
  class Vertex {public:
    I2 i;                          //    allow to use i.x, and i.y in long int
    R2 r;                            //    allow to use r.x, and r.y in double
    R  h;                                                 //    Mesh size
    int Label;

    Triangle * t;                     //    un triangle t contenant le sommet
```

```
    short vint;                    //    numéro du sommet dans le triangle t

    operator        I2   () const {return i;}     //    cast en coor. entières
    operator  const R2 & () const {return r;}      //    cast en coor. réelles
    int  DelaunayOptim(int  = 1);           //    optimisation Delaunay autour
    friend ostream& operator «(ostream& f, const  Vertex & v)
           {f « "(" « v.i  « "," « v.r  « ")";   return f;}
  };



//   ////////////////////////////////////////////////////////////////////////
  class TriangleAdjacent {
  public:
    Triangle * t;                                          //    le triangle
    int  a;                                          //    le numéro de l'arête

    TriangleAdjacent(Triangle  * tt,int  aa): t(tt),a(aa &3) {};
    TriangleAdjacent() {};

    operator Triangle * () const   {return t;}
    operator Triangle & () const   {return *t;}
    operator int() const           {return a;}
    TriangleAdjacent operator-()
         {   a= PreviousEdge[a];
             return *this;}
    inline  TriangleAdjacent  Adj() const;
    inline void SetAdj2(const TriangleAdjacent& , int =0);
    inline Vertex *  EdgeVertex(const int &) const;
    inline Vertex *  OppositeVertex() const;
    inline Icoor2 & det() const;
    inline int Locked() const ;
    inline int GetAllFlag_UnSwap() const;
    inline void SetLock();
    friend ostream& operator «(ostream& f, const  TriangleAdjacent & ta)
          {f « "{" « ta.t « "," « ((int) ta.a) « "}";
           return f;}


  };                                                  //    end of Vertex class


  //   ////////////////////////////////////////////////////////////////////////
  class Triangle {
    friend class TriangleAdjacent;
  private:                          //    les arêtes sont opposées à un sommet
    Vertex * ns [3];                                          //    les 3 sommets
    Triangle * at [3];                                  //    nu triangle adjacent
    short  aa[3];             //    les n° des arêtes dans le triangle (mod 4)
                 //    on utilise aussi aa[i] pour marquer l'arête i (i=0,1,2)
                //    si (aa[i] & 4 ) => arête bloquée (lock) marque de type 0
                    //    si (aa[i] & 2^{n+2} => arête marquée de type n (n=0..7)
                //    la marque de type 1 était pour déja basculé (markunswap)
                              //    (aa[i] & 1020 ) l'ensemble des marques
                                    //    1020 = 1111111100 (en binaire)
                                 //    ---------------------------------
```

```
  public:
    Icoor2 det;      //    det. du triangle (2 fois l'aire des coor. entières)

    bool  NonDegenere() const {return ns[0] && ns[1] && ns[2];}
    Triangle() {}
    inline void Set(const Triangle &,const Triangles &,Triangles &);
    inline int In(Vertex *v) const
           { return ns[0]==v || ns[1]==v || ns[2]==v;}

    const Vertex & operator[](int i) const {return *ns[i];};
    Vertex & operator[](int i)             {return *ns[i];};

    const Vertex  *  operator()(int i) const {return ns[i];};
    Vertex  * & operator()(int i)            {return ns[i];};

    TriangleAdjacent Adj(int  i) const         //    triangle adjacent + arête
              { return TriangleAdjacent(at[i],aa[i]&3);};

    Triangle * TriangleAdj(int  i) const
              {return at[i&3];}              //    triangle adjacent + arête

    short  NuEdgeTriangleAdj(int  i) const
              {return aa[i&3]&3;}       //    º de l'arête adj. dans adj tria

    void SetAdjAdj(short a)
           { a &= 3;
             Triangle *tt=at[a];
             aa [a] &= 1015;         //    supprime les marques sauf « swap »
                                     //    (1015 == 1111110111 en binaire)
             register short aatt = aa[a] & 3;
             if(tt){
                tt->at[aatt]=this;
                tt->aa[aatt]=a + (aa[a] & 1020 );}}    //    copie toutes les
marques


    void SetAdj2(short a,Triangle *t,short aat)
           {  at[a]=t;aa[a]=aat;
              if(t) {t->at[aat]=this;t->aa[aat]=a;}}

    void SetTriangleContainingTheVertex()
           { if (ns[0]) (ns[0]->t=this,ns[0]->vint=0);
             if (ns[1]) (ns[1]->t=this,ns[1]->vint=1);
             if (ns[2]) (ns[2]->t=this,ns[2]->vint=2);}

    int DelaunaySwap(short a);       //    bascule le quadrilataire formé avec
                             //    le triangle adj en a si il est non Delaunay

    int DelaunayOptim(short a);             //    Met tous les quad. contenant
                                    //    le sommet a du triangle Delaunay

    int  Locked(int a)const            //    retourne si l'arête est frontière
             { return aa[a]&4;}                //    (plus d'échange d'arête)


    void SetLocked(int a){           //    mark l'arête comme frontière (lock)
```

```
              Triangle * t = at[a];
              t->aa[aa[a] & 3] |=4;
              aa[a] |= 4;}
  };                              //    ----- Fin de la class Triangle -------

  //   ////////////////////////////////////////////////////////////////////

class Triangles {
  public:
    int nbvx,nbtx;                  //    nombre max de sommets, de triangles

    int nbv,nbt;                                  //    nb sommet, de triangles,
    int nbiv,nbtf;                //   nb de triangle dégénéré (arête du bord)
    int NbOfSwapTriangle,NbUnSwap;
    Vertex * vertices;                               //    tableau des sommets

    R2 pmin,pmax;                                                  //    extrema
    R coefIcoor;                              //    coef pour les coor. entière

