Maillage 2d, 3d, adaptation

Frédéric Hecht
Université Pierre et Marie Curie, Paris

20 mars 2006

Table des matières

1 Quelques éléments de syntaxe **5**

1.1 Les déclarations du C++ 6

1.2 Comment Compile et éditer de liens 7

1.3 Quelques règles de programmation 10

1.4 Vérificateur d'allocation 13

1.5 Compréhension des constructeurs, destructeurs et des passages d'arguments 14

2 Exemples de Classe C++ **15**

2.1 Le Plan $\mathbb{R}^2$ 15

2.1.1 La classe R2 15

2.1.2 Utilisation de la classe R2 17

2.2 Les classes tableaux 18

2.2.1 Version simple d'une classe tableau 19

2.2.2 les classes RNM 21

2.2.3 Exemple d'utilisation de classes RNM 22

2.2.4 Une resolution de systéme linéaire avec le gradient conjugué 25

2.3 Des classes pour les Graphes 30

2.4 Definition et Mathématiques 30

2.4.1 Premiere Implémentation 32

2.5 Maillage et Triangulation 2D 37

2.5.1 La classe `Label` 37

2.5.2 La classe `Vertex` (modélisation des sommets) 38

2.5.3 La classe `Triangle` (modélisation des triangles) 39

2.5.4 La classe `Mesh` (modélisation du maillage) 40

3 Algorithmes **45**

3.1 Chaînes et Chaînages 45

3.1.1 Introduction 45
3.1.2 Construction de l'image réciproque d'une fonction 45
3.1.3 Construction des arêtes d'un maillage 46
3.1.4 Construction des triangles contenant un sommet donné 49
3.1.5 Construction de la structure d'une matrice morse 50
3.2 Algorithmes de trie 53
3.3 Algorithmes de recherche 53
3.3.1 Arbre quaternaire 54
3.4 Parcours récurcif d'un maillage 54

4 Construction d'un maillage bidimensionnel 55

4.1 Bases théoriques 55
4.1.1 Notations 55
4.1.2 Introduction 56
4.1.3 Les données pour construire un maillage 57
4.1.4 Triangulation et Convexifié 58
4.1.5 Maillage de Delaunay-Voronoï 58
4.1.6 Forçage de la frontière 65
4.1.7 Recherche de sous-domaines 67
4.1.8 Génération de points internes 68
4.2 Algorithme de construction du maillage 69
4.3 Programme C++ 70

Chapitre 1

Quelques éléments de syntaxe

Il y a tellement de livres sur la syntaxe du C++ qu'il me paraît déraisonnable de réécrire un chapitre sur ce sujet, je vous propose le livre de Thomas Lachand-Robert qui est disponible sur la toile à l'adresse suivante `http://www.ann.jussieu.fr/courscpp/`, ou le cours C,C++ [?] plus moderne aussi disponible sur la toile `http://casteyde.christian.free.fr/cpp/cours` ou bien sur d'utiliser le livre The C++ , programming language [Stroustrup-1997]

Je veux décrire seulement quelques trucs et astuces qui sont généralement utiles comme les déclarations des types de bases et l'algèbre de typage.

Donc à partir de ce moment je suppose que vous connaissez, quelques rudiments de la syntaxe C++ . Ces rudiments que je sais difficile, sont (pour le connaître il suffit de comprendre ce qui est écrit après) :

- Les types de base, les définitions de pointeur et référence (je vous rappelle qu'une référence est défini comme une variable dont l'adresse mémoire est connue et cet adresse n'est pas modifiable, donc une référence peut être vue comme une pointeur constant automatiquement déférencé, ou encore donné un autre nom à une zone mémoire de la machine).
- L'écriture d'un fonction, d'un prototypage,
- Les structures de contrôle associée aux mots clefs suivants : `if, else, switch, case, default, while, for, repeat, continue, break`.
- L'écriture d'une classe avec constructeur et destructeur, et des fonctions membres.
- Les passages d'arguments
  - par valeur (type de l'argument sans `&`), donc une copie de l'argument est passée à la fonction. Cette copie est créée avec le constructeur par copie, puis est détruite avec le destructeur. L'argument ne peut être modifié dans ce cas.
  - par référence (type de l'argument avec `&`) donc l'utilisation du constructeur par copie.
  - par pointeur (le pointeur est passé par valeur), l'argument peut-être modifié.
  - paramètre non modifiable (cf. mot clef `const`).
  - La valeur retournée par copie (type de retour sans `&`) ou par référence (type de retour avec `&`)
- Polymorphisme et surcharge des opérateurs. L'appel d'une fonction est déterminée par son nom et par le type de ses arguments, il est donc possible de créer des fonctions de même nom pour des type différents. Les opérateurs n-naire (unaire n=1 ou binaire n=2) sont des fonctions à n argument de nom `operator ♣ (n-args )` où ♣ est l'un des

opérateurs du C++ :

```
+    -    *    /    %    ^    & |  ~   !    =    <      >        +=
-=   *=   /=   %=   ^=   &=   |= «  »   «=   »=   ==     !=       <=
>=   &&   ||   ++   -    ->*  , ->  []  ()   new  new[]  delete   delete[]
(T)
```

où (T est un type), et où `(n-args )` est la déclaration classique des n arguments. Remarque si opérateur est défini dans une classe alors le premier argument est la classe elle même et donc le nombre d'arguments est $n-1$.

- Les règles de conversion d'un type T en A par défaut qui sont génèrées à partir d'un constructeur `A(T)` dans la classe A ou avec l'opérateur de conversion `operator (A) ()` dans la classe T, `operator (A) (T)` hors d'une classe. De plus il ne faut pas oublier que C++ fait automatiquement un au plus un niveau de conversion pour trouver la bonne fonction ou le bon opérateurs.
- Programmation générique de base (c.f. `template`). Exemple d'écriture de la fonction `min` générique suivante `template<class T> T & min(T & a,T & b){return a<b? a :b;}`
- Pour finir, connaître seulement l'existence du macro générateur et ne pas l'utiliser.

1.1 Les déclarations du C++

Les types de base du C++ sont respectivement : `char`, `short`, `int`, `long`, `long long`, `float`, `double`, plus des pointeurs, ou des références sur ces types, des tableaux, des fonctions sur ces types. Le tout nous donne une algèbre de type qui n'est pas triviale.

Voilà les principaux types généralement utilisé pour des types `T`,`U` :

déclaration	Prototypage	description du type en français
`T * a`	`T *`	un pointeur sur `T`
`T a[10]`	`T[10]`	un tableau de `T` composé de 10 variable de type T
`T a(U)`	`T a(U)`	une fonction qui a `U` retourne un `T`
`T &a`	`T &a`	une référence sur un objet de type `T`
`const` `T a`	**`const`** `T`	un objet constant de type `T`
`T` **`const`** `* a`	`T` **`const`** `*`	un pointeur sur objet constant de type `T`
`T *` **`const`** `a`	`T *` **`const`**	un pointeur constant sur objet de type `T`
`T` **`const`** `*` **`const`** `a`	`T` **`const`** `*` **`const`**	un pointeur constant sur objet constant
`T * & a`	`T * &`	une référence sur un pointeur sur `T`
`T ** a`	`T **`	un pointeur sur un pointeur sur `T`
`T * a[10]`	`T *[10]`	un tableau de 10 pointeurs sur `T`
`T (* a)[10]`	`T (*)[10]`	un pointeur sur tableau de 10 `T`
`T (* a)(U)`	`T (*)(U)`	un pointeur sur une fonction `U` →`T`
`T (* a[])(U)`	`T (*[])(U)`	un tableau de pointeur sur des fonctions `U`→`T`
...		

Remarque il n'est pas possible de construire un tableau de référence car il sera impossible à initialiser.

Exemple d'allocation d'un tableau data de *ldata* pointeurs de fonctions de R à valeur dans

R :

```
R (**data)(R) = new (R (*[ldata])(R));
```

ou encore avec déclaration et puis allocation :

```
R (**data)(R); data = new (R (*[ldata])(R));
```

1.2 Comment Compile et éditer de liens

Comme en `C`; dans les fichiers `.cpp`, il faut mettre les corps des fonctions et de les fichiers `.hpp`, il faut mettre les prototype, et le définition des classes, ainsi que les fonctions `inline` et les fonctions `template`, Voilà, un exemple complète avec trois fichiers `a.hpp`,`a.cpp`,`tt.hpp`, et un Makefile dans `ftp://www.freefem.org/snc++/l1/a.tar.gz`

remarque, pour déarchiver un fichier `xxx.tar.gz`, il suffit d'entrer dans une fenêtre shell `tar zxvf xxx.tar.gz`.

Listing 1.1 *(a.hpp)*

```
class A { public:
  A();                                    //    constructeur de la class A
};
```

Listing 1.2 *(a.cpp)*

```
#include <iostream>
#include "a.hpp"
using namespace std;

A::A()
{
  cout « " Constructeur A par défaut " « this « endl;
}
```

Listing 1.3 *(tt.cpp)*

```
#include <iostream>
#include "a.hpp"
using namespace std;
int main(int argc,char ** argv)
```

```
{
  for(int i=0;i<argc;++i)
    cout « " arg " « i « " = " « argv[i] « endl;
  A a[10];                                   //    un tableau de 10 A
  return 0;                                  //    ok
}
```

les deux compilations et l'édition de liens qui génère un executable `tt` dans une fenêtre `Terminal`, avec des commandes de type `shell; sh, bash tcsh, ksh, zsh, ...` , sont obtenues avec les trois lignes :

```
[brochet:P6/DEA/sfemGC] hecht% g++ -c a.cpp
[brochet:P6/DEA/sfemGC] hecht% g++ -c tt.cpp
[brochet:P6/DEA/sfemGC] hecht% g++ a.o tt.o -o tt
```

Pour faire les trois choses en même temps, entrez :

```
[brochet:P6/DEA/sfemGC] hecht% g++  a.cpp tt.cpp -o tt
```

Puis, pour executer la commande `tt`, entrez par exemple :

```
[brochet:P6/DEA/sfemGC] hecht% ./tt aaa bb cc dd qsklhsqkfhsd " -----------
 arg 0 = ./tt
 arg 1 = aaa
 arg 2 = bb
 arg 3 = cc
 arg 4 = dd
 arg 5 = qsklhsqkfhsd
 arg 6 =  -----------
 Constructeur A par défaut 0xbfffeef0
 Constructeur A par défaut 0xbfffeef1
 Constructeur A par défaut 0xbfffeef2
 Constructeur A par défaut 0xbfffeef3
 Constructeur A par défaut 0xbfffeef4
 Constructeur A par défaut 0xbfffeef5
 Constructeur A par défaut 0xbfffeef6
 Constructeur A par défaut 0xbfffeef7
 Constructeur A par défaut 0xbfffeef8
 Constructeur A par défaut 0xbfffeef9
```

remarque : les `aaa bb cc dd qsklhsqkfhsd " ----------- "` après la commande `./tt` sont les paramètres de la commande est sont bien sur facultatif.

remarque, il est aussi possible de faire un `Makefile`, c'est à dire de créé le fichier :

Listing 1.4 *(Makefile)*

```
CPP=g++
CPPFLAGS= -g
LIB=
%.o:%.cpp
(caractere de tabulation  -->/)  $(CPP) -c $(CPPFLAGS) $^
tt: a.o tt.o
(caractere de tabulation  -->/)  $(CPP) tt.o  a.o -o tt
clean:
(caractere de tabulation  -->/) rm *.o tt

# les dependences
#
a.o: a.hpp  # il faut recompilé a.o si a.hpp change
tt.o: a.hpp  # il faut recompilé tt.o si a.hpp change
```

Pour l'utilisation :

- pour juste voir les commandes exécutées sans rien faire :
  ```
  [brochet:P6/DEA/sfemGC] hecht% make -n tt
  g++  -c  tt.cpp
  g++  -c  a.cpp
  g++  -o tt tt.o a.o
  ```
- pour vraiment compiler
  ```
  [brochet:P6/DEA/sfemGC] hecht% make  tt
  g++  -c  tt.cpp
  g++  -c  a.cpp
  g++  -o tt tt.o a.o
  ```
- pour recompiler avec une modification du fichier `a.hpp` via la command `touch` qui change la date de modification du fichier.
  ```
  [brochet:P6/DEA/sfemGC] hecht% touch a.hpp
  [brochet:P6/DEA/sfemGC] hecht% make tt
  g++  -c  tt.cpp
  g++  -c  a.cpp
  g++  -o tt tt.o a.o
  ```
  remarque : les deux fichiers sont bien à recompiler car il font un *include* du fichier `a.hpp`.
- pour nettoyer :
  ```
  [brochet:P6/DEA/sfemGC] hecht% touch a.hpp
  [brochet:P6/DEA/sfemGC] hecht% make clean
  rm *.o tt
  ```

Remarque : Je vous conseille très vivement d'utiliser un `Makefile` pour compiler vous programme.

Exercice 1.1 Ecrire un Makefile pour compile et tester tous les programmes `ftp://www.freefem.org/snc++/l1/exemple_de_base`

1.3 Quelques règles de programmation

Malheureusement, il est très facile de faire des erreurs de programmation, la syntaxe du C++ n'est pas toujours simple à comprendre et comme l'expressibilité du langage est très grande, les possibilités d'erreur sont innombrables. Mais avec un peu de rigueur, il est possible d'en éviter un grand nombre.

La plupart des erreurs sont dû à des problèmes des pointeurs (débordement de tableau, destruction multiple, oublie de destruction), retour de pointeur sur des variable locales.

Voilà quelques règles à respecté.

Règle 1.1 *absolue Dans une classe avec des pointeurs et avec un destructeur, il faut que les deux opérateurs de copie (création et affection) soient définis. Si vous considérez que ces deux opérateurs ne doivent pas exister alors les déclarez en privé sans les définir.*

```
class sans_copie { public:
   long * p;                                              //    un pointeur
   . . .
   sans_copie();
   ~sans_copie() { delete p;}
   private:
   sans_copie(const sans_copie &);         //    pas de constructeur par copie
   void operator=(const sans_copie &);         //    pas d'affection par copie
};
```

Dans ce cas les deux opérateurs de copies ne sont pas programmer pour qu'une erreur à l'édition des liens soit généré.

```
class avec_copie { public:
   long * p;                                              //    un pointeur
   ~avec_copie() { delete p;}
   . . .
   avec_copie();
   avec_copie(const avec_copie &);       //    construction par copie possible
   void operator=(const avec_copie &);       //    affection par copie possible
};
```

Par contre dans ce cas, il faut programmer les deux opérateurs construction et affectation par copie.

Effectivement, si vous ne définissez ses opérateurs, il suffit d'oublier une esperluette (&) dans un passage argument pour que plus rien ne marche, comme dans l'exemple suivante :

```
  class Bug{ public:
   long * p;                                              //    un pointeur
   Bug() p(new long[10]);
   ~Bug() { delete p;}
  };
  long & GetPb(Bug   a,int i){ return a.p[i];}                //    copie puis
                                                //    destruction de la copie
  long & GetOk(Bug & a,int i){ return a.p[i];}                       //    ok
```

```
int main(int argc,char ** argv) {
 bug a;
 GetPb(a,1) = 1;                    //    bug le pointeur a.p est détruit ici
                                      //    l'argument est copie puis détruit
 cout « GetOk(a,1) « "\n";       //    bug on utilise un zone mémoire libérée

 return 0;                       //    le pointeur a.p est encore détruit ici
}
```

Le pire est que ce programme marche sur la plupart des ordinateurs et donne le résultat jusqu'au jour où l'on ajoute du code entre les 2 get (2 ou 3 ans après), c'est terrible mais ça marchait !...

Règle 1.2 *Dans une fonction, ne jamais retournez de référence ou le pointeur sur une variable locale*

Effectivement, retourne du référence sur une variable local implique que l'on retourne l'adresse mémoire de la pile, qui n'est libéré automatique en sortie de fonction, qui est invalide hors de la fonction. mais bien sur le programme écrire peut marche avec de la chance.

Il ne faut jamais faire ceci :

```
int & Add(int i,int j)
{ int l=i+j;
  return l; }                          //    bug return d'une variable local l
```

Mais vous pouvez retourner une référence définie à partir des arguments, ou à partir de variables static ou global qui sont rémanentes.

Règle 1.3 *Si, dans un programme, vous savez qu'un expression logique doit être vraie, alors Vous devez mettre une assertion de cette expression logique.*

Ne pas penser au problème du temps calcul dans un premier temps, il est possible de retirer toutes les assertions en compilant avec l'option `-DNDEBUG`, ou en définissant la macro du preprocesseur `#define NDEBUG`, si vous voulez faire du filtrage avec des assertions, il suffit de définir les macros suivante dans un fichier `ftp://www.freefem.org/snc++/assertion.hpp` qui active les assertions

```
#ifndef ASSERTION_HPP_
#define ASSERTION_HPP_
//    to compile all assertion
//   #define ASSERTION
//    to remove all the assert
//   #define NDEBUG
#include <assert.h>
#define ASSERTION(i)  0
#ifdef ASSERTION
#undef ASSERTION
#define ASSERTION(i) assert(i)
#endif
                        //    4 small function to test if you are inside a segment
[a,b[,[a,b],]a,b],]a,b[
```

```
template<class T> inline bool  Inside_of(const T &  a,const T & i,const T & b)
 {return (a <= i) && ( i < b);}
template<class T> inline bool  Inside_oo(const T &  a,const T & i,const T & b)
 {return (a <= i) && ( i <= b);}
template<class T> inline bool  Inside_fo(const T &  a,const T & i,const T & b)
 {return (a > i) && ( i <= b);}
template<class T> inline bool  Inside_ff(const T &  a,const T & i,const T & b)
 {return (a > i)  && ( i < b);}
#endif
```

comme cela il est possible de garder un niveau d'assertion avec `assert`. Pour des cas plus fondamentaux et qui sont négligeables en temps calcul. Il suffit de définir la macro `ASSERTION` pour que les testes soient effectués sinon le code n'est pas compilé et est remplacé par `0`.

Il est fondamental de vérifier les bornes de tableaux, ainsi que les autres bornes connues. Aujourd'hui je viens de trouver une erreur stupide, un déplacement de tableau dû à l'échange de 2 indices dans un tableau qui ralentissait très sensiblement mon logiciel (je n'avais respecté cette règle).

Exemple d'une petite classe qui modélise un tableau d'entier

```
  class Itab{ public:
    int n;
    int *p;
    Itab(int nn)
     { n=nn);
       p=new int[n];
       assert(p);}                                      //    vérification du pointeur
    ~Itab()
     { assert(p);                                       //    vérification du pointeur
       delete p;
       p=0;}                                  //    pour éviter les doubles destruction
    int & operator[](int i) { assert( i >=0 && i < n && p ); return p[i];}

  private:     //    la règle 1 : pas de copie par défaut il y a un destructeur
    Itab(const Itab &);                            //    pas de constructeur par copie
    void operator=(const Itab &);                      //    pas d'affection par copie
  }
```

Règle 1.4 *N'utilisez le macro générateur que si vous ne pouvez pas faire autrement, ou pour ajoutez du code de vérification ou test qui sera très utile lors de la mise au point.*

Règle 1.5 *Une fois toutes les erreurs de compilation et d'édition des liens corrigées, il faut éditer les liens en ajoutant `CheckPtr.o` (le purify du pauvre) à la liste des objets à éditer les liens, afin de faire les vérifications des allocations.*

Corriger tous les erreurs de pointeurs bien sûr, et les erreurs assertions avec le débogueur.

1.4 Vérificateur d'allocation

L'idée est très simple, il suffit de surcharger les opérateurs **`new`** et **`delete`**, de stocker en mémoire tous les pointeurs alloués et de vérifier avant chaque déallacation s'il fut bien alloué (cf. `AllocExtern::MyNewOperator(size_t )` et `AllocExternData.MyDeleteOperator(`**`void`**` *)`. Le tout est d'encapsuler dans une classe `AllocExtern` pour qu'il n'y est pas de conflit de nom.
De plus, on utilise `malloc` et `free` du C, pour éviter des problèmes de récurrence infinie dans l'allocateur. Pour chaque allocation, avant et après le tableau, deux petites zones mémoire de 8 octets sont utilisées pour retrouver des débordement amont et aval.
Et le tout est initialisé et terminé sans modification du code source en utilisant la variable `AllocExternData` globale qui est construite puis détruite. À la destruction la liste des pointeurs non détruits est écrite dans un fichier, qui est relue à la construction, ce qui permet de déboguer les oublis de déallocation de pointeurs.

Remarque 1.1 *Ce code marche bien si l'on ne fait pas trop d'allocations, destructions dans le programme, car le nombre d'opérations pour vérifier la destruction d'un pointeur est en nombre de pointeurs alloués. L'algorithme est donc proportionnel au carré du nombre de pointeurs alloués par le programme. Il est possible d'améliorer l'algorithme en triant les pointeurs par adresse et en faisant une recherche dichotomique pour la destruction.*

Le source de ce vérificateur `CheckPtr.cpp` est disponible à l'adresse suivante `ftp://www.freefem.org/snc++/CheckPtr.cpp`. Pour l'utiliser, il suffit de compiler et d'éditer les liens avec les autres parties du programme.
Il est aussi possible de retrouver les pointeurs non désalloués, en utilisant votre débogueur favori (par exemple gdb).
Dans `CheckPtr.cpp`, il y a une macro du préprocesseur `DEBUGUNALLOC` qu'il faut définir, et qui active l'appel de la fonction `debugunalloc()` à la création de chaque pointeur non détruit. La liste des pointeurs non détruits est stockée dans le fichier `ListOfUnAllocPtr.bin`, et ce fichier est génèré par l'exécution de votre programme.
Donc pour déboguer votre programme, il suffit de faire :

1. Compilez `CheckPtr.cpp`.
2. Editez les liens de votre programme `C++` avec `CheckPtr.o`.
3. Exécutez votre programme avec un jeu de donné.
4. Réexécutez votre programme sous le debogueur et mettez un point d'arrêt dans la fonction `debugunalloc()` (deuxième ligne `CheckPtr.cpp` .
5. remontez dans la pile des fonctions appelées pour voir quel pointeur n'est pas désalloué.
6. etc...

Exercice 1.2 un Makefile pour compiler et tester tous les programmes du `ftp://www.freefem.org/snc++/l1/exemple_de_base` avec `CheckPtr`

1.5 Compréhension des constructeurs, destructeurs et des passages d'arguments

Faire une classe T avec un constructeur par copie et le destructeur, et la copie par affectation : qui imprime quelque chose comme par exemple :

```
class T { public:
  T() { cout «  "Constructeur par défaut " « this « "\n"}
  T(const & T a) { cout «"Constructeur par copie "
                          « this « "\n"}
  ~T() { cout «  "destructeur "« this « "\n"}
  T & operator=(T & a) {cout « " copie par affectation :"
                          « this « " = " « &a « endl;}
  };
```

Puis tester, cette classe faisant un programme qui appelle les 4 fonctions suivantes, et qui contient une variable globale de type T.

```
 T f1(T a){ return a;}
 T f2(T &a){ return a;}
 T &f3(T a){ return a;}          //    il y a un bug, le quel?
 T &f4(T &a){ return a;}
```

Analysé et discuté très finement les résultat obtenus, et comprendre pourquoi, la classe suivant ne fonctionne pas, ou les opérateur programme font la même chose que les opérateurs par défaut.

Exercice 1.3

```
class T { public:
  int * p;                                //    un pointeur
  T() {p=new int;
       cout «  "Constructeur par defaut " « this
            « " p=" « p « "\n"}
  T(const & T a) {
       p=a.p;
       cout «  "Constructeur par copie "« this
            « " p=" « p « "\n"}
  ~T() {
       cout «  "destructeur "« this
            « " p=" « p « "\n";
       delete p;}
  T & operator=(T & a) {
     cout « "copie par affectation  "« this
          « "old  p=" « p
          «  "new p="« a.p « "\n";
     p=a.p; }
  };
```

remarque : bien sur vous pouvez et même vous devez tester ses deux classes, avec le vérificateur d'allocation dynamique.

Chapitre 2

Exemples de Classe C++

2.1 Le Plan $\mathbb{R}^2$

Voici une modélisation de $\mathbb{R}^2$ disponible à `ftp://www.freefem.org/snc++/R2/R2.hpp` qui permet de faire des opérations vectorielles et qui définit le produit scalaire de deux points A et B, le produit scalaire sera défini par (A, B).

2.1.1 La classe R2

```
//    Définition de la class R2
//    sans compilation sépare toute les fonctions
//    sous défini dans ce R2.hpp avec des inline
                                                                        //
//    remarque la fonction abort est déclaré dans #include <cstdlib>

//    definition R (les nombres reals)
typedef double R;                              //    définition de R (les réel)

//    The class R2
class R2 {
  //    utilisation de friend pour definir de fonction exterieur a
  //    la class dans la class
  friend std::ostream& operator «(std::ostream& f, const R2 & P )
       { f « P.x « ’ ’ « P.y  ; return f; }
  friend std::istream& operator »(std::istream& f,  R2 & P)
       { f »  P.x »  P.y ; return f; }

public:
  R x,y;                                          //    declaration des membre
                                                  //    les 3 constructeurs --
  R2 () :x(0.),y(0.) {}              //    rappel : x(0), y(0) sont initialiser
   R2 (R a,R b):x(a),y(b)  {}              //    via le constructeur de double
  R2 (const R2 & a,const R2 & b):x(b.x-a.x),y(b.y-a.y)  {}

  //    le constucteur par defaut est inutile
  R2 (const R2 & a) :x(a.x),y(a.y) {}

  //    rappel les operator definis dans une class on un parametre
  //    caché qui est la class elle meme (*this)
```

```
  //    les opérateurs affectations
  //    operateur affection = est inutil car par défaut,il est correct
  R2 &  operator=(const R2 & P)  {x = P.x;y = P.y;return *this;}
                                      //    les autre opérateur d'affectations
  R2 &  operator+=(const R2 & P)  {x += P.x;y += P.y;return *this;}
  R2 &  operator-=(const R2 & P) {x -= P.x;y -= P.y;return *this;}
  //    les operateur binaire + - * , ^
  R2   operator+(const R2 & P)const  {return R2(x+P.x,y+P.y);}
  R2   operator-(const R2 & P)const {return R2(x-P.x,y-P.y);}
  R    operator,(const R2 & P)const {return  x*P.x+y*P.y;}       //    produit
scalaire
  R    operator^(const R2 & P)const {return  x*P.y-y*P.x;} //    produit mixte
  R2   operator*(R c)const {return R2(x*c,y*c);}
  R2   operator/(R c)const {return R2(x/c,y/c);}
                                                          //    operateur unaire
  R2   operator-()const  {return R2(-x,-y);}
  R2   operator+()const  {return *this;}
                                                                //    un methode
  R2   perp() const {return R2(-y,x);}                 //    la perpendiculaire

  //    les operators tableau
  //    version qui peut modifie la class via l'adresse de x ou y
  R & operator[](int i) { if(i==0) return x;             //    donc pas const
                          else if (i==1) return y;
                          else {assert(0);exit(1);};}
  //    version qui retourne une reference const qui ne modifie pas la class
  const R & operator[](int i) const { if(i==0) return x;     //    donc  const
                                      else if (i==1) return y;
                                      else {assert(0);exit(1);};}

  //    les operateurs fonction
             //    version qui peut modifie la class via l'adresse de x ou y
  R & operator()(int i) { if(i==1) return x;             //    donc pas const
                          else if (i==2) return y;
                          else {assert(0);abort();};}
    //    version qui retourne une reference const qui ne modifie pas la class
  const R & operator()(int i) const { if(i==1) return x;     //    donc  const
                                      else if (i==2) return y;
                                      else {assert(0);abort();};}

};                                                     //    fin de la class R2


inline R2 operator*(R c,const R2 & P) {return P*c;}
inline R2 perp(const R2 & P) { return R2(-P.y,P.x); }
inline R2 Perp(const R2 & P) { return P.perp(); }   //    autre ecriture de la
fonction perp
```

Quelques remarques sur la syntaxe :
- Les opérateurs binaires dans une classe n'ont seulement qu'un paramètre. Le premier paramètre étant la classe et le second étant le paramètre fourni ;
- si un opérateur ou une fonction membre d'une classe ne modifie pas la classe alors il est conseillé de dire au compilateur que cette fonction est « constante »en ajoutant le mots clef **`const`** après la définition des paramètres ;

– dans le cas d'un opérateur défini hors d'une classe le nombre de paramètres est donné par le type de l'opérateur uniare (`+ - * ! [] etc..` : 1 paramètre), binaire (`+ - * / | & || &&^ == <= >= < > etc..` 2 paramètres), n-aire (`()` : n paramètres).
– `ostream`, `istream` sont les deux types standards pour respectivement écrire et lire dans un fichier ou sur les entrées sorties standard. Ces types sont définis dans le fichier « iostream »inclue avec l'ordre `#include<iostream>` qui est mis en tête de fichier ;
– les deux opérateurs « et » sont les deux opérateurs qui généralement et respectivement écrivent ou lisent dans un type `ostream`, `istream`, ou `iostream`.