    Triangle * triangles;
                                                            //    end of variable

    Triangles(int i);                                         //    constructeur
    ~Triangles();
    void SetIntCoor();                    //    construction des coor. entières
    void  RandomInit();           //    construction des sommets aléatoirement

                                                                //    sauce C++
    const Vertex & operator()  (int i) const
             { return vertices[i];};
    Vertex & operator()(int i)
             { return vertices[i];};
    const Triangle & operator[]  (int i) const
             { return triangles[i];};
    Triangle & operator[](int i)
             { return triangles[i];};
                                     //    transformation des coordonnées ...
    I2 toI2(const R2 & P) const {
            return  I2( (Icoor1) (coefIcoor*(P.x-pmin.x))
                       ,(Icoor1) (coefIcoor*(P.y-pmin.y)) );}

    R2 toR2(const I2 & P) const {
            return  R2( (double) P.x/coefIcoor+pmin.x,
                        (double) P.y/coefIcoor+pmin.y);}

                                                //    ajoute sommet à un triangle
    void Add( Vertex & s,Triangle * t,Icoor2 *  =0);

    void Insert();                                  //    insère tous les sommets


    Triangle * FindTriangleContening(const I2 & B, //    recherche le triangle
                                     Icoor2 dete[3], //    contenant le sommet
B
                                     Triangle *tstart) const;    //    partant
de tstart
```

```
  void ReMakeTriangleContainingTheVertex();


  int Number(const Triangle & t) const  { return &t - triangles;}
  int Number(const Triangle * t) const  { return t - triangles;}
  int Number(const Vertex & t) const  { return &t - vertices;}
  int Number(const Vertex * t) const  { return t - vertices;}
private:
  void PreInit(int);
};                                                    //    End Class Triangles
//   ////////////////////////////////////////////////////////////////////////

inline Triangles::Triangles(int i) {PreInit(i);}

//   ////////////////////////////////////////////////////////////////////////

inline  void  TriangleAdjacent::SetAdj2(const TriangleAdjacent & ta, int l  )
{                                              //    set du triangle adjacent
  if(t) {
    t->at[a]=ta.t;
    t->aa[a]=ta.a|l;}
  if(ta.t) {
    ta.t->at[ta.a] = t;
    ta.t->aa[ta.a] = a| l;
  }
}

                              //    l'arête Locked est telle Lock (frontière)
inline int  TriangleAdjacent::Locked() const
{ return t->aa[a] &4;}

                                  //    récupération des tous les flag (Lock)
inline int  TriangleAdjacent::GetAllFlag_UnSwap() const      //   donne tous
{ return t->aa[a] & 1012;}                  //    les marque sauf MarkUnSwap

                                                 //    Construit l' Adjacent
inline  TriangleAdjacent TriangleAdjacent::Adj() const
{ return  t->Adj(a);}

                                                      //    sommet de l'arête
inline Vertex  * TriangleAdjacent::EdgeVertex(const int & i) const
{return t->ns[VerticesOfTriangularEdge[a][i]]; }

                                                    //    sommet opposé à l'arête
inline Vertex  * TriangleAdjacent::OppositeVertex() const
{return t->ns[bamg::OppositeVertex[a]]; }

                                                        //    det du triangle
inline Icoor2 &  TriangleAdjacent::det() const
{ return t->det;}

                                                     //    Construit l'adjacent
inline  TriangleAdjacent Adj(const TriangleAdjacent & a)
{ return  a.Adj();}
```

```
                                        //    Adjacence suivante dans le triangle
  inline TriangleAdjacent Next(const TriangleAdjacent & ta)
  { return TriangleAdjacent(ta.t,NextEdge[ta.a]);}

                                      //    Adjacence précédente dans le triangle
  inline TriangleAdjacent Previous(const TriangleAdjacent & ta)
  { return TriangleAdjacent(ta.t,PreviousEdge[ta.a]);}

                                                  //    Optimisation de Delaunay
  int inline  Vertex::DelaunayOptim(int i)
  {
    int ret=0;
    if ( t && (vint >= 0 ) && (vint <3) )
      {
        ret = t->DelaunayOptim(vint);
        if(!i)
          {
            t =0;                 //    pour supprimer les optimisation future
            vint= 0; }
      }
    return ret;
  }

                                           //    calcul de det du triangle a,b,c
  Icoor2 inline det(const Vertex & a,const Vertex & b,const Vertex & c)
  {
    register  Icoor2 bax = b.i.x - a.i.x ,bay = b.i.y - a.i.y;
    register  Icoor2 cax = c.i.x - a.i.x ,cay = c.i.y - a.i.y;
    return  bax*cay - bay*cax;}

                            //    la fonction qui fait l'échange sans aucun test
  void  swap(Triangle *t1,short a1,
             Triangle *t2,short a2,
             Vertex *s1,Vertex *s2,Icoor2 det1,Icoor2 det2);

        //    la fonction qui fait une échange pour le forçage de la frontière
  int SwapForForcingEdge(Vertex   *  & pva ,Vertex  * &   pvb ,
                         TriangleAdjacent & tt1,Icoor2 & dets1,
                         Icoor2 & detsa,Icoor2 & detsb, int & nbswap);
                      //    la fonction qui force l'arête a,b dans le maillage
  int ForceEdge(Vertex &a, Vertex & b,TriangleAdjacent & taret);