2.1.2 Utilisation de la classe R2

Cette classe modélise le plan $\mathbb{R}^2$, pour que les opérateurs classiques fonctionnent, c'est-à-dire :

Un point P du plan définit via modélisé par ces 2 coordonnées x, y, et nous pouvons écrire des lignes suivantes par exemple :

```
R2 P(1.0,-0.5),Q(0,10);
R2 O,PQ(P,Q);                    //  le point O est initialiser à 0,0
R2 M=(P+Q)/2;                    //  espace vectoriel à droite
R2 A = 0.5*(P+Q);                //  espace vectoriel à gauche
R  ps = (A,M);                   //  le produit scalaire de R2
R  pm = A^M;                     //  le deteminant de A,M
                     //  l'aire du parallélogramme formé par A,M
R2 B = A.perp();                 //  B est la rotation de A par π/2
R  a= A.x + B.y;
A  = -B;
A += M;                          //  ie. A = A + M;
A -= B;                          //  ie. A = -A;
double abscisse= A.x;            //  la composante x de A
double ordonne = A.y;            //  la composante y de A
A.y= 0.5;                        //  change la composante de A
A(1) =5;                         //  change la 1 composante (x) de A
A(2) =2;                         //  change la 2 composante (y) de A
A[0] =10;                        //  change la 1 composante (x) de A
A[1] =100;                       //  change la 2 composante (y) de A
cout «  A;                       //  imprime sur la console A.x et A.x
cint » A;                  //  vous devez entrer 2 double à la console
```

Exercice 2.1

Le but de cette exercice de d'affiche graphiquement les bassins d'attraction de la méthode de Newton, pour la résolution du problème $z^p - 1 = 0$, où $p = 3, 4, ...$

Rappel : la méthode de Newton s'écrit :

$$\text{Soit } z_0 \in \mathbb{C}, \quad z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)};$$

Q 1 Faire une classe `C` qui modélise le corps $\mathbb{C}$ des nombres complexes ($z = a + i * b$), avec toutes les opérations algébriques.

Q 2 Ecrire une fonction `C Newton(C z0,C (*f)(C),C (*df)(C) )` pour qui retourne la racines de l'équation $f(z) = 0$, limite des itérations de Newton : $z_{n+1} = z_n - \frac{f(z_n)}{df(z_n)}$ avec par exemple $f(z) = z^p - 1$ et $df(z) = f'(z) = p * z^{p-1}$, partant de $z0$. Cette fonction retournera le nombre complexe nul en cas de non convergence.

Q 3 Faire un programme, qui appelle la fonction Newton pour les points de la grille $z_0 + dn + dmi$ pour $(n, m) \in \{0, \ldots, N-1\} \times \{0, \ldots, M-1\}$ où les variables z_0, d, N, M sont données par l'utilisateur.

Afficher un tableau $N \times M$ résultat, qui nous dit vers quel racine la méthode de Newton a convergé en chaque point de la grille.

Q4 modifier les sources `ftp://www.freefem.org/snc++/l1/pj0.tar.gz` afin d'affiché graphique, il suffit de modifier la fonction `void Draw()`, de plus les bornes de l'écran sont entières et sont stockées dans les variables global `int Width , Height;`.

Sous linux pour compiler et éditer de lien, il faut entrer les commandes `shell` suivantes :

```
g++ ex1.cpp -L/usr/X11R6/lib  -lGL -lGLUT -lX11 -o ex1
 # pour afficher le dessin dans une fenêtre X11.
```

ou mieux utiliser la commande unix `make` ou `gmake` en créant le fichier `Makefile` contenant :

```
CPP=g++
CPPFLAGS= -I/usr/X11R6/include
LIB= -L/usr/X11R6/lib -lglut -lGLU -lGL -lX11 -lm
%.o:%.cpp
(caractere de tabulation  -->/)$(CPP) -c $(CPPFLAGS) $^
cercle: cercle.o
(caractere de tabulation  -->/)g++ ex1.o  $(LIB) -o ex1
```

2.2 Les classes tableaux

Nous commencerons sur une version didactique, nous verrons la version complète qui est dans le fichier « tar compressé » `ftp://www.freefem.org/snc++/RNM.tar.gz`.

2.2.1 Version simple d'une classe tableau

Mais avant toute chose, me paraît clair qu'un vecteur sera un classe qui contient au minimum la taille n, et un pointeur sur les valeurs. Que faut'il dans cette classe minimale (noté A).

```
typedef double K;                                    //    définition du corps
class A { public:                                        //    version 1 -----

  int n;                                             //    la taille du vecteur
  K *v;                                        //    pointeur sur les n valeurs
  A() { cerr «" Pas de constructeur pas défaut " « endl; exit(1);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                //    constructeur
  ~A() {delete [] v;}                                       //    destructeur
   K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
};
```

Cette classe ne fonctionne pas car le constructeur par copie par défaut fait une copie bit à bit et donc le pointeur v est perdu, il faut donc écrire :

```
class A { public:                                        //    version 2 -----
  int n;                                             //    la taille du vecteur
  K *v;                                        //    pointeur sur les n valeurs
  A() { cerr «" Pas de constructeur pas défaut " « endl; exit(1);}
  A(const A& a) :n(a.n),v(new K[a.n])           //    constructeur par copie
       { operator=(a);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                //    constructeur
  A& operator=(A &a) {                                             //    copie
        assert(n==a.n);
        for(int i=0;i<n;i++) v[i]=a.v[i];
        return *this;}
  ~A() {delete [] v;}                                       //    destructeur
  K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
};
```

Maintenant nous voulons ajouter les opérations vectorielles $+, -, *, \ldots$

```
class A { public:                                        //    version 3 -----
  int n;                                             //    la taille du vecteur
  K *v;                                        //    pointeur sur les n valeurs
  A() { cerr «" Pas de constructeur pas défaut " « endl; exit(1);}
  A(const A& a) :n(a.n),v(new K[a.n]) { operator=(a);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                //    constructeur
  A& operator=(A &a) {assert(n==a.n);
                      for(int i=0;i<n;i++) v[i]=a.v[i];
                      return *this}
  ~A() {delete [] v;}                                       //    destructeur
  K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
  A operator+(const &a) const;                                  //    addition
  A operator*(K &a) const;                     //    espace vectoriel à droite

 private:                //    constructeur privé pour faire des optimisations
  A(int i, K* p) : n(i),v(p){assert(v);}
  friend A operator*(const K& a,const A& );    //    multiplication à gauche

};
```

Il faut faire attention dans les paramètres d'entrée et de sortie des opérateurs. Il est clair que l'on ne veut pas travailler sur des copies, mais la sortie est forcément un objet et non une référence sur un objet car il faut allouer de la mémoire dans l'opérateur et si l'on retourne une référence aucun destructeur ne sera appelé et donc cette mémoire ne sera jamais libérée.

```
              //   version avec avec une copie du tableau au niveau du return
A A::operator+(const A & a) const {
 A b(n); assert(n == a.n);
 for (int i=0;i<n;i++) b.v[i]= v[i]+a.v[i];
 return b;                  //    ici l'opérateur A(const A& a) est appeler
}
```

Pour des raisons optimisation nous ajoutons un nouveau constructeur `A(int,K*)` qui évitera de faire une copie du tableau.

```
                                             //    -- version optimisée sans copie---
A A::operator+(const A & a) const {
 K  *b(new K[n]); assert(n == a.n);
 for (int i=0;i<n;i++) b[i]= v[i]+a.[i];
 return A(n,b);            //    ici l'opérateur A(n,K*) ne fait pas de copie
}
```

Pour la multiplication par un scalaire à droite on a :

```
                                                       //    -- version optimisée ---
A A::operator*(const K & a) const {
 K  *b(new K[n]); assert(n == a.n);
 for (int i=0;i<n;i++) b[i]= v[i]*a;
 return A(n,b);            //    ici l'opérateur A(n,K*) ne fait pas de copie
}
```

Pour la version à gauche, il faut définir `operator*` extérieurement à la classe car le terme de gauche n'est pas un vecteur.

```
A operator*(const K & a,const T & c) {
 K  *b(new K[n]); assert(n == a.n);
 for (int i=0;i<n;i++) b[i]= c[i]*a;
 return A(n,b);                //    attention c'est opérateur est privé donc
                     //    cette fonction doit est ami ( friend) de la classe
}
```

Maintenant regardons ce qui est exécuté dans le cas d'une expression vectorielle.

```
 int n=100000;
 A a(n),b(n),c(n),d(n);
 ...                                    //    initialisation des tableaux a,b,c
 d = (a+2.*b)+c*2.0;
```

voilà le pseudo code généré avec les 3 fonctions suivantes : `add(a,b,ab)`, `mulg(s,b,sb)`, `muld(a,s,as)`, `copy(a,b)` où le dernier argument retourne le résultat.

```
 A a(n),b(n),c(n),d(n);
 A t1(n),t2(n),t3(n),t4(n);
 muld(2.,b),t1);                                           //    t1 = 2.*b
 add(a,t1,t2);                                           //    t2 = a+2.*b
```

```
  mulg(c,2.,t3);                                          //     t3 = c*2.
  add(t2,t3,t4);                                     //     t4 = (a+2.*b)+c*2.0;
  copy(t4,d);                                         //     d = (a+2.*b)+c*2.0;
```

Nous voyons que quatre tableaux intermédiaires sont créés ce qui est excessif. D'où l'idée de ne pas utiliser toutes ces possibilités car le code généré sera trop lent.

Remarque 2.1 *Il est toujours possible de créer des classes intermédiaires pour des opérations prédéfinies afin obtenir se code générer :*

```
  A a(n),b(n),c(n),d(n);
  for (int i;i<n;i++)
    d[i] = (a[i]+2.*b[i])+c[i]*2.0;
```

Ce cas me paraît trop compliquer, mais nous pouvons optimiser raisonnablement toutes les combinaisons linéaires à 2 termes.

Mais, il est toujours possible d'éviter ces problèmes en utilisant les opérateurs `+=, -=, *=, /=` ce que donnerai de ce cas

```
A a(n),b(n),c(n),d(n);
for (int i;i<n;i++)
   d[i] = (a[i]+2.*b[i])+c[i]*2.0;
```

D'où l'idée de découper les classes vecteurs en deux types de classe les classes de terminer par un _ sont des classes sans allocation (cf. new), et les autres font appel à l'allocateur `new` et au déallocateur `delete`. De plus comme en fortran 90, il est souvent utile de voir une matrice comme un vecteur ou d'extraire une ligne ou une colonne ou même une sous-matrice. Pour pouvoir faire tous cela comme en fortran 90, nous allons considère un tableau comme un nombre d'élément n, un incrément s et un pointeur v et du tableau suivant de même type afin de extraire des sous-tableau.

2.2.2 les classes RNM

Nous définissons des classes tableaux à un, deux ou trois indices avec ou sans allocation. Ces tableaux est défini par un pointeur sur les valeurs et par la forme de l'indice (class `ShapeOfArray`) qui donne la taille, le pas entre de valeur et le tableau suivant de même type (ces deux données supplémentaire permettent extraire des colonnes ou des lignes de matrice, ...).
La version est dans le fichier « tar compressé » `ftp://www.freefem.org/snc++/RNM.tar.gz`.
Nous voulons faire les opérations classiques sur `A, C,D` tableaux de type `KN<R>` suivante par exemples :

```
  A = B; A += B; A -= B; A = 1.0; A = 2*.C; A /=C; A *=C;
  A = A+B; A = 2.0*C+ B; C = 3.*A-5*B;
  R c = A[i];
  A[j] = c;
```

Pour des raisons évidentes nous ne sommes pas allés plus loin que des combinaisons linéaires à plus de deux termes. Toutes ces opérations sont faites sans aucune allocation et avec une seule boucle.

De plus nous avons défini, les tableaux 1,2 ou 3 indices, il est possible extraire une partie d'un tableau, une ligne ou une colonne.

```
#include<RNM.hpp>

 ....

  typedef double R;
  KNM<R> A(10,20);                                                  //    un matrice
  . . .
  KN_<R> L1(A(1,'.'));                                  //    la ligne 1 de la matrice A;
  KN<R> cL1(A(1,'.'));                        //    copie de la ligne 1 de la matrice A;
  KN_<R> C2(A('.',2));                                //    la colonne 2 de la matrice A;
  KN<R> cC2(A('.',2));                      //    copie de la colonne 2 de la matrice A;
  KNM_<R> pA(FromTo(2,5),FromTo(3,7));   //    partie de la matrice A(2:5,3:7)
                                                             //    vue comme un matrice 4x5

  KNM B(n,n);
  B(SubArray(n,0,n+1))                                    //    le vecteur diagonal de B;
  KNM_ Bt(B.t());                                   //    la matrice transpose sans copie
```

Pour l'utilisation, utiliser l'ordre `#include "RNM.hpp"`, et les flags de compilation `-DCHECK_KN` ou en définisant la variable du preprocesseur `cpp` du `C++` avec l'ordre `#defined CHECK_KN`, avant la ligne include.
Les définitions des classes sont faites dans 4 fichiers `RNM.hpp`, `RNM_tpl.hpp`, `RNM_op.hpp`, `RNM_op.hpp`.
Pour plus de détails voici un exemple d'utilisation assez complet.