                    //    la fonction qui marque l'arête comme lock (frontière)
  inline void  TriangleAdjacent::SetLock(){ t->SetLocked(a);}
}
```

Les algorithmes théoriques présentés dans ce chapitre sont implémentés dans le fichier *Mesh.cpp* sous la forme des fonctions suivantes :

`swap` la fonction qui fait un échange de diagonale sans aucun test ;

`ForceEdge` la fonction qui force l'arête $[a, b]$ dans le maillage ;

`SwapForForcingEdge` la fonction qui fait un échange diagonale pour forcer une arête ;

`Triangle::DelaunaySwap` la fonction qui fait l'échange de diagonale pour rendre le quadrilatère de Delaunay ;

`Triangles::Add` la fonction qui ajoute un sommet s à un triangle (qui peut être dégénéré) ;

`Triangles::Insert` la fonction qui fait l'insertion de tous les points ;

`Triangles::RandomInit` la fonction qui initialise les coordonnées de manière aléatoire ;

`Triangles::SetIntCoor` la fonction qui construit les sommets entiers à partir des coordonnées réelles ;

`Triangle::DelaunayOptim` la fonction qui rend le maillage de Delaunay autour du sommet s par échange de diagonale.

`Triangles::FindTriangleContening` fonction qui recherche un triangle K de $\mathcal{T}_{d,h}$ contenant un point $B = (x, y)$ à partir d'un triangle de départ T défini avec `tstart`. L'algorithme utilisé est le suivant :

Algorithme 2.3

Partant de du triangle K_s=T,
pour les trois arêtes $(a_i, b_i), i = 0, 1, 2$ du triangle K, tournant dans le sens trigonométrique,
calculer l'aire des trois triangles (a_i, b_i, p)
si les trois aires sont positives alors $p \in K$ (stop),
sinon nous choisirons comme nouveau triangle K l'un des triangles adjacent à l'une des arêtes associées à une aire négative (les ambiguïtés sont traitées aléatoirement). Si le sommet recherché est à l'extérieur, nous retournerons une arête frontière (ici un triangle dégénéré).

Exercice 2.3 Prouver que l'algorithme 3.3 précédent marche si le maillage est convexe.

Listing 2.3 *(Mesh.cpp)*

```
#include "Mesh.hpp"

using namespace std;

namespace bamg {
  int verbosity=10;                                    //    niveau d'impression


//   ////////////////////////////////////////////////////////////////////////

  void  swap(Triangle *t1,short a1,
             Triangle *t2,short a2,
             Vertex *s1,Vertex *s2,Icoor2 det1,Icoor2 det2)
  {
    //    swap
    //    ----------------------------------------
    //    les 2 numéro de l arête dans les 2 triangles
    //
    //                  sb                    sb
```

```
    //                / | \                    /   \
    //           as1/  |  \                  /a2   \
    //             /   |   \                /    t2 \
    //          s1 /t1  | t2 \s2  -->   s1 /---as2---\s2
    //             \  a1|a2  /             \   as1   /
    //              \   |   /               \ t1    /
    //               \  |  / as2             \   a1/
    //                \ | /                   \   /
    //                 sa                       sa
    //    ----------------------------------------
    int as1 = NextEdge[a1];
    int as2 = NextEdge[a2];
                                            //    int ap1 = PreviousEdge[a1];
                                            //    int ap2 = PreviousEdge[a2];
    (*t1)(VerticesOfTriangularEdge[a1][1]) = s2;                //    avant sb
    (*t2)(VerticesOfTriangularEdge[a2][1]) = s1 ;               //    avant sa
                                    //    mise a jour des 2 adjacences externes
    TriangleAdjacent taas1 = t1->Adj(as1),    taas2 = t2->Adj(as2),
      tas1(t1,as1), tas2(t2,as2),   ta1(t1,a1),  ta2(t2,a2);
                                                  //    externe haut gauche
    taas1.SetAdj2(ta2,taas1.GetAllFlag_UnSwap());
                                                   //    externe bas droite
    taas2.SetAdj2(ta1, taas2.GetAllFlag_UnSwap());
                                                         //    interne
    tas1.SetAdj2(tas2);

    t1->det = det1;
    t2->det = det2;

    t1->SetTriangleContainingTheVertex();
    t2->SetTriangleContainingTheVertex();

  }                                                      //    end swap

//   ////////////////////////////////////////////////////////////////////////

  int SwapForForcingEdge(Vertex   *  & pva ,Vertex  * &   pvb ,
                         TriangleAdjacent & tt1,Icoor2 & dets1,
                         Icoor2 & detsa,Icoor2 & detsb, int & NbSwap)
  {                 //    l'arête ta coupe l'arête pva,pvb de cas apres le swap
                                                    //    sa coupe toujours
          //    on cherche l'arête suivante on suppose que detsa > 0 et detsb < 0
    //    attention la routine échange pva et pvb

    if(tt1.Locked()) return 0;                          //    frontiere croise

    TriangleAdjacent tt2 = Adj(tt1);
    Triangle *t1=tt1,*t2=tt2;                   //    les 2 triangles adjacent
    short a1=tt1,a2=tt2;   //    les 2 numéros de l'arête dans les 2 triangles
    assert ( a1 >= 0 && a1 < 3 );

    Vertex & sa= (* t1)[VerticesOfTriangularEdge[a1][0]];
                   //    Vertex & sb= (*t1)[VerticesOfTriangularEdge[a1][1]];
    Vertex & s1= (*t1)[OppositeVertex[a1]];
    Vertex & s2= (*t2)[OppositeVertex[a2]];
```