2.2.3 Exemple d'utilisation de classes RNM

```
namespace std

#define CHECK_KN
#include "RNM.hpp"
#include "assert.h"

using namespace std;
       //    definition des 6 types de base des tableaux a 1,2 et 3 parametres
typedef double R;
typedef KN<R> Rn;
typedef KN_<R> Rn_;
typedef KNM<R> Rnm;
typedef KNM_<R> Rnm_;
typedef KNMK<R> Rnmk;
typedef KNMK_<R> Rnmk_;
R f(int i){return i;}
R g(int i){return -i;}
int main()
{
  const int n= 8;
  cout « "Hello World, this is RNM use!" « endl « endl;
  Rn a(n,f),b(n),c(n);
```

```
b =a;
c=5;
b *= c;

cout « " a = " « (KN_<const_R>) a « endl;
cout « " b = " «  b « endl;

                                          //    les operations vectorielles
c = a + b;
c = 5. *b + a;
c = a + 5. *b;
c = a - 5. *b;
c = 10.*a - 5. *b;
c = 10.*a + 5. *b;
c += a + b;
c += 5. *b + a;
c += a + 5. *b;
c += a - 5. *b;
c += 10.*a - 5. *b;
c += 10.*a + 5. *b;
c -= a + b;
c -= 5. *b + a;
c -= a + 5. *b;
c -= a - 5. *b;
c -= 10.*a - 5. *b;
c -= 10.*a + 5. *b;


cout «" c = " « c « endl;
Rn u(20,f),v(20,g);                                //    2 tableaux u,v de 20
                                  //   initialiser avec u_i = f(i), v_j = g(i)
Rnm A(n+2,n);                                      //    Une matrice n+2 x n

for (int i=0;i<A.N();i++)                                        //    ligne
   for (int j=0;j<A.M();j++)                                     //    colonne
     A(i,j) = 10*i+j;

cout « "A=" « A « endl;
cout « "Ai3=A('.', 3 ) = " « A('.', 3 ) « endl;          //    la colonne 3
cout « "A1j=A( 1 ,'.') = " « A( 1 ,'.') « endl;           //    la ligne 1
Rn CopyAi3(A('.', 3 ));                       //    une copie de la colonne 3
cout « "CopyAi3 = " « CopyAi3;

Rn_ Ai3(A('.', 3 ));                          //    la colonne 3 de la matrice
CopyAi3[3]=100;
cout « CopyAi3 « endl;
cout « Ai3 « endl;

assert( & A(0,3) == & Ai3(0));                //    verification des adresses


Rnm S(A(SubArray(3),SubArray(3)));                      //    sous matrice 3x3

Rn_ Sii(S,SubArray(3,0,3+1));       //    la diagonal de la matrice sans copy

cout « "S=  A(SubArray(3),SubArray(3)  = " « S «endl;
```

```
  cout « "Sii     = " « Sii «endl;
  b = 1;
                                                              //    Rn Ab(n+2) = A*b; error
  Rn Ab(n+2);
  Ab = A*b;
  cout « " Ab = A*b  =" « Ab « endl;

  Rn_ u10(u,SubArray(10,5));                 //    la partie [5,5+10[ du tableau u
  cout « "u10 " « u10 « endl;
  v(SubArray(10,5)) += u10;
  cout « " v = " « v « endl;
  cout « " u(SubArray(10))    "  « u(SubArray(10)) « endl;
  cout « " u(SubArray(10,5))  "  « u(SubArray(10,5)) « endl;
  cout « " u(SubArray(8,5,2))"  « u(SubArray(8,5,2))
       « endl;


  cout « " A(5,'.')[1] " « A(5,'.')[1] « " " « " A(5,1) = "
       « A(5,1) « endl;
  cout « " A('.',5)(1) = "« A('.',5)(1) «  endl;
  cout « " A(SubArray(3,2),SubArray(2,4)) = " « endl;
  cout «  A(SubArray(3,2),SubArray(2,4)) « endl;
  A(SubArray(3,2),SubArray(2,4)) = -1;
  A(SubArray(3,2),SubArray(2,0)) = -2;
  cout « A « endl;

  Rnmk B(3,4,5);
  for (int i=0;i<B.N();i++)                                              //    ligne
     for (int j=0;j<B.M();j++)                                          //    colonne
        for (int k=0;k<B.K();k++)                                        //    ....
           B(i,j,k) = 100*i+10*j+k;
  cout « " B      = " « B « endl;
  cout « " B(1  ,2  ,'.') " « B(1  ,2  ,'.') « endl;
  cout « " B(1  ,'.',3  ) " « B(1  ,'.',3  ) « endl;
  cout « " B('.',2  ,3  ) " « B('.',2  ,3  ) « endl;
  cout « " B(1  ,'.','.') " « B(1,'.'  ,'.') « endl;
  cout « " B('.',2  ,'.') " « B('.',2  ,'.') « endl;
  cout « " B('.','.',3  ) " « B('.','.',3  ) « endl;

  cout « " B(1:2,1:3,0:3)   = "
       « B(SubArray(2,1),SubArray(3,1),SubArray(4,0)) « endl;

                                                               //    copie du sous tableaux

  Rnmk Bsub(B(FromTo(1,2),FromTo(1,3),FromTo(0,3)));
  B(SubArray(2,1),SubArray(3,1),SubArray(4,0))  = -1;
  B(SubArray(2,1),SubArray(3,1),SubArray(4,0)) += -1;
  cout « " B      = " « B « endl;
  cout « Bsub « endl;

  return 0;
}
```

2.2.4 Une resolution de systéme linéaire avec le gradient conjugué

L'algorithme du gradient conjugué présenté dans cette section est utilisé pour résoudre le système linéaire $Ax = b$, où A est une matrice symétrique positive
$n \times n$.
Cet algorithme est basé sur la minimisation de la fonctionnelle quadratique $E : \mathbb{R}^n \to \mathbb{R}$ suivante :

$$E(x) = \frac{1}{2}(Ax, x)_C - (b, x)_C,$$

où $(.,.)_C$ est le produit scalaire associer à une une matrice C, symétrique définie positive de $\mathbb{R}^n$.

Algorithme 2.1

Le gradient conjugué preconditionné soient $x^0 \in \mathbb{R}^n$, ε, C donnés
$G^0 = Ax^0 - b$
$H^0 = -CG^0$
– *pour* $i = 0$ *à* n
$\rho = -\frac{(G^i, H^i)}{(H^i, AH^i)}$
$x^{i+1} = x^i + \rho H^i$
$G^{i+1} = G^i + \rho AH^i$
$\gamma = \frac{(G^{i+1}, G^{i+1})_C}{(G^i, G^i)_C}$
$H^{i+1} = -CG^{i+1} + \gamma H^i$
si $(G^{i+1}, G^{i+1})_C < \varepsilon$ *stop*

Voila comment écrire un gradient conjugué avec ces classes.

Gradient conjugué préconditionné

Listing 2.1 *(GC.hpp)*

```
            //    exemple de programmation du gradient conjugue precondiconnée
template<class R,class M,class P>
int GradienConjugue(const M & A,const P & C,const KN_<R> &b,KN_<R> &x,
                    int nbitermax,double eps)
{
   int n=b.N();
   assert(n==x.N());
   KN<R> g(n), h(n), Ah(n), & Cg(Ah);       //    on utilise Ah pour stocke Cg
   g = A*x;
   g -= b;                                                    //    g = Ax-b
   Cg = C*g;                                       //    gradient preconditionne
   h =-Cg;
   R g2 = (Cg,g);
   R reps2 = eps*eps*g2;                                   //    epsilon relatif
   for (int iter=0;iter<=nbitermax;iter++)
     {
       Ah = A*h;
       R ro =  - (g,h)/ (h,Ah);       //    ro optimal (produit scalaire usuel)
       x += ro *h;
       g += ro *Ah;   //    plus besoin de Ah, on utilise avec Cg optimisation
       Cg = C*g;
```

```
       R g2p=g2;
       g2 = (Cg,g);
       cout « iter «  "  ro = " « ro « " ||g||^2 = " « g2 « endl;
       if (g2 < reps2) {
          cout « iter «  "  ro = " « ro « " ||g||^2 = " « g2 « endl;
          return 1;                                              //    ok
          }
       R gamma = g2/g2p;
       h *= gamma;
       h -= Cg;                                    //    h = -Cg * gamma* h
     }
   cout « " Non convergence de la méthode du gradient conjugue " «endl;
   return 0;
}
//    la matrice Identite -------------
template <class R>
class MatriceIdentite: VirtualMatrice<R> { public:
 typedef  VirtualMatrice<R>::plusAx plusAx;
 MatriceIdentite() {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {   Ax+=x; }
 plusAx operator*(const KN<R> &  x) const {return plusAx(this,x);}
};
```

Test du gradient conjugué Pour finir voilà, un petit programme pour le testé sur cas différent. Le troisième cas étant la résolution de l'équation au différencielle 1d $-u'' = 1$ sur $[0,1]$ avec comme conditions aux limites $u(0) = u(1) = 0$, par la méthode de l'élément fini. La solution exact est $f(x) = x(1-x)/2$, nous verifions donc l'erreur sur le resultat.

Listing 2.2 *(GradConjugue.cpp)*

```
#include <fstream>
#include <cassert>
#include <algorithm>

using namespace std;

#define KN_CHECK
#include "RNM.hpp"
#include "GC.hpp"

typedef double R;
class MatriceLaplacien1D: VirtualMatrice<R> { public:
 MatriceLaplacien1D() {};
 void addMatMul(const KN_<R> & x, KN_<R> & Ax) const;
 plusAx operator*(const KN<R> &  x) const {return plusAx(*this,x);}
};

 void MatriceLaplacien1D::addMatMul(const KN_<R> & x, KN_<R> & Ax) const {
  int n= x.N(),n_1=n-1;
  double h=1./(n_1), h2= h*h, d = 2/h, d1 = -1/h;
  R Ax0=Ax[0], Axn_1=Ax[n_1];
  Ax=0;
  for (int i=1;i< n_1; i++)
```

```
    Ax[i] = (x[i-1] +x[i+1]) * d1 + x[i]*d ;
                                                                             //    CL
   Ax[0]=x[0];
   Ax[n_1]=x[n_1];
  }


int main(int argc,char ** argv)
{
 typedef KN<double> Rn;
 typedef KN_<double> Rn_;
 typedef KNM<double> Rnm;
 typedef KNM_<double> Rnm_;
 {
  int n=10;
  Rnm A(n,n),C(n,n),Id(n,n);
  A=-1;
  C=0;
  Id=0;
  Rn_ Aii(A,SubArray(n,0,n+1));      //   la diagonal de la matrice A sans copy
  Rn_ Cii(C,SubArray(n,0,n+1));      //   la diagonal de la matrice C sans copy
  Rn_ Idii(Id,SubArray(n,0,n+1));  //   la diagonal de la matrice Id sans copy
  for (int i=0;i<n;i++)
     Cii[i]= 1/(Aii[i]=n+i*i*i);
  Idii=1;
  cout « A;
  Rn x(n),b(n),s(n);
  for (int i=0;i<n;i++) b[i]=i;
  cout « "GradienConjugue preconditionne par la diagonale " «  endl;
  x=0;
  GradienConjugue(A,C,b,x,n,1e-10);
  s = A*x;
  cout « " solution : A*x= "  « s « endl;
  cout « "GradienConjugue preconditionnee par la identity  " « endl;
  x=0;
  GradienConjugue(A,MatriceIdentite<R>(),b,x,n,1e-6);
  s = A*x;
  cout « s « endl;
 }
 {
  cout « "GradienConjugue laplacien 1D  par la identity  " « endl;
  int N=100;
  Rn b(N),x(N);
  R h= 1./(N-1);
  b= h;
  b[0]=0;
  b[N-1]=0;
  x=0;
  R t0=CPUtime();
  GradienConjugue(MatriceLaplacien1D(),MatriceIdentite<R>() ,b,x,N,1e-5);
  cout « " Temps cpu = " « CPUtime() - t0« "s" « endl;
  R err=0;
  for (int i=0;i<N;i++)
    {
    R xx=i*h;
```

```
    err= max(fabs(x[i]- (xx*(1-xx)/2)),err);
    }
  cout « "Fin err=" « err « endl;
 }
 return 0;
}
```

Sortie du test `10x10 :`

```
          10  -1  -1  -1  -1  -1  -1  -1  -1  -1
          -1  11  -1  -1  -1  -1  -1  -1  -1  -1
          -1  -1  18  -1  -1  -1  -1  -1  -1  -1
          -1  -1  -1  37  -1  -1  -1  -1  -1  -1
          -1  -1  -1  -1  74  -1  -1  -1  -1  -1
          -1  -1  -1  -1  -1 135  -1  -1  -1  -1
          -1  -1  -1  -1  -1  -1 226  -1  -1  -1
          -1  -1  -1  -1  -1  -1  -1 353  -1  -1
          -1  -1  -1  -1  -1  -1  -1  -1 522  -1
          -1  -1  -1  -1  -1  -1  -1  -1  -1 739
        GradienConjugue preconditionne par la diagonale
6  ro = 0.990712 ||g||^2 = 1.4253e-24
 solution : A*x= 10     :    1.60635e-15  1 2 3 4 5 6 7 8 9

GradienConjugue preconditionnee par la identity
9  ro = 0.0889083 ||g||^2 = 2.28121e-15
10      :       6.50655e-11    1 2 3 4 5 6 7 8 9

GradienConjugue laplacien 1D  preconditionnee par la identity
48  ro = 0.00505051 ||g||^2 = 1.55006e-32
 Temps cpu = 0.02s
Fin err=5.55112e-17
```

Exercice 2.2

Modifier, l'exemple `GradConjugue.cpp`, pour résoudre le problème suivant, trouver $u(x,t)$ une solution

$$\frac{\partial u}{\partial t} - \Delta u = f; \quad \text{d ans }]0,L[$$

$$\text{pour } t=0, u(x,0)=u_O(x) \quad \text{et } u(0,t)=u(L,t)=0$$

en utilisant un θ schéma pour discrétiser en temps, c'est à dire que

$$\frac{u^{n+1}-u^n}{\Delta t} - \Delta\big(\theta u^{n+1} + (1-\theta)u^n\big) = f; \quad \text{dans }]0,L[$$

où u^n est une approximation de $u(x,n\Delta t)$, faite avec des éléménts finis P_1, avec un maillage régulier de $]0,L|$ en M éléments. les fonctions élémentaires seront noté w^i, avec pour $i=0,...,M$, avec $x_i = \frac{i/N}{L}$

$$w^i|_{]x_{i-1},x_i[\cup]0,L[} = \frac{x-x_i}{x_{i-1}-x_i}, \quad w^i|_{]x_i,x_{i+1}[\cup]0,L[} = \frac{x-x_i}{x_{i+1}-x_i}, \quad w^i|_{]0,L[\setminus]x_{i-1},x_{i+1}[} = 0$$

C'est a dire qu'il faut commencer par construire du classe qui modélise la matrice

$$\mathcal{M}_{\alpha\beta} = \left(\int_{]O,L[} \alpha w^i w^j + \beta w^{i\prime} w^{j\prime}\right)_{i=1...M, j=1...M}$$

en copiant la classe `MatriceLaplacien1D`.
Puis, il suffit, d'approcher le vecteur $F = (f_i)_{i=0..M} = \big(\int_{]O,L[} f w^i\big)_{i=0..M}$ par le produit $F_h = \mathcal{M}_{1,0}\left(f(x_i)\right)_{i=1,M}$, ce qui revient à remplacer f par

$$f_h = \sum_i f(x_i) w^i$$

.

Q1 ecrire l'algorithme mathematiquement, avec des matrices

Q2 Trancrire l'algorithme

Q3 Visualiser les résultat avec gnuplot, pour cela il suffit de crée un fichier par pas de temps contenant la solution, stocker dans un fichier, en inspirant de

```
#include<sstream>
#include<ofstream>
#include<iostream>
  ....
  stringstream ff;
  ff « "sol-"« temps « ends;
  cout « " ouverture du fichier" «ff.str.c_str()« endl;
  {
     ofstream file(ff.str.c_str());
     for (int i=0;i<=M;++i)
       file «x[i]« endl;
   }        //    fin bloque => destruction de la variable file
                                //    => fermeture du fichier
  ...
```

2.3 Des classes pour les Graphes

Les sources sont disponibles : `ftp://www.freefem.org/snc++/Graphe`.

2.4 Definition et Mathématiques

- On définit un graphe **G** comme étant un couple (X, G) où X est un ensemble appelé l'ensemble des sommets et G un sous ensemble de X^2 appelé l'ensemble des arcs ou arêtes orientées.
- une arête est un arc qui a perdu son orientation
- une chaine est une suite d'arêtes $(\mu_i, (i = 1, n))$ de G telle que les arêtes μ_i et μ_{i-1} aient un sommet s_i en commun si $i > 0$ et $i < n$ et telle que pour $i > 0$ et $i < n$, l'arête μ_i ait pour autre sommet le sommet s_{i+1}. Soit s_o (resp. s_n) le sommet de μ_1 (resp. μ_n) non dans μ_2 (resp. μ_{n-1}). Si la chaine est réduite a une arête μ_1, s_o et s_1 seront les deux sommets de l'arête μ_1.
- On définit la relation d'équivalence sur les sommets de **G** par : $x \equiv y$ il existe une chaine reliant x à y.
- les composantes connexes de **G** seront les classes d'équivalence de la relation $\equiv$.
- un cycle est une chaine fermée (le sommet s_o est égale à s_n) telle que tous les arcs soient distincts.
- à chaque cycle on peut associer un vecteur μ de $\mathbb{R}^G$ dont les composantes sont définies par

$$\mu_i = \begin{cases} +0 & \text{si l'arc } i \text{ n'est pas dans le cycle,} \\ +1 & \text{l'arc } i \text{ a le même sens que le sens du parcours du cycle,} \\ -1 & \text{sinon.} \end{cases} \quad (2.1)$$

  par abus de langage, on confondra souvent un cycle et son vecteur associé. On notera par $V(\mathbf{G})$ l'espace vectoriel engendré par les cycles de **G** .

(x) **Théorème 2.1** *Soit* **G** *un graphe à n sommets, m arcs et p composantes connexes, alors la dimension de $V(\mathbf{G})$ est égal à $m - n + p$.*

Démonstration . voir Berge [?]

(x) **Théorème 2.2** *Définition Soit $\mathbf{H} = (X, H)$ un graphe. Les propriétés suivantes sont équivalentes et caractérisent un arbre (n nombre de sommets, m le nombre d'arcs).*

1. **H** *est connexe, sans cycle*
2. **H** *est connexe et $m = n - 1$*
3. **H** *est sans cycle et $m = n - 1$*
4. **H** *est sans cycle et en ajoutant un arc quelconque on crée un cycle unique*
5. **H** *est connexe et si l'on supprime un arc quelconque il n'est plus connexe.*

Démonstration. voir Berge [?]

(x) **Théorème 2.3** *Définition Soit* $\mathbf{G} = (X, G)$ *un graphe connexe alors il existe un sous graphe* $\mathbf{H} = (X, H)$ *tel que* H *soit inclus dans* G *et tel que* $\mathbf{H}$ *soit un arbre.* $\mathbf{H}$ *sera appelé un arbre maximal de* $\mathbf{G}$ *.*

Démonstration. Soit $\mathbf{H} = (X, H)$ un graphe connexe tel que pour tout sous ensemble H' de H et différent de H, le graphe (X, H') est non connexe (il en existe car le graphe (X, G) est connexe, et les ensembles G et X sont finis). II est alors clair que $\mathbf{H}$ est un arbre d'après la propriété v) du théorème A.2.

(x) **Théorème 2.4** *Soit* $\mathbf{G} = (X, G)$ *un graphe connexe et soit* $\mathbf{H} = (X, H)$ *un arbre maximal de* G*, si* i *est un arc de* $\mathbf{G}$ *et non de* $\mathbf{H}$ *son adjonction à* H *détermine un cycle* γ^i *unique tel que son vecteur associé* $(\gamma^i_j)_{j\in G}$ *vérifie :* γ^i_i *est égal à 1. L'ensemble* $\{\gamma^i / i \in G\backslash H\}$ *forme une base de* $V(\mathbf{G})$*.*

Démonstration. Berge [?]

Soit $\mathbf{G} = (X, G)$ un graphe, on appelera un flot un vecteur $(\alpha_i)_{i\in G}$ tel que pour tout sommet s de X on ait

$$\sum_{i\in I_s^+} \alpha_i - \sum_{i\in I_s^-} \alpha_i = 0; \tag{2.2}$$

où I_s^+ (resp. I_s^-) est l'ensemble des arcs (x, y) de G tel que s soit égal à x (resp. y) ce qui est identique à l'ensemble des arcs partant (resp. arrivant) de s.

Remarque 2.2 *L'équation (2.2) nous dit que pour un flot tout ce qui entre en un sommet en ressort.*

Proposition A.1. l'ensemble des flots d'un graphe sans cycle est réduit à $\{0\}$.

Démonstration. Faisons une démonstration par récurrence sur l'ordre de $\mathbf{G}$ (nombre de sommets de $\mathbf{G}$). Si l'ordre de $\mathbf{G}$ est égal à un, on a trivialement la propriété. On suppose la propriété vraie pour tout graphe sans cycle d'ordre n.
Soit $\mathbf{G} = (X, G)$ un graphe sans cycle d'ordre $n + 1$ avec m arc. Soit $\alpha = (\alpha_i)_{i\in G}$ un flot de $\mathbf{G}$. Comme $\mathbf{G}$ est sans cycle, le théorème A.1 nous dit que $m - n + 1 < 0$, ce qui prouve qu'il existe un sommet j de G contenu dans une seule arête k de G. D'où d'après (2.2) , la composante α_k est nulle. On en déduit que $(\alpha_i)_{i\in G\backslash\{k\}}$ est un flot de $(X\backslash\{j\}, G\backslash\{k\})$. Il suffit d'appliquer l'hypothèse de récurrence pour montrer la propriété.

(x) **Théorème 2.5** *l'espace vectoriel des flots est égal à l'espace vectoriel* $V(\mathbf{G})$ *engendré par les cycles du graphe* $\mathbf{G}$

Démonstration. On peut supposer $\mathbf{G}$ connexe car sinon on travaillera sur les composantes connexes de $\mathbf{G}$. Si $\mathbf{G}$ est connexe, il existe un arbre maximal $\mathbf{H} = (X, H)$ de $\mathbf{G} = (X, G)$. Soient les cycles (γ^i) associés à l'arbre pour $i \in G\backslash H$ (cf. théorème A.4). Il est clair que les

vecteurs $(\gamma^i_j)_{j\in G}$ de $\mathbb{R}^G$ sont des flots. Donc tout vecteur de $V(\mathbf{G})$ est un flot. Soit $(\alpha_i)_{i\in G}$ un flot, on construit le flot $(\beta_j)_{j\in G}$ suivant

$$\beta_j = \alpha_j - \sum_{i\in G\backslash H} \alpha_j \gamma^i_j. \tag{2.3}$$

Comme par construction $\beta_j = 0$ pour tout $j \in G\backslash H$,on en déduit que le vecteur $(\beta_j)_{j\in H}$ est un flot de H. La proposition précédente et le théorème A.2 i) nous montrent que β_j est nul pour tout j dans H. D'où tout flot est somme de cycles ce qui prouve l'autre inclusion.

Pour construire des relévements des conditions aux limites, ou pour retrouver la pression nous sommes amenés à introduire les deux problèmes abstrait duaux suivant :
Soit $\mathbf{G} = (X, G)$ un graphe connexe composé de n sommets et de m arêtes.
A chaque arc j on associe le vecteur $\mathbf{e}^j = (e^j_i)_{i\in X}$ de $\mathbb{R}^X$ tel que

$$e^j_i = \begin{cases} 0 & \text{si } i \text{ n'est pas un sommet de l'arc } j\,; \\ +1 & \text{si l'arc } j \text{ est de la forme } (i,k)\ (k \in X)\,; \\ -1 & \text{si l'arc } j \text{ est de la forme } (k,i)\ (k \in X). \end{cases} \tag{2.4}$$

2.4.1 Premiere Implémentation

Ici nous voulons, pourvoir calculer l'ordre des sommets d'un graphe, respectivement $o^{\pm}(s) = \#I^{\pm}(s)$ où les deux $I^{\pm}$ sont définies pour (2.2).
Puis, pour calculer les componsante connexe du graphe, nous aurons besoin de construire Les ensemble $I^+(s) \cup I^-(s)$

```
#include <cassert>
#include <iostream>
using namespace std;
                                                    //    cas 1 avec des copie de sommet
                                                        //    cas 2 sans copie de sommet

class Sommet { public :
  int numero;                                                 //    le numero du sommet
  int op,om;                                     //    nombre arc partant et incidante
  int couleur;                                              //    la couleur du sommet
  Sommet(int n=0) : numero(n),op(0),om(0),couleur(0) {} //    constructeur par
default
  void Set(int i) {numero=i;}
  Sommet & operator=(int i) { numero=i;return *this;}
  int o() const {return om+op;}

                      //    pas d'operateurs de copies, c'est une class simple
                            //    dans le cas 2 on interdit la copie de sommet
private:
  Sommet(const  Sommet &);
  void operator =(const  Sommet &);
};

inline ostream & operator<<(ostream & f, const Sommet &  s)
{ if(&s) f << " s " << s.numero << " c " << s.couleur << " ";
 else cout << " s NULL ";
```

```
 return f;}

class Arc {public:
  int numero;                                         //    le numero de l'arc
  /*                                                   //    cas (1) bug copie
  Sommet extremite[2];                        //    un tableau il faut refechir
  */
                                                                   //    cas 2
  Sommet *extremite[2];   //    un tableau de 2 pointeurs ici il faut refechir

  const Sommet & operator[](int i) const
  { assert(i >=0 && i <=1); return (*extremite[i]);}    //    cas 1 retire l'*

  Sommet & operator[](int i)
  { assert(i >=0 && i <=1); return (*extremite[i]);}    //    cas 1 retire l'*

  Arc(): numero(0) {extremite[0]=extremite[1]=0; }   //    un constructeur par
defaut pour faire de tableau arc

   //   cas 1 void Set( int i,Sommet & a, Sommet & b) numero=i;extremite[0]=a;
extremite[1]=b;
  void Set(int i,Sommet & a, Sommet & b) {numero=i;extremite[0]=&a; extremite[1]=&b;}
                      //    pas d'operateurs de copies, c'est une class simple

                                //    dans le cas 2 on interdit la copie d'arc
private:
  Arc(const  Arc & );
  void operator =(const  Arc & );
};

inline ostream & operator«(ostream & f, const Arc &  a)
{ f « " a " « a.numero « "\t: " « a[0] « " -> " « a[1] « " "; return f;}




class Graphe { public:
  int n;                                           //    n nombre de sommets
  int m;                                               //    m nombre d'arcs
  Sommet * s;                                   //    le tableau des sommets
  Arc * a;                                         //    le tableau des Arcs

  int *p , *v;                                //    tableau pour les voisins
                       //    la liste des voisins du sommets s seront dans
                               //    for (k=p[s]; k<p[s+1]; k++) s[v[k]];


  void MiseAJour();
  Graphe(int  );                                           //    un graphe c
  Graphe(const char * filename);              //    un graphe lue dans filename
  ~Graphe()
  {
    cout « " delete Graphe: nb sommets " « n « " , nb arcs  " « m « endl;
    delete [] s;
    delete [] a;
```

```
  }

                                                   //    pas operateur de copie
private:
  Graphe(const Graphe &);
  Graphe & operator=(const Graphe &);
};

                       //    trop grandes le fonction est mise dans Graphe.cpp
ostream & operator«(ostream & f, const Graphe &  a);
```