```
Icoor2 dets2 = det(*pva,*pvb,s2);
Icoor2 det1=t1->det , det2=t2->det;
Icoor2 detT = det1+det2;
assert((det1>0 ) && (det2 > 0));
assert ( (detsa < 0) && (detsb >0) );    //    [a,b] coupe la droite va,bb
Icoor2 ndet1 = bamg::det(s1,sa,s2);
Icoor2 ndet2 = detT - ndet1;

int ToSwap =0;                                          //   pas de échange
if ((ndet1 >0) && (ndet2 >0))
  {                                                    //    on peut échanger
    if ((dets1 <=0 && dets2 <=0) || (dets2 >=0 && detsb >=0))
      ToSwap =1;
    else                                              //    échange aléatoire
      if (BinaryRand())
        ToSwap =2;
  }

if (ToSwap) NbSwap++,
              bamg::swap(t1,a1,t2,a2,&s1,&s2,ndet1,ndet2);

int ret=1;

if (dets2 < 0) {                                                  //    haut
  dets1 = ToSwap? dets1 : detsa;
  detsa = dets2;
  tt1 =  Previous(tt2);}
else if (dets2 > 0){                                               //    bas
  dets1 = ToSwap? dets1 : detsb;
  detsb = dets2;
  if(!ToSwap) tt1 =  Next(tt2);
}
else {                                                 //    changement de sens
  ret = -1;
  Exchange(pva,pvb);
  Exchange(detsa,detsb);
  Exchange(dets1,dets2);
  Exchange(tt1,tt2);
  dets1=-dets1;
  dets2=-dets2;
  detsa=-detsa;
  detsb=-detsb;

  if (ToSwap)
    if (dets2 < 0) {                                              //    haut
      dets1 = (ToSwap? dets1 : detsa);
      detsa = dets2;
      tt1 =  Previous(tt2);}
    else if (dets2 > 0){                                           //    bas
      dets1 = (ToSwap? dets1 : detsb);
      detsb =  dets2;
      if(!ToSwap) tt1 =  Next(tt2);
    }
    else {                                  //    on a enfin fini le forçage
      tt1 = Next(tt2);
```

```
        ret =0;}
    }
    return ret;
  }

//   ////////////////////////////////////////////////////////////////////////

  int ForceEdge(Vertex &a, Vertex & b,TriangleAdjacent & taret)
  {

    int NbSwap =0;
    assert(a.t && b.t);              //    les 2 sommets sont dans le maillage
    int k=0;
    taret=TriangleAdjacent(0,0);                                    //    erreur

    TriangleAdjacent tta(a.t,EdgesVertexTriangle[a.vint][0]);
    Vertex   *v1, *v2 = tta.EdgeVertex(0),*vbegin =v2;
                        //    on tourne autour du sommet a dans le sens trigo.

    Icoor2 det2 = v2? det(*v2,a,b): -1 , det1;
    if(v2)                                                          //    cas normal
      det2 = det(*v2,a,b);
    else {                            //    pas de chance sommet ∞, au suivant
      tta= Previous(Adj(tta));
      v2 = tta.EdgeVertex(0);
      vbegin =v2;
      assert(v2);
      det2 = det(*v2,a,b);
    }


    while (v2!= &b) {
      TriangleAdjacent tc = Previous(Adj(tta));
      v1 = v2;
      v2 = tc.EdgeVertex(0);
      det1 = det2;
      det2 =  v2? det(*v2,a,b): det2;

      if((det1 < 0) && (det2 >0)) {
                                                //    on essaye de forcé l'arête
        Vertex * va = &a, *vb = &b;
        tc = Previous(tc);
        assert ( v1 && v2);
        Icoor2 detss = 0,l=0,ks;
        while ((ks=SwapForForcingEdge(  va,  vb, tc, detss, det1,det2,NbSwap)))
          if(l++ > 1000000) {
            cerr « " Loop in forcing Egde AB" « endl;
            MeshError(990);
          }
        Vertex *aa = tc.EdgeVertex(0), *bb = tc.EdgeVertex(1);
        if (( aa == &a ) && (bb == &b) ||  (bb ==  &a ) && (aa == &b)) {
          tc.SetLock();
          a.DelaunayOptim(1);
          b.DelaunayOptim(1);
          taret = tc;
          return NbSwap;
```

```
        }
      }
      tta = tc;
      assert(k++<2000);
      if ( vbegin == v2 ) return -1;     //    erreur la frontière est croisée
    }

    tta.SetLock();
    taret=tta;
    a.DelaunayOptim(1);
    b.DelaunayOptim(1);
    return NbSwap;
  }

//   ////////////////////////////////////////////////////////////////////////

  int Triangle::DelaunaySwap(short a){
    if(a/4!=0) return 0;                                        //    arête lock

    Triangle *t1=this,*t2=at[a];               //    les 2 triangles adjacents
    short a1=a,a2=aa[a];   //    les 2 numéros de l'arête dans les 2 triangles
    if(a2/4!=0) return 0;                                       //    arête lock

    Vertex  *sa=t1->ns[VerticesOfTriangularEdge[a1][0]];
    Vertex  *sb=t1->ns[VerticesOfTriangularEdge[a1][1]];
    Vertex  *s1=t1->ns[OppositeVertex[a1]];
    Vertex  *s2=t2->ns[OppositeVertex[a2]];

    Icoor2 det1=t1->det , det2=t2->det;
    Icoor2 detT = det1+det2;
    Icoor2 detA = Abs(det1) + Abs(det2);
    Icoor2 detMin = Min(det1,det2);

    int OnSwap = 0;
                              //    si 2 triangles infini (bord) => detT = -2;
    if (sa == 0) {                      //    les deux triangles sont frontières
      det2=bamg::det(s2->i,sb->i,s1->i);
      OnSwap = det2 >0;}
    else if (sb == 0) {                 //    les deux triangles sont frontières
      det1=bamg::det(s1->i,sa->i,s2->i);
      OnSwap = det1 >0;}
    else if(( s1!= 0) && (s2!= 0) ) {
      det1 = bamg::det(s1->i,sa->i,s2->i);
      det2 = detT - det1;
      OnSwap = (Abs(det1) + Abs(det2)) < detA;

      Icoor2 detMinNew=Min(det1,det2);
      if ( ! OnSwap &&(detMinNew>0)) {
        OnSwap = detMin ==0;
        if ( ! OnSwap) {

          while (1)
            {
                                      //    critère de Delaunay pure isotrope
              Icoor2 xb1 = sb->i.x - s1->i.x,
                x21 = s2->i.x - s1->i.x,
```