les methodes de la classe

```
#include <cassert>
#include <iostream>
using namespace std;
                                            //    cas 1 avec des copie de sommet
                                                //    cas 2 sans copie de sommet

class Sommet { public :
  int numero;                                           //    le numero du sommet
  int op,om;                                //    nombre arc partant et incidante
  int couleur;                                         //    la couleur du sommet
  Sommet(int n=0) : numero(n),op(0),om(0),couleur(0) {} //    constructeur par
default
  void Set(int i) {numero=i;}
  Sommet & operator=(int i) { numero=i;return *this;}
  int o() const {return om+op;}

                      //    pas d'operateurs de copies, c'est une class simple
                            //    dans le cas 2 on interdit la copie de sommet
private:
  Sommet(const  Sommet &);
  void operator =(const  Sommet &);
};

inline ostream & operator«(ostream & f, const Sommet &  s)
{ if(&s) f « " s " « s.numero « " c " « s.couleur « " ";
 else cout « " s NULL ";
 return f;}

class Arc {public:
  int numero;                                            //    le numero de l'arc
  /*                                                      //    cas (1) bug copie
  Sommet extremite[2];                          //    un tableau il faut refechir
  */
                                                                    //    cas 2
  Sommet *extremite[2];   //    un tableau de 2 pointeurs ici il faut refechir

  const Sommet & operator[](int i) const
  { assert(i >=0 && i <=1); return (*extremite[i]);}    //    cas 1 retire l'*

  Sommet & operator[](int i)
  { assert(i >=0 && i <=1); return (*extremite[i]);}    //    cas 1 retire l'*
```

```
  Arc(): numero(0) {extremite[0]=extremite[1]=0; }   //    un constructeur par
defaut pour faire de tableau arc

   //   cas 1 void Set( int i,Sommet & a, Sommet & b) numero=i;extremite[0]=a;
extremite[1]=b;
  void Set(int i,Sommet & a, Sommet & b) {numero=i;extremite[0]=&a; extremite[1]=&b;}
                      //    pas d'operateurs de copies, c'est une class simple

                                //    dans le cas 2 on interdit la copie d'arc
private:
  Arc(const  Arc & );
  void operator =(const  Arc & );
};

inline ostream & operator«(ostream & f, const Arc &  a)
{ f « " a " « a.numero « "\t: " « a[0] « " -> " « a[1] « " "; return f;}




class Graphe { public:
  int n;                                                  //    n nombre de sommets
  int m;                                                      //    m nombre d'arcs
  Sommet * s;                                          //    le tableau des sommets
  Arc * a;                                                //    le tableau des Arcs

  int *p , *v;                                       //    tableau pour les voisins
                           //    la liste des voisins du sommets s seront dans
                                   //    for (k=p[s]; k<p[s+1]; k++) s[v[k]];


  void MiseAJour();
  Graphe(int  );                                                //    un graphe c
  Graphe(const char * filename);               //    un graphe lue dans filename
  ~Graphe()
  {
    cout « " delete Graphe: nb sommets " « n « " , nb arcs  " « m « endl;
    delete [] s;
    delete [] a;
  }

                                                       //    pas operateur de copie
private:
  Graphe(const Graphe &);
  Graphe & operator=(const Graphe &);
};

                       //    trop grandes le fonction est mise dans Graphe.cpp
ostream & operator«(ostream & f, const Graphe &  a);
```

la solution

```
#include "Graphe.hpp"
#include <fstream>
```

```
       //    ruse pour calcule la profondeur maximal de pile d'appele recursif
                                        //    on utilise des variables global.
int kpile=0;                                             //    profondeur courante
int mpile =0;                                             //    profondeur maximal
                                                              //    ------------

void Coloriage(const Graphe & G,Sommet & s,int couleur)
{
  kpile++;                                         //    incrementation de la pile
  mpile = max(kpile,mpile);                                      //    stokage max
  assert(couleur);
  s.couleur = couleur;
                                                              //   cout « s « endl;
  for (int k=G.p[s.numero]; k <G.p[s.numero+1];k++)
    {
      Sommet & sv = G.s[G.v[k]];                         //    sv est voisin de s
      if  (!sv.couleur ) Coloriage(G,sv,couleur);
    }
  kpile-;                                           //    decrementation de pile
}

int main(int argc,char ** argv)
{
  const char * filename = argc>1? argv[1] : "graphe.txt";
  Graphe G(filename);

                                         //    si le graphe est petit on affiche
  if (G.m < 100)
  cout « " " « filename « " " «  G « endl;


  int couleur=0;
  for (int s=0;s<G.n;++s)
    G.s[s].couleur = 0;

  for (int s=0;s<G.n;++s)
    if ( G.s[s].couleur == 0)
      {
couleur++;
cout « " Nouvelle composante connexe sommet depart s :  " « s
     « "  Nu = " « couleur « endl;
Coloriage(G,G.s[s],couleur);
      }
                                         //    complexite de l'algorithme n+m
          //    effectivement on a vue, 1 fois les sommet et une fois les arcs
                      //    par contre la profondeur de recurcivite est enorme
                          //    donc cet algoritme ne marche moralemenent pas.
                                 //    il suffit de « derecuciver » algorithme
                                          //    en gerant une pile a la main.
  Cout « " Nb de composantes connexe du Graphe " « couleur « endl;
  cout « " profondeur max de pile = " « mpile « endl;

  return 0;}
```

2.5 Maillage et Triangulation 2D

Nous allons définir les outils informatiques en C++ pour utiliser des maillages, pour cela nous utilisons la classe R2 définie en ??. Le maillage est formé de triangles qui sont définis par leurs trois sommets. Mais attention, il faut numéroter les sommets. La question classique est donc de définir un triangle : soit comme trois numéro de sommets, ou soit comme trois pointeurs sur des sommets (nous ne pouvons pas définir un triangle comme trois références sur des sommets car il est impossible initialiser des références par défaut). Les deux sont possibles, mais les pointeurs sont plus efficaces pour faire des calculs, d'où le choix de trois pointeurs pour définir un triangle. Maintenant les sommets seront stockés dans un tableau donc il est inutile de stocker le numéro du sommet dans la classe qui définit un sommet, nous ferons une différence de pointeur pour retrouver le numéro d'un sommet, ou du triangle du maillage.
Un maillage (classe de type Mesh) contiendra donc un tableau de triangles (classe de type Triangle) et un tableau de sommets (classe de type Vertex), bien sur le nombre de triangles (nt) , le nombre de sommets (nv), de plus il me paraît naturel de voir un maillage comme un tableau de triangles et un triangle comme un tableau de 3 sommets.
Remarque : Les sources de ces classes et méthodes sont disponible dans ftp://www.freefem.org/snc++/mesh/Mesh2d.hpp et ftp://www.freefem.org/snc++/mesh/Mesh2d.cpp.

2.5.1 La classe Label

Nous avons vu que le moyen le plus simple de distinguer les sommets appartenant à une frontière était de leur attribuer un numéro logique ou étiquette (*label* en anglais). Rappelons que dans le format FreeFem++ les sommets intérieurs sont identifiés par un numéro logique nul.
De manière similaire, les numéros logiques des triangles nous permettront de séparer des sous-domaines Ω_i, correspondant, par exemple, à des types de matériaux avec des propriétés physiques différentes.
La classe Label va contenir une seule donnée (lab), de type entier, qui sera le numéro logique.

Listing 2.3 *(Mesh2d.hpp - la classe Label)*

```
 class Label {
  friend ostream& operator <<(ostream& f,const Label & r  )
  { f <<  r.lab; return f; }
  friend istream& operator >>(istream& f, Label & r  )
  { f >>  r.lab; return f; }
public:
  int lab;
  Label(int r=0):lab(r){}
  int onGamma() const { return lab;}
};
```

Cette classe n'est pas utilisée directement, mais elle servira dans la construction des classes pour les sommets et les triangles. Il est juste possible de lire, écrire un label, et tester si elle est nulle.

Listing 2.4 *(utilisation de la classe Label)*

```
Label r;
cout « r;                                          //    écrit r.lab
cin  » r;                                          //    lit r.lab
if(r.onGamma()) { ..... }                    //    à faire si la r.lab != 0
```

2.5.2 La classe `Vertex` (modélisation des sommets)

Il est maintenant naturel de définir un sommet comme un point de $\mathbb{R}^2$ et un numéro logique `Label`. Par conséquent, la classe `Vertex` va dériver des classes `R2` et `Label` tout en héritant leurs données membres et méthodes (voir le paragraphe **??**) :

Listing 2.5 *(Mesh2d.hpp - la classe Vertex)*

```
class Vertex : public R2,public Label {
public:
  friend ostream& operator «(ostream& f, const Vertex & v )
  { f « (R2) v « ' ' « (Label &) v   ; return f; }
  friend istream& operator » (istream& f,  Vertex & v )
  { f » (R2 &) v » (Label &) v; return f; }
  Vertex() : R2(),Label(){};
  Vertex(R2 P,int r=0): R2(P),Label(r){}
private:
  Vertex(const Vertex &);              //    interdit la construction par copie
  void operator=(const Vertex &);        //    interdit l'affectation par copie
};
```

Nous pouvons utiliser la classe `Vertex` pour effectuer les opérations suivantes :

Listing 2.6 *(utilisation de la classe Vertex)*

```
Vertex V,W;                              //    construction des sommets V et W
cout « V ;                                    //    écrit V.x, V.y , V.lab
cin  » V;                                       //    lit V.x, V.y , V.lab
R2  O = V;                                         //    copie d'un sommet
R2  M = ( (R2) V +  (R2) W ) *0.5;     //    un sommet vu comme un point de R2
(Label) V               //    la partie label d'un sommet (opérateur de cast)
if (!V.onGamma()) { .... }  //    si V.lab = 0, pas de conditions aux limites
```

Remarque 2.3 *Les trois champs `(x,y,lab)` d'un sommet sont initialisés par `(0.,0.,0)` par défaut, car les constructeurs sans paramètres des classes de base sont appelés dans ce cas.*

2.5.3 La classe `Triangle` (modélisation des triangles)

Un triangle sera construit comme un tableau de trois pointeurs sur des sommets, plus un numéro logique (*label*). Nous rappelons que l'ordre des sommets dans la numérotation locale ($\{0,1,2\}$) suit le sens trigonométrique. La classe `Triangle` contiendra également une donnée supplémentaire, l'aire du triangle (`area`), et plusieurs fonctions utiles :

- `Edge(i)` qui calcule le vecteur «arête du triangle» opposée au sommet local i ;
- `H(i)` qui calcule directement le gradient de la i coordonnée barycentrique λ^i par la formule :

$$\nabla\lambda^i|_K = H^i_K = \frac{(q^j - q^k)^\perp}{2\,aire_K} \tag{2.5}$$

  où l'opérateur $\perp$ de $\mathbb{R}^2$ est défini comme la rotation de $\pi/2$, ie. $(a,b)^\perp = (-b,a)$, et où les q^i, q^j, q^k sont les coordonnées des 3 sommets du triangle.

Listing 2.7 *(Mesh2d.hpp - la classe Triangle)*

```
class Triangle: public Label {
  Vertex *vertices[3];         //    tableau de trois pointeurs de type Vertex
public:
  R area;
  Triangle(){};                                  //    constructeur par défaut vide

  Vertex & operator[](int i) const {
    ASSERTION(i>=0 && i <3);
    return *vertices[i];}   //    évaluation du pointeur -> retourne un sommet

  void set(Vertex * v0,int i0,int i1,int i2,int r) {
    vertices[0]=v0+i0; vertices[1]=v0+i1; vertices[2]=v0+i2;
    R2 AB(*vertices[0],*vertices[1]);
    R2 AC(*vertices[0],*vertices[2]);
    area = (AB^AC)*0.5;
    lab=r;
    ASSERTION(area>0); }

  R2 Edge(int i) const {
    ASSERTION(i>=0 && i <3);
    return R2(*vertices[(i+1)%3],*vertices[(i+2)%3]);}     //    vecteur arête
opposé au sommet i

  R2 H(int i) const { ASSERTION(i>=0 && i <3);               //    valeur de ∇λ^i
  R2 E=Edge(i);return E.perp()/(2*area);}
  R lenEdge(int i) const {
    ASSERTION(i>=0 && i <3);
    R2 E=Edge(i);return sqrt((E,E));}
```

```
private:
  Triangle(const Triangle &);         //    interdit la construction par copie
  void operator=(const Triangle &);     //    interdit l'affectation par copie
};
```

Remarque 2.4 *La construction effective du triangle n'est pas réalisée par un constructeur, mais par la fonction* `set`. *Cette fonction est appelée une seule fois pour chaque triangle, au moment de la lecture du maillage (voir plus bas la classe* `Mesh`*).*

Remarque 2.5 *Les opérateurs d'entrée-sortie ne sont pas définis, car il y a des pointeurs dans la classe qui ne sont pas alloués dans cette classe, et qui ne sont que des liens sur les trois sommets du triangle (voir également la classe* `Mesh`*).*

Regardons maintenant comment utiliser cette classe :

Listing 2.8 *(utilisation de la classe Triangle)*

```
                                  //    soit T un triangle de sommets A,B,C ∈ IR²
                                                  //    ------------------------
  const Vertex & V =  T[i];                     //    le sommet i de T (i ∈ {0,1,2}
  double a = T.area;                                        //    l'aire de T
  R2 AB = T.Edge(2);                    //    "vecteur arête" opposé au sommet 2
  R2 hC = T.H(2);             //    gradient de la fonction de base associé au
sommet 2
  R l = T.lenEdge(i);           //    longueur de l'arête opposée au sommet i
  (Label) T  ;                                  //    la référence du triangle T

  Triangle T;
  T.set(v,ia,ib,ic,lab);     //    construction d'un triangle avec les sommets
                                //    v[ia],v[ib],v[ic] et l'étiquette lab
                                      //    (v est le tableau des sommets)
```

2.5.4 La classe `Mesh` (modélisation du maillage)

Nous présentons, pour finir, la classe maillage (`Mesh`) qui contient donc :

- le nombre de sommets (`nv`), le nombre de triangles (`nt`), le nombre d'arêtes frontières (`neb`) ;
- le tableau des sommets ;
- le tableau des triangles ;
- et le tableau des arêtes frontières (sa construction sera présentée dans la section suivante).

Listing 2.9 *(Mesh2d.hpp - la classe Mesh)*

```
class Mesh { public:
  int nt,nv,neb;
  R area;

  Vertex *vertices;
  Triangle *triangles;

  Triangle & operator[](int i) const {return triangles[CheckT(i)];}
  Vertex & operator()(int i) const {return vertices[CheckV(i)];}

  inline Mesh(const char * filename);        //    lecture du fichier def. par
filename

  int operator()(const Triangle & t) const {return CheckT(&t - triangles);}
  int operator()(const Triangle * t) const {return CheckT(t - triangles);}
  int operator()(const Vertex & v) const {return CheckV(&v - vertices);}
  int operator()(const Vertex * v) const{return CheckT(v - vertices);}
  int operator()(int it,int j) const {return (*this)(triangles[it][j]);}

                                //    vérification des dépassements de tableau
  int CheckV(int i) const { ASSERTION(i>=0 && i < nv); return i;}
  int CheckT(int i) const { ASSERTION(i>=0 && i < nt); return i;}

private:
  Mesh(const Mesh &);                    //    interdit la construction par copie
  void operator=(const Mesh &);          //    interdit l'affectation par copie
};
```

Avant de voir comment utiliser cette classe, quelques détails techniques nécessitent plus d'explications :

- Pour utiliser les opérateurs qui retournent un numéro, il est fondamental que leur argument soit un pointeur ou une référence ; sinon, les adresses des objets seront perdues et il ne sera plus possible de retrouver le numéro du sommet qui est donné par l'adresse mémoire.
- Les tableaux d'une classe sont initialisés par le constructeur par défaut qui est le constructeur sans paramètres. Ici, le constructeur par défaut d'un triangle ne fait rien, mais les constructeurs par défaut des classes de base (ici les classes `Label, Vertex`) sont appelés. Par conséquent, les labels de tous les triangles sont initialisées et les trois champs `(x,y,lab)` des sommets sont initialisés par `(0.,0.,0)`. Par contre, les pointeurs sur sommets sont indéfinis (tout comme l'aire du triangle).