```
              yb1 = sb->i.y - s1->i.y,
              y21 = s2->i.y - s1->i.y,
              xba = sb->i.x - sa->i.x,
              x2a = s2->i.x - sa->i.x,
              yba = sb->i.y - sa->i.y,
              y2a = s2->i.y - sa->i.y;
            double
              cosb12 =  double(xb1*x21 + yb1*y21),
              cosba2 =  double(xba*x2a + yba*y2a) ,
              sinb12 = double(det2),
              sinba2 = double(t2->det);

            OnSwap =  ((double) cosb12 * (double)  sinba2)
              <  ((double) cosba2 * (double) sinb12);

            break;
          }

      }                                                    //    OnSwap
    }                          //    ( ! OnSwap &&(det1 > 0) && (det2 > 0) )
  }

  if( OnSwap )
    bamg::swap(t1,a1,t2,a2,s1,s2,det1,det2);

  return OnSwap;
}

//   ////////////////////////////////////////////////////////////////////////

 void Triangles::Add( Vertex & s,Triangle * t, Icoor2 * det3)
 {
   //    -----------------------------
   //                s2
   //
   //                /|\
   //               / | \
   //              /  |  \
   //       tt1   /   |   \ tt0
   //            /    |s   \
   //           /     .     \
   //          /  .       '   \
   //         / .             ' \
   //         ----------------
   //       s0       tt2       s1
   //   ----------------------------

   Triangle * tt[3];                            //    les 3 nouveaux Triangles
   Vertex &s0 = (* t)[0], &s1=(* t)[1], &s2=(* t)[2];
   Icoor2  det3local[3];

   int infv = &s0?  ((  &s1? ( &s2 ? -1 : 2) : 1  )) : 0;

   int nbd0 =0;
   int izerodet=-1,iedge;        //    izerodet = arête contenant le sommet s
   Icoor2 detOld = t->det;
```

```
    if ( ( infv <0 ) && (detOld <0) ||  ( infv >=0  ) && (detOld >0) )
      {
        cerr « "  infv " « infv « " det = " « detOld « endl;
        MeshError(3);                        //    il y a des sommets confondus
      }

    if (!det3) {
      det3 = det3local;                                                  //    alloc
      if ( infv<0 ) {
        det3[0]=bamg::det(s ,s1,s2);
        det3[1]=bamg::det(s0,s ,s2);
        det3[2]=bamg::det(s0,s1,s );}
      else {
                      //    one of &s1 &s2 &s0 is NULL so (&si || &sj) <=>!&sk
        det3[0]=  &s0? -1  : bamg::det(s ,s1,s2);
        det3[1]=  &s1? -1 : bamg::det(s0,s ,s2);
        det3[2]=  &s2? -1 : bamg::det(s0,s1,s );}}


    if (!det3[0]) izerodet=0,nbd0++;
    if (!det3[1]) izerodet=1,nbd0++;
    if (!det3[2]) izerodet=2,nbd0++;

    if  (nbd0 >0 )                              //    s est sur une ou des arêtes
      if (nbd0 == 1) {
        iedge = OppositeEdge[izerodet];
        TriangleAdjacent ta = t->Adj(iedge);

                                                //    le point est sur une arête
              //    si l'arête est frontière on ajoute le point dans la partie
externe
        if ( t->det >=0) {                                  //    triangle interne
          if ((( Triangle *) ta)->det < 0 ) {
                                                  //    add in outside triangle
            Add(s,( Triangle *) ta);
            return;}
        }}
      else {
        cerr « " bug  " « nbd0 «endl;
        cerr « " Bug double points in " « endl;
        MeshError(5);}

    tt[0]= t;
    tt[1]= &triangles[nbt++];
    tt[2]= &triangles[nbt++];

    if (nbt>nbtx) {
      cerr « " No enougth triangles " « endl;
      MeshError(999);                               //    pas assez de triangle
    }

    *tt[1]=   *tt[2]= *t;


    (* tt[0])(OppositeVertex[0])=&s;
```

```
    (* tt[1])(OppositeVertex[1])=&s;
    (* tt[2])(OppositeVertex[2])=&s;

    tt[0]->det=det3[0];
    tt[1]->det=det3[1];
    tt[2]->det=det3[2];

                                //    mise à jour des adj. des triangles externe
    tt[0]->SetAdjAdj(0);
    tt[1]->SetAdjAdj(1);
    tt[2]->SetAdjAdj(2);
                               //    mise à jour des adj. des 3 triangle interne
    const int i0 = 0;
    const int i1= NextEdge[i0];
    const int i2 = PreviousEdge[i0];

    tt[i0]->SetAdj2(i2,tt[i2],i0);
    tt[i1]->SetAdj2(i0,tt[i0],i1);
    tt[i2]->SetAdj2(i1,tt[i1],i2);

    tt[0]->SetTriangleContainingTheVertex();
    tt[1]->SetTriangleContainingTheVertex();
    tt[2]->SetTriangleContainingTheVertex();


                                  //    échange si le point s est sur une arête
    if(izerodet>=0) {
      int rswap =tt[izerodet]->DelaunaySwap(iedge);

      if (!rswap)
        {
          cout « " Pb swap the point s is on a edge =>swap " « iedge « endl;
          MeshError(98);
        }
    }
  }

//   ////////////////////////////////////////////////////////////////////////

  void Triangles::Insert()
  {
    NbUnSwap=0;
    if (verbosity>2)
      cout « " - Insert initial " « nbv « " vertices " « endl;

    SetIntCoor();
    int i;
    Vertex**  ordre=new (Vertex* [nbvx]);

    for (i=0;i<nbv;i++)
      ordre[i]= &vertices[i];

                                       //    construction d'un ordre aléatoire
    const int PrimeNumber=  (nbv % 567890621L) ? 567890629L : 567890621L;
    int k3 = rand()%nbv;
    for (int is3=0; is3<nbv; is3++)
```

```
  ordre[is3]= &vertices[k3 = (k3 + PrimeNumber)% nbv];

for (i=2; det( ordre[0]->i, ordre[1]->i, ordre[i]->i ) == 0;)
  if  ( ++i >= nbv) {
    cerr « " All the vertices are aline " « endl;
    MeshError(998); }

                                        //    échange i et 2 dans ordre afin
                      //    que les 3 premiers sommets ne soit pas alignés
Exchange( ordre[2], ordre[i]);

           //    on ajoute un point à l'infini pour construire le maillage
            //    afin d'avoir une définition simple des arêtes frontières
nbt = 2;
 //    on construit un maillage trival formé d'une arête et de 2 triangles
                               //    construit avec le 2 arêtes orientéess
Vertex *  v0=ordre[0], *v1=ordre[1];

triangles[0](0) = 0;              //    sommet pour infini (la frontière)
triangles[0](1) = v0;
triangles[0](2) = v1;

triangles[1](0) = 0;              //    sommet pour infini (la frontière)
triangles[1](2) = v0;
triangles[1](1) = v1;

const int e0 = OppositeEdge[0];
const int e1 = NextEdge[e0];
const int e2 = PreviousEdge[e0];
triangles[0].SetAdj2(e0, &triangles[1] ,e0);
triangles[0].SetAdj2(e1, &triangles[1] ,e2);
triangles[0].SetAdj2(e2, &triangles[1] ,e1);

triangles[0].det = -1;                                  //    faux triangles
triangles[1].det = -1;                                  //    faux triangles

triangles[0].SetTriangleContainingTheVertex();
triangles[1].SetTriangleContainingTheVertex();
nbtf = 2;             //    ici, il y a deux triangles frontières invalide

                         //    ----- on ajoute les sommets un à un -------

int NbSwap=0;

if (verbosity>3) cout « " - Begin of insertion process " « endl;

Triangle * tcvi=triangles;

for (int icount=2; icount<nbv; icount++)
  {
    Vertex *vi  = ordre[icount];
    Icoor2 dete[3];
    tcvi = FindTriangleContening(vi->i,dete,tcvi);
    Add(*vi,tcvi,dete);
    NbSwap += vi->DelaunayOptim(1);
  }                                          //    fin de boucle en icount
```

```
    if (verbosity>3)
      cout « "    NbSwap of insertion " «    NbSwap
           « " NbSwap/Nbv " «  (float) NbSwap / (float) nbv
           « " NbUnSwap " « NbUnSwap « " Nb UnSwap/Nbv "
           « (float)NbUnSwap /(float) nbv
           «endl;
    NbUnSwap = 0;
    delete [] ordre;
  }

//   ////////////////////////////////////////////////////////////////////////

  void Triangles::RandomInit()
  {
    nbv = nbvx;
    for (int i = 0; i < nbv; i++)
      {
        vertices[i].r.x= rand();
        vertices[i].r.y= rand();
        vertices[i].Label = 0;
      }
  }

//   ////////////////////////////////////////////////////////////////////////

  Triangles::~Triangles()
  {
    if(vertices)  delete [] vertices;
    if(triangles) delete [] triangles;
     PreInit(0);                                //    met to les sommets à zèro

  }

//   ////////////////////////////////////////////////////////////////////////

  void Triangles::SetIntCoor()
  {
    pmin =  vertices[0].r;
    pmax =  vertices[0].r;

                              //    recherche des extrema des sommets pmin,pmax
    int i;
    for (i=0;i<nbv;i++) {
      pmin.x = Min(pmin.x,vertices[i].r.x);
      pmin.y = Min(pmin.y,vertices[i].r.y);
      pmax.x = Max(pmax.x,vertices[i].r.x);
      pmax.y = Max(pmax.y,vertices[i].r.y);
    }
    R2 DD = (pmax-pmin)*0.05;
    pmin = pmin-DD;
    pmax = pmax+DD;
    coefIcoor= (MaxICoor)/(Max(pmax.x-pmin.x,pmax.y-pmin.y));
    assert(coefIcoor >0);

                                    //    génération des coordonnées entières
```

```
    for (i=0;i<nbv;i++) {
      vertices[i].i = toI2(vertices[i].r);
    }

    //    calcule des determinants si nécessaire
    int Nberr=0;
    for (i=0;i<nbt;i++)
      {
        Vertex & v0 = triangles[i][0];
        Vertex & v1 = triangles[i][1];
        Vertex & v2 = triangles[i][2];
        if ( &v0 && &v1 &&  &v2 )                    //    un bon triangle;
          {
            triangles[i].det= det(v0,v1,v2);
            if (triangles[i].det <=0 && Nberr++ <10)
              {
                if(Nberr==1)
                    cerr « "+++ Fatal Error "
                         « "(SetInCoor)  Error :  area of Triangle < 0\n";
              }
          }
        else
          triangles[i].det= -1;              //    le triangle est dégénéré;
      }
    if (Nberr) MeshError(899);
  }

//   ////////////////////////////////////////////////////////////////////////

  int  Triangle::DelaunayOptim(short i)
  {
                              //    nous tournons dans le sens trigonométrique
    int NbSwap =0;
    Triangle  *t = this;
    int k=0,j =OppositeEdge[i];
    int jp = PreviousEdge[j];
                           //    initialise tp, jp avec l'arête précédente de j
    Triangle *tp= at[jp];
    jp = aa[jp]&3;
    do {
      while (t->DelaunaySwap(j))
        {
          NbSwap++;
          assert(k++<20000);
          t=  tp->at[jp];
          j=  NextEdge[tp->aa[jp]&3];
        }                                          //    on a fini ce triangle
      tp = t;
      jp = NextEdge[j];

      t=  tp->at[jp];
      j=  NextEdge[tp->aa[jp]&3];