Tous les problèmes d'initialisation sont résolus une fois que le constructeur avec arguments est appelé. Ce constructeur va lire le fichier `.msh` contenant la triangulation.

Listing 2.10 *(Mesh2d.hpp - constructeur de la classe Mesh)*

```
inline Mesh::Mesh(const char * filename)
```

```
{                                                   //    lecture du maillage
  int i,i0,i1,i2,ir;
  ifstream f(filename);
  if(!f) {cerr « "Mesh::Mesh Erreur a l'ouverture - fichier " « filename«endl;
           exit(1);}
  cout « " Lecture du fichier \"" «filename«"\""«  endl;

  f » nv » nt » neb;
  cout « " Nb de sommets " « nv « " " « " Nb de triangles " « nt
                               « " Nb d'aretes frontiere " « neb « endl;
  assert(f.good() && nt && nv);

  triangles = new Triangle [nt];         //   alloc mémoire - tab des triangles
  vertices = new Vertex[nv];               //   alloc mémoire - tab des sommets

  area=0;
  assert(triangles && vertices);

  for (i=0;i<nv;i++)                      //    lecture de nv sommets (x,y,lab)
    f » vertices[i],assert(f.good());

  for (i=0;i<nt;i++) {
    f » i0 » i1 » i2 » ir;        //    lecture de nt triangles (ia,ib,ic,lab)
    assert(f.good() && i0>0 && i0<=nv && i1>0 && i1<=nv && i2>0 && i2<=nv);
    triangles[i].set(vertices,i0-1,i1-1,i2-1,ir);        //    construction du
triangle
    area += triangles[i].area;}                 //    calcul de l'aire totale du
maillage

  for (i=0;i<neb;i++) {
    f » i0 » i1  » ir;                     //    lecture de neb arêtes frontières
    assert(f.good() && i0>0 && i0<=nv && i1>0 && i1<=nv );
    bedges[i].set(vertices,i0-1,i1-1,ir);}  //    construction de chaque arête

   cout « " Fin lecture : aire du maillage = " « area «endl;
}
```

L'utilisation de la classe `Mesh` pour gérer les sommets, les triangles et les arêtes frontières devient maintenant très simple et intuitive.

Listing 2.11 *(utilisation de la classe Mesh)*

```
  Mesh Th("filename");          //    lit le maillage Th du fichier "filename"
  Th.nt;                                            //    nombre de triangles
  Th.nv;                                              //    nombre de sommets
  Th.neb;                                    //    nombre d'arêtes frontières
  Th.area;                                    //    aire du domaine de calcul

  Triangle & T = Th[i];                          //    triangle i , int i∈ [0,nt[
  Vertex   & V = Th(j);                            //    sommet j , int j∈ [0,nv[
  int j = Th(i,k);      //    numéro global du sommet local k∈ [0,3[ du triangle
i∈ [0,nt[
```

```
  Vertex & W=Th[i][k];       //    référence du sommet local k∈ [0,3[ du triangle
i∈ [0,nt[

  int ii = Th(T);                                          //    numéro du triangle T
  int jj = Th(V);                                            //    numéro du sommet V

  assert( i == ii && j == jj);                                     //    vérification
```

Chapitre 3

Algorithmes

3.1 Chaînes et Chaînages

3.1.1 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.

3.1.2 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 3.1

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.*

3.1.3 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

$$\mathtt{nbe} = \mathtt{nt} + \mathtt{nv} + \mathtt{nb_de_trous} - \mathtt{nb_composantes_connexes}, \quad (3.1)$$

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 3.2

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 3.1 seront simultanées.

Algorithme 3.3

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

```
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`.

3.1.4 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 3.1.3), `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 3.1, 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 :

$$\mathtt{p} \longrightarrow (\mathtt{k},\mathtt{j}) = (\mathtt{k} = \mathtt{p}/\mathtt{m}, \mathtt{j} = \mathtt{p}\%\mathtt{m}). \tag{3.2}$$

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

Algorithme 3.4

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`.

3.1.5 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.

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∈[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 :

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

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

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 3.4 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.

Algorithme 3.5

Construction de la structure d'une matrice morse

```
#include "MatMorse.hpp"
MatriceMorseSymetrique::MatriceMorseSymetrique(const  Mesh &
Th)
{
  int color=0, * mark;
  int i,j,jt,k,p,t;
  n = m = Th.nv;
  ligne.set(new int [n+1],n+1);
  mark = new int [n];
   //   construction optimisee 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 non nuls a priori
                                    //    de la matrice (step 0)
                                     //    et stokaquae (step 1)
  for (int step=0;step<2;++step)
    {
      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;                  //    marquage
                if(step) colonne[nbcoef]=j;        //    ajout
                nbcoef++;             //    nouveau => comptage
              }

      if(step==0) {                //    2) allocations mémoires
        colonne.set( new int [ nbcoef],nbcoef);
        a.set(new double [nbcoef],nbcoef);
      }
    }
                    //    3) 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;
}
```

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

3.2 Algorithmes de trie

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;}
    }
  }
}
```

3.3 Algorithmes de recherche

Le but est d'écrire un algorithme de recherche d'un triangle K dans un pour maillage est convexe $\mathcal{T}_h$ contenant un point (x,y) en $O(log_2(n_T))$.
Pour cela voila un algorithme tres simple et facile a programmer :

Algorithme 3.6

Partant du triangle K,
Pour les 3 arêtes $(a_i,b_i), i = 0,1,2$ du triangle K, tournant dans le sens trigonométrique, calculer l'aire des 3 triangles (a_i,b_i,p) si le 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 arête associée à une aire négative (les ambiguïtés sont levées aléatoirement).

Maintenant il faut initialiser cette algorithme, il faut une méthode pour trouver, rapidement un triangle proche d'un point donné.
Il y a deux méthodes :
- Si le point est arbitraire : il suffit de construire un arbre quaternaire, et mettre tous les points du maillage dans cette arbre, et associe a chaque sommet du maillage un triangle le contenant.
- Si les points recherche sont dans un autre maillage, alors l'on peut remarquer que les sommets d'un triangle (ou élément) sont généralement proche, et il suffit utiliser un parcours récurcif d'un maillage

3.3.1 Arbre quaternaire

Pour cela nous utiliserons, un arbre quaternaire comme suit :

Algorithme 3.7

Soit B_0 une boite carre contenant tous les points du maillage. On applique la procédure récursive suivante à B_0.
Soit B_α la boite indicée par α qui est une chaîne de 0,1,2 ou 3 (exemple $\alpha = 011231$) la longueur de la chaîne est note $l(\alpha)$.
procedure appliqué à la boite B_α :
- *la boite B_α contient moins de 5 points, on les stockes dans la boite et on a fini,*
- *sinon on découpe la boite B_α en 4 sous boites égale noté $B_{\alpha 0}, B_{\alpha 1}, B_{\alpha 2}, B_{\alpha 3}$ (on stocke les 4 pointeurs vers ces boites dans la boite), et on applique la procédure au 4 nouvelles boites.*

Remarque : pour simplifier la programmation de l'arbre quaternaire, on peut construire une application qui transforme le point de $\mathbb{R}^2$ en points à coordonnées entières en construisant une application affine qui transforme par exemple B_0 en $[0, 2^{31}[^2$. Dans ce cas, les coordonnées entières permettent de limiter la profondeur de l'arbre à 32 niveaux, et d'utiliser les opérateurs entiers en nombre binaire du C comme `&`, `|`, `^`.

3.4 Parcours récurcif d'un maillage

Chapitre 4

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.

4.1 Bases théoriques

4.1.1 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 :

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

   .
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 ;
   La mesure d'un $n - simplex$ en dimension n est donnée par

$$det \begin{vmatrix} x_1^0 & \dots & x_1^d \\ \vdots & \dots & \vdots \\ x_d^0 & \dots & x_d^d \\ 1 & \dots & 1 \end{vmatrix}$$

et le d-simplex sera dit positif sa mesure est positive.
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 $\frac{(k+1)\times k\times(k-1)}{6}$ 2-simplex ,
- des hyperfaces sera l'ensemble des $\frac{(k+1)\times k}{2}$ (d-1)-simplex ,
-

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

4.1.2 Introduction

Commençons par définir la notion de maillage simplexial.

Définition 4.1 *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 :

$$\mathcal{O}_h \overset{def}{=} \overline{\bigcup_{K\in\mathcal{T}_{d,h}} K}^{\circ} \tag{4.1}$$

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 4.1 *Les triangles sont les composantes connexes de*

$$\mathcal{O}_h \setminus \bigcup_{(a,b)\in\mathcal{T}_{1,h}} [a, b]. \tag{4.2}$$

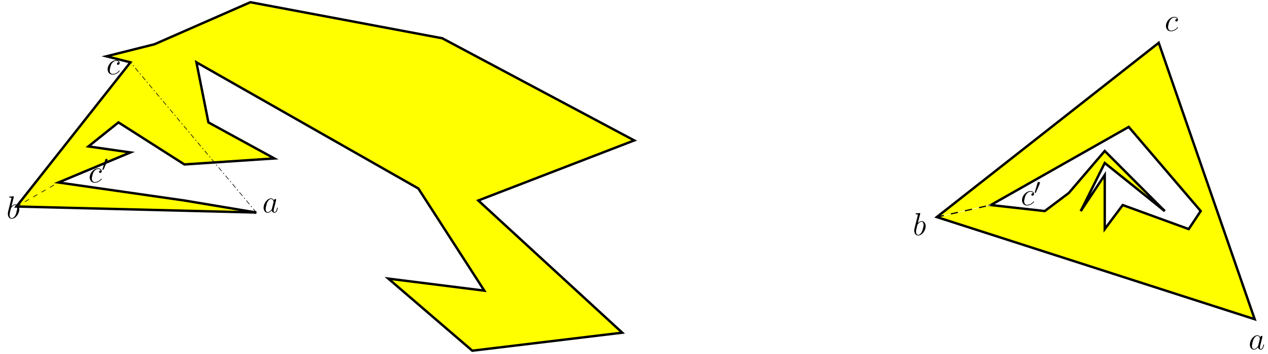
Commençons par donner un théorème fondamental en dimension $d = 2$.

Théorème 4.1 *Pour tout ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^2$, il existe un maillage de cet ouvert sans sommet interne.*

L'ensemble S des sommets de ce maillage sont les points anguleux du bord $\partial\mathcal{O}_h$.
La démonstration de ce théorème utilise le lemme suivant :

Lemme 4.1 *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 $\widehat{abc} < \pi$, et notons $T = \mathcal{C}(\{a, b, c\})$ le triangle fermé formé de a, b, c et $\overset{\circ}{T}$ le triangle ouvert.



Il y a deux cas :
- $\overset{\circ}{T} \cap S = \emptyset$ alors $]a, c[\subset \mathcal{O}_h$ et il suffit de pendre $]\alpha, \beta[=]a, c[$.
- Sinon, soit $\mathcal{C}$ le convexifié de cette intersection $\overset{\circ}{T} \cap S$. Par construction, ce convexifié $\mathcal{C}$ est inclus dans le triangle ouvert $\overset{\circ}{T}$. Soit le point anguleux c' du bord du convexifié le plus proche de b ; il sera tel que $]b, c'[\subset \mathcal{O}_h$, et il suffit de prendre $]\alpha, \beta[=]b, c'[$.

∎

Preuve du théorème 4.1:
Construisons par récurrence une suite d'ouverts $\mathcal{O}^i, i = 0, ..., k$, avec $\mathcal{O}^0 \overset{def}{=} \mathcal{O}$.
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.

$$\mathcal{O}^{i+1} \overset{def}{=} \mathcal{O}^i \setminus]a_i, b_i[\tag{4.3}$$

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 4.2 *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.*

4.1.3 Les données pour construire un maillage

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

$$\mathcal{S} \overset{def}{=} \{x^i \in \mathbb{R}^2 / i \in \{1, ..., N_p\}\} \tag{4.4}$$

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

$$\mathcal{A} \overset{def}{=} \{(sa_1^j, sa_2^j) \in 1, ..., N_p{}^2 / j \in \{1, ..., N_a\}\} \tag{4.5}$$

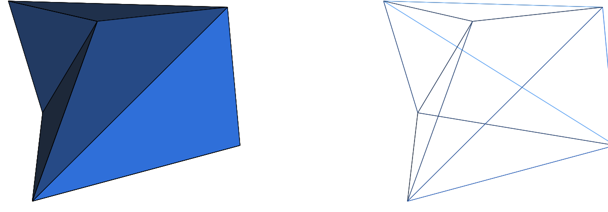


Figure 4.1 – example caption

- 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

$$\mathcal{SD} \stackrel{def}{=} \{(a^i, sens^i) \in \{1, ..., N_a\} \times \{-1, 1\} / i = 1, N_{sd}\} \tag{4.6}$$

qui peut être vide (cas par défaut).

4.1.4 Triangulation et Convexifié

Construction d'un premier maillage, soit f une fonction strictement convexe de $\mathbb{R}^d$ à valeur de $\mathbb{R}$, soit $F_f(x)$ l'application de $\mathbb{R}^d$ à valeur de $\mathbb{R}^{d+1}$ telle que $x = (x^1, .., x^d) \longrightarrow \tilde{x} = (x^1, .., x^d, f(x))$. et notons $e_{d+1} = (0, .., 0, 1) \in \mathbb{R}^{d+1}$.
Notons $\partial_\mathcal{C}(F_f(S)) = \{\tilde{x} \in \partial\mathcal{C}(F_f(S)) \quad / \quad n_{\tilde{x}}.e_{d+1} < 0\}$ où $n_{\tilde{x}}$ est la normal extérieur. la partie inférieure du bord du convexifié des points de S transformés par F_f.

Définition 4.2 *Le projeté $\partial_\mathcal{C}(F_f(S))$ sur R^d défini une triangulation du convexifié de S que l'on notera $\mathcal{T}_f$.*

La construction de maillage est très lie à la construction de convexifie en dimension $d + 1$. Je conjecture que quand f parcourt l'ensemble des fonction strictement convexe alors; $\mathcal{T}_f$ parcourt l'ensemble des triangulations conforme de sommet S.