    } while( t != this);
    return NbSwap;
  }
```

```
//   ////////////////////////////////////////////////////////////////////////

  void Triangles::PreInit(int inbvx)
  {
                    //    fonction d'initialisation ------------------------
                            //    ----------------------------------------
    srand(19999999);
    NbOfSwapTriangle =0;
    nbiv=0;
    nbv=0;
    nbvx=inbvx;
    nbt=0;
    nbtx=2*inbvx-2;

    if (inbvx) {
      vertices=new Vertex[nbvx];
      assert(vertices);
      triangles=new Triangle[nbtx];
      assert(triangles);}
    else {
      vertices=0;
      triangles=0;
      nbtx=0;
    }
  }

//   ////////////////////////////////////////////////////////////////////////

  Triangle * Triangles::FindTriangleContening(const I2 & B,
                                              Icoor2 dete[3],
                                              Triangle *tstart) const
  {                                                          //    entrée: B
                                                             //    sortie: t
                                                       //    sortie : dete[3]
                                        //    t triangle et s0,s1,s2 le sommet de t
                      //    dete[3] =  det(B,s1,s2) , det(s0,B,s2), det(s0,s1,B)
            //    avec det(a,b,c) = -1 si l'un des 3 sommet a,b,c est NULL (infini)
    Triangle * t=0;
    int j,jp,jn,jj;
    t=tstart;
    assert(t>= triangles && t < triangles+nbt);
    Icoor2  detop;
    int kkkk =0;                                  //    nombre de triangle testé

    while ( t->det < 0)
      {            //    le triangle initial est externe (une arête frontière)
        int k0=(*t)(0)?  ((  (*t)(1)? ( (*t)(2)? -1 : 2) : 1  )) : 0;
        assert(k0>=0);                                    //    k0 the NULL vertex
        int k1=NextVertex[k0],k2=PreviousVertex[k0];
        dete[k0]=det(B,(*t)[k1],(*t)[k2]);
        dete[k1]=dete[k2]=-1;
        if (dete[k0] > 0)                      //    B n'est pas dans le domaine
          return t;
        t = t->TriangleAdj(OppositeEdge[k0]);
        assert(kkkk++ < 2);
```

```
      }

    jj=0;
    detop = det(*(*t)(VerticesOfTriangularEdge[jj][0]),
                *(*t)(VerticesOfTriangularEdge[jj][1]),B);

    while(t->det  > 0 )
      {
        assert( kkkk++ < 2000 );
        j= OppositeVertex[jj];

        dete[j] = detop;                                    //   det(*b,*s1,*s2);
        jn = NextVertex[j];
        jp = PreviousVertex[j];
        dete[jp]= det(*(*t)(j),*(*t)(jn),B);
        dete[jn] = t->det-dete[j] -dete[jp];


        int k=0,ii[3];                        //    compte le nombre k de dete < 0
        if (dete[0] < 0 ) ii[k++]=0;
        if (dete[1] < 0 ) ii[k++]=1;
        if (dete[2] < 0 ) ii[k++]=2;
                                                          //    0 => on a trouvé
                                                 //    1 => on va dans cet direction
                                        //    2 => deux choix, on choisi aléatoirement

        if (k==0)
          break;
        if (k == 2 && BinaryRand())
          Exchange(ii[0],ii[1]);
        assert ( k  < 3);
        TriangleAdjacent t1 = t->Adj(jj=ii[0]);
        if ((t1.det() < 0 ) && (k == 2))
          t1 = t->Adj(jj=ii[1]);
        t=t1;
        j=t1;
        detop = -dete[OppositeVertex[jj]];
        jj = j;
      }

    if (t->det<0)                          //    triangle externe (arête frontière)
      dete[0]=dete[1]=dete[2]=-1,dete[OppositeVertex[jj]]=detop;

    return t;
  }
}                       //                            ----- end namespace -------
```

Finalement, le programme principal est :

Listing 2.4 *(main.cpp)*

```
#include <cassert>
#include <fstream>
#include <iostream>
using namespace std;
#include "Mesh.hpp"

using namespace bamg;

void GnuPlot(const Triangles & Th,const char *filename) {
   ofstream ff(filename);
   for (int k=0;k<Th.nbt;k++)
    {
     const Triangle &K=Th[k];
     if (K.det>0)                                          //    true triangle
      {
        for (int k=0;k<4;k++)
          ff « K[k%3].r « endl;
         ff « endl « endl;}
   }
  }
int main(int argc, char ** argv)
{
 int nbv = argc > 1? atoi(argv[1]) : 100;
 int na= argc > 2? atoi(argv[2]) : 0;
 int nb= argc > 3? atoi(argv[3]) : nbv-1;
 assert ( na!= nb && na >=0 && nb >=0 && na < nbv && nb < nbv);

 Triangles Th(nbv);
 Th.RandomInit();
 Th.Insert();
 TriangleAdjacent ta(0,0);
 GnuPlot(Th,"Th0.gplot");
 int nbswp =  ForceEdge(Th(na),Th(nb),ta);
 if(nbswp<0) { cerr « " -Impossible de force l'arête " « endl;}
 else {
   cout « " Nb de swap pour force l'arete [" « na « " " « nb
« "] =" « nbswp « endl;
   GnuPlot(Th,"Th1.gplot"); }

 return(0);
}
```

Exercice 2.4

Écrire un programme qui maille le disque unité avec un pas de maillage en $1/n$, avec des points équirépartis sur des cercles concentriques de rayons $i/r, i = 1..n$.

Exercice 2.5

Soit *Rot* un rotation du plan d'angle 11°, écrire un programme qui maille le carré $Rot(]0,1[^2)$ découpé en 10×10 mailles. Calculer la surface des triangles, expliquer le résultat (remarquer que le domaine à mailler n'est pas convexe à cause de la représentation flottante des nombres dans l'ordinateur).
Ajouter le forçage de la frontière et utiliser l'algorithme (3.2)de coloriage des sous-domaines pour supprimer tous les triangles qui sont hors du domaine.
Vérifier l'utilité des coordonnées entières sur cet exemple.

Exercice 2.6

Faire une étude de complexité de l'algorithme, premièrement avec les points générés aléatoirement, deuxièmement avec des points générés sur un même cercle. Trouver les parties de l'algorithme de complexité plus que linéaire.

Exercice 2.7

Améliorer le choix du triangle de départ dans la recherche du triangle contenant un sommet. Pour cela, nous stockons un sommet déjà inséré par maille de la grille régulière (par exemple, 50×50). Nous partirons du triangle associé au sommet le plus proche dans la grille.

Exercice 2.8

Écrire le générateur de maillage complet qui lit les données définies en §3.1.3 à partir d'un fichier et qui le sauve dans un fichier au format `msh` (voir §**??**).

Bibliographie

[J. Barton, Nackman-1994] J. Barton, L. Nackman *Scientific and Engineering, C++*, Addison-Wesley, 1994.

[Bejan-1993] A. Bejan :*Heat transfer*, John Wiley & Sons, 1993.

[Ciarlet-1978] P.G. Ciarlet , *The Finite Element Method*, North Holland. n and meshing. Applications to Finite Elements, Hermès, Paris, 1978.

[Ciarlet-1982] P. G. Ciarlet *Introduction à l'analyse numérique matricielle et à l'optimisation*, Masson ,Paris,1982.

[Ciarlet-1991] P.G. Ciarlet , Basic Error Estimates for Elliptic Problems, in Handbook of Numerical Analysis, vol II, Finite Element methods (Part 1), P.G. Ciarlet and J.L. Lions Eds, North Holland, 17-352, 1991.

[Dupin-1999] S. Dupin *Le langage C++*, Campus Press 1999.

[Forsythe, Wasow-1960] G. E. Forsythe, W. R. Wasow :*Finite-difference methods for partial differential equations*, John Wiley & Sons, 1960.

[Frey, George-1999] P. J. Frey, P-L George *Maillages*, Hermes, Paris, 1999.

[George,Borouchaki-1997] P.L. George et H. Borouchaki , *Triangulation de Delaunay et maillage. Applications aux éléments finis*, Hermès, Paris, 1997. Also as
P.L. George and H. Borouchaki , *Delaunay triangulation and meshing. Applications to Finite Elements*, Hermès, Paris, 1998.

[FreeFem++] F. Hecht, O. Pironneau, K. Othsuka FreeFem++ : Manual http ://www.freefem.org/

[Hirsh-1988] C. Hirsch *Numerical computation of internal and external flows*, John Wiley & Sons, 1988.

[Koenig-1995] A. Koenig (ed.) : *Draft Proposed International Standard for Information Systems - Programming Language C++*, ATT report X3J16/95-087 (ark@research.att.com)., 1995

[Knuth-1975] D.E. Knuth , The Art of Computer Programming, 2nd ed., *Addison-Wesley*, Reading, Mass, 1975.

[Knuth-1998a] D.E. Knuth The Art of Computer Programming, Vol I : Fundamental algorithms, *Addison-Wesley*, Reading, Mass, 1998.

[Knuth-1998b] D.E. Knuth The Art of Computer Programming, Vol III : Sorting and Searching, *Addison-Wesley*, Reading, Mass, 1998.

[Lachand-Robert] T. Lachand-Robert, A. Perronnet (http://www.ann.jussieu.fr/courscpp/)

[Löhner-2001] R. Löhner Applied CFD Techniques, Wiley, Chichester, England, 2001.

[Lucquin et Pironneau-1996] B. Lucquin, O. Pironneau *Introduction au calcul scientifique*, Masson 1996.

[Numerical Recipes-1992] W. H. Press, W. T. Vetterling, S. A. Teukolsky, B. P. Flannery : *Numerical Recipes : The Art of Scientific Computing*, Cambridge University Press, 1992.

[Raviart,Thomas-1983] P.A. Raviart et J.M. Thomas, *Introduction à l'analyse numérique des équations aux dérivées partielles*, Masson, Paris, 1983.

[Richtmyer et Morton-1667] R. D. Richtmyer, K. W. Morton : *Difference methods for initial-value problems*, John Wiley & Sons, 1967.

[Shapiro-1991] J. Shapiro *A C++ Toolkit*, Prentice Hall, 1991.

[Stroustrup-1997] B. Stroustrup *The C++ Programming Language*, Third Edition, Addison Wesley, 1997.

[Wirth-1975] N Wirth *Algorithms + Dat Structure = program*, Prentice-Hall, 1975.

[Aho et al -1975] A. V. Aho, R. Sethi, J. D. Ullman , Compilers, Principles, Techniques and Tools, Addison-Wesley, Hardcover, 1986.

[Lex Yacc-1992] J. R. Levine, T. Mason, D. Brown Lex & Yacc, O'Reilly & Associates, 1992.

[Campione et Walrath-1996] M. Campione and K. Walrath The Java Tutorial : Object-Oriented Programming for the Internet, Addison-Wesley, 1996.

Voir aussi Integrating Native Code and Java Programs. `http://java.sun.com/nav/read/Tutorial/native1.1/index.html`.

[Daconta-1996] C. Daconta Java for C/C++ Programmers, Wiley Computer Publishing, 1996.

[Flanagan-1996] D. Flanagan Java in a Nutshell, O'Reilly and Associates, 1996.

[Bossavit-1998] A. Bossavit Computational Electromagnetism. Academic Press, 1998

[Mohammadi-2001] B. Mohammadi, O. Pironneau : *Applied Optimal Shape Design* Oxford University Press, 2001

[Hasliger-2003] J Hasliger and R. Makinen Introduction to shape optimization, Saim serie, Philadelphia, 2003

[Griewank-2001] Griewank A., *Computational differentiation,* Springer, 2001.