4.1.5 Maillage de Delaunay-Voronoï

La méthode est basée sur les diagrammes de Voronoï :

Définition 4.3 *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 4.2).*

On peut donc écrire formellement :

$$V^i \stackrel{def}{=} \{x \in \mathbb{R}^2 / ||x - x^i|| \leq ||x - x^j||, \forall j \in \{1, ..., N_p\}\}. \tag{4.7}$$

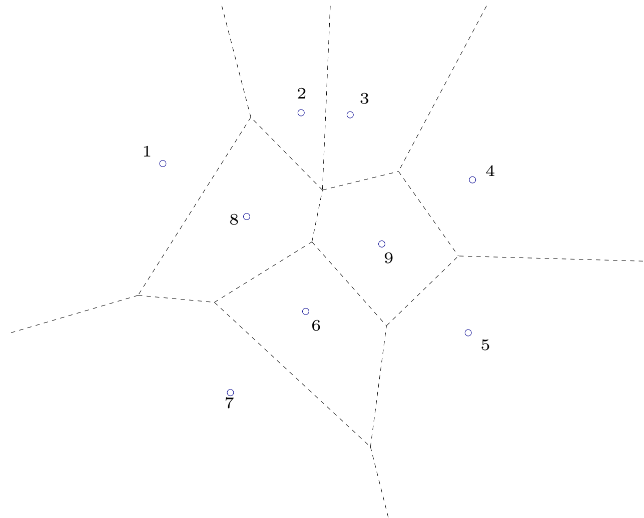


Figure 4.2 – Diagramme de Voronoï : les ensembles des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j.

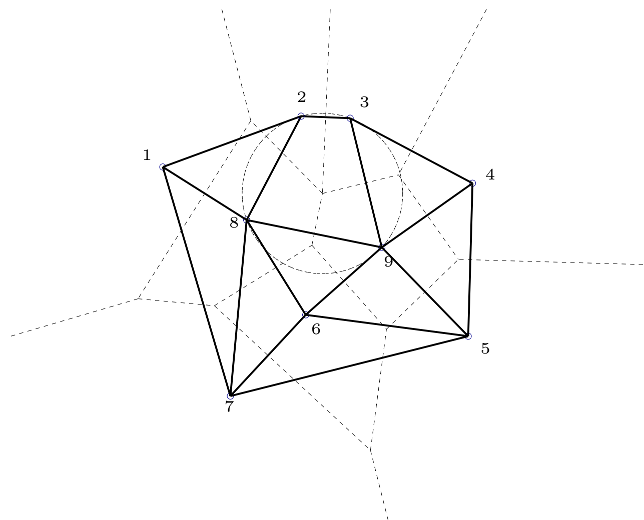


Figure 4.3 – Maillage de Delaunay : nous relions deux points x^i et x^j si les diagrammes V^i et V^j ont un segment en commun.

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 $\{x^{i^k_j}/j = 1, ..., n_k\}$ 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 $\{x^{i^k_j}, j = 1, ..., n_k\}$ en tournant dans le sens trigonométrique. Ce maillage est généralement formé de triangles, sauf si il y a des points cocycliques (voir figure 4.3 où $n_k > 3$).

Définition 4.4

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 4.3

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 $\mathcal{S}$.

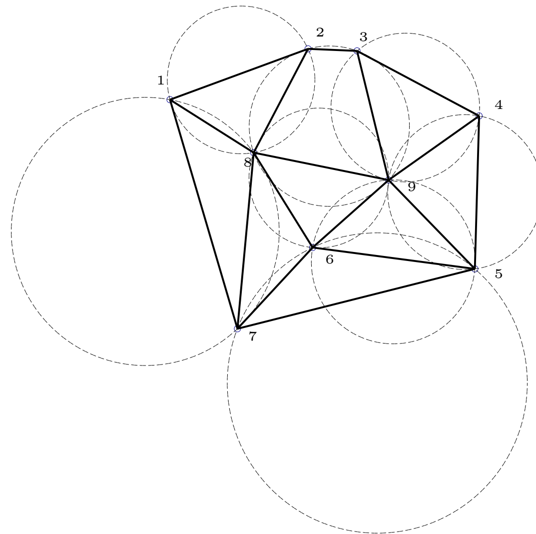


Figure 4.4 – Propriété de la boule vide : le maillage de Delaunay et les cercles circonscrits aux triangles.

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

Théorème 4.2

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 :

$$D(T) \cap \mathcal{T}_{0,h} = \emptyset. \tag{4.8}$$

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 : Montrons par l'absurde que si le maillage est de Delaunay alors il vérifie la propriété de la boule vide. Soit x^p un point de $\mathcal{S}$ dans la boule ouvert de d'élément $K = \{x^{k_1}, ...\}$ de centre c, on a $c \in V^{k_1}$ et $||c - x^p|| < ||c - x^{k_1}||$ ce qui est en contradiction avec la définition des V^k
Réciproquement : soit un maillage vérifiant (4.8).
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 (4.7), 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 4.4

Il est possible d'obtenir un maillage de Delaunay strict en changeant la propriété de la boule vide définie en (4.8) par la propriété stricte de la boule vide, définie comme suit :

$$\overline{D(T)} \cap \mathcal{T}_{0,h} = \overline{T} \cap \mathcal{T}_{0,h}, \tag{4.9}$$

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 (4.9) et (4.8) est qu'il peut exister dans (4.8) 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 4.2 *[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 vide, 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 4.1 La démonstration de l'alternative est laissée en exercice au lecteur.

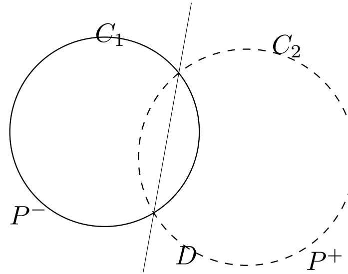


Figure 4.5 – Représentation graphique de l'alternative 1.

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 $j = 0, .., \underline{k_a}$ 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.

■

Remarque 4.5 *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 en déduisons une autre caractérisation du maillage de Delaunay :

Théorème 4.3 *La triangulation $\mathcal{T}_f$ défini en est la triangulation de Delaunay si $f(x) = ||x||^2$ et les maillages de Delaunay sont les uniques triangulations rendant convexe l'interpolé P_1 Lagrange de la fonction $x \rightarrow ||x||^2$.*

Preuve : Il suffit de montre le théoreme pour tous le maillages formés de deux éléments adjacent T_a et T_b par une hyper-face F de sommet $(p^1, .., p^d)$ et de sommets opposé a (resp. b), grâce au lemme de Delaunay. Nos deux éléments T_a et T_b(d-simplex positif) s'écrivent $T_a = (p^1, .., p^d, a)$ et $T_b = (p^2, p^1, p^3, .., p^d, b)$, notons p le point l'intersection $p = A(p^1, .., p^d) \cap [a, b]$ où $A(p^1, .., p^d)$ est l'espace affine engendre par les points $p^1, .., p^d$ et notons λ la coordonnée barycentrique telle que $p = \lambda a + (1-\lambda)b$ et les coordonnées barycentriques $\mu_1, .., \mu_d$ tel que $p = \sum \mu_i p^i$ et tel que $\sum \mu_i = 1$.
L'interpolé est convexe si seulement si on a

$$\sum \mu_i ||p^i||^2 \leq ||a||^2 \lambda + ||b||^2 (1-\lambda). \tag{4.10}$$

Pour finir, il suffit de remarque que pour tout d-simplex $(q^0, ..., q^d)$ de l'intersection du graphe de la fonction $f(x) = ||x||^2$ avec l'hyperplan affine de $\mathbb{R}^{d+1}$ passant par $F_f(q^0), ..., F_f(q^d)$ est l'image du la sphere inscrite de $(q^O, ..., q^d)$ par F_f (à demontrer en exercice en utilisant la puissance d'un point par un cercle) et donc on a (4.10) si et seulement si b est dans la sphère inscrit de T_a, ce qui acheve la démonstration. Ceci est equivalent aussi le $d+1$ simplex $(F_f(p^1), ..., F_f(p^d), F_f(a), F_f(b))$ est positif.

■

Nous avons retrouver la formule [George,Borouchaki-1997, equation (1.13)] qui décrit la boule circonscrite au d-simplex positif $(q^0, \ldots, q^d)$. le point b est dans la boule si et seulement si

$$det \begin{vmatrix} q_1^0 & \ldots & q_1^d & b_1 \\ \vdots & \ldots & \vdots & \vdots \\ q_d^0 & \ldots & q_d^d & b_d \\ ||q^0||^2 & \ldots & ||q^0||^2 & ||b||^2 \\ 1 & \ldots & 1 & 1 \end{vmatrix} \geq 0 \tag{4.11}$$

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.

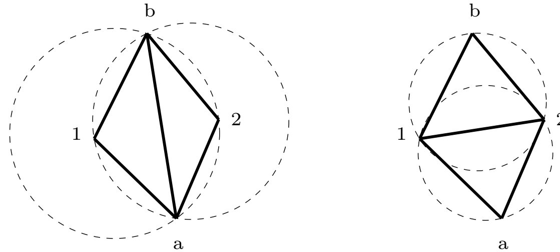


Figure 4.6 – Échange de diagonale d'un quadrilatère convexe selon le critère de la boule vide.

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 4.6.

(x) **Lemme 4.3** *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) :*

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

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

$$aire^{12b} \quad (s^1 - s^a, s^b - s^a) \quad > \quad aire^{1ab} \quad (s^1 - s^2, s^b - s^2). \tag{4.12}$$

∎

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.
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 4.8. Pour éviter ce type de maillage, nous pouvons suivre deux pistes :

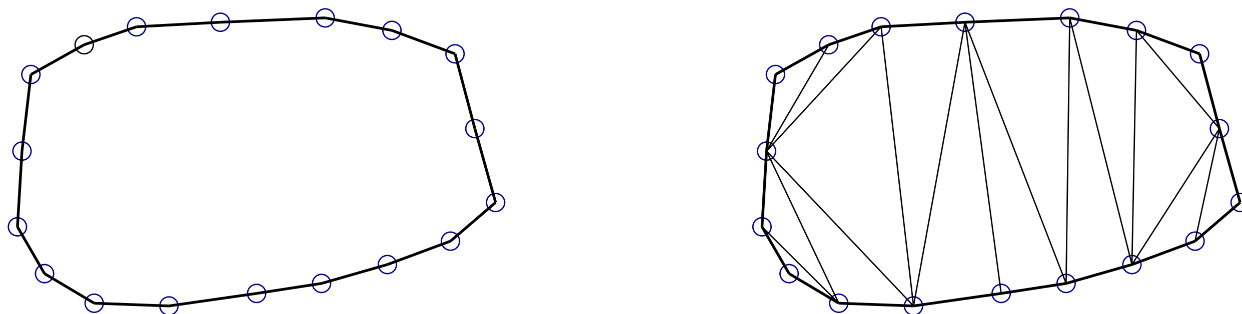


Figure 4.7 – Exemple de maillage d'un polygone sans point interne.

- 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.

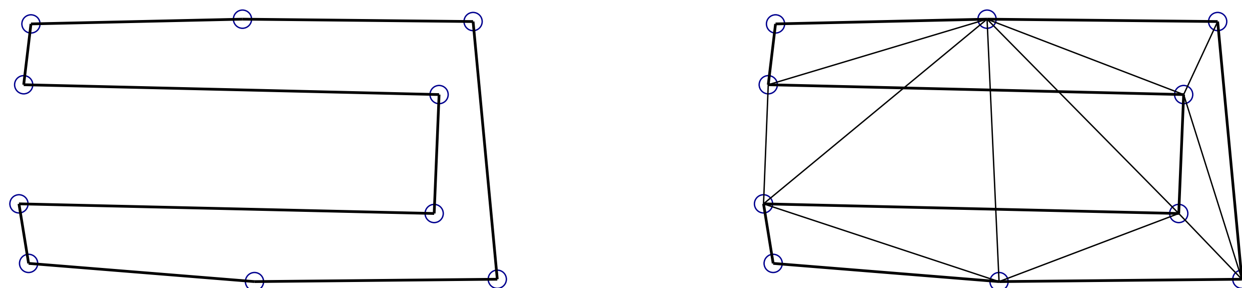


Figure 4.8 – Exemple de maillage de Delaunay ne respectant pas la frontière.

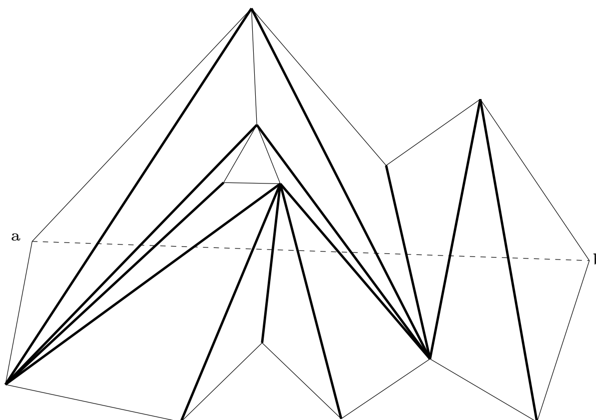


Figure 4.9 – Exemple d'arête $]a, b[$ *manquante dans un maillage compliqué.*

4.1.6 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 4.8 et 4.9).

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

(x) **Théorème 4.4** *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*

$$]a, b[\cap \mathcal{T}_{0,h} = \emptyset \quad \text{et} \quad]a, b[\subset \mathcal{O}_h. \tag{4.13}$$

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

$$[\alpha_i, \beta_j] \stackrel{def}{=} \overline{T}^{i-1} \cap \overline{T}^i, \quad \text{avec} \quad \alpha_i \in P_{ab}^+ \quad \text{et} \quad \beta_i \in P_{ab}^-, \tag{4.14}$$

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 (4.13) 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 (4.13) 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 :

$$]a, \alpha_{i^+}[\subset \overbrace{\bigcup_{i=0}^{i^+-1} \overline{T}^i}^{\circ} \quad \text{et} \quad]\alpha_{i^+}, b[\subset \overbrace{\bigcup_{i=i^+}^{n} \overline{T}^i}^{\circ}. \tag{4.15}$$

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 (4.14). 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 4.6 *On en déduit facilement une autre démonstration du théorème 4.1 : 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 4.4, il découle :

Théorème 4.5 *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 4.1 *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.

4.1.7 Recherche de sous-domaines

L'idée est de repérer les parties qui sont les composantes connexes de $\mathcal{O}_h \setminus \cup_{j=1}^{N_a}[x^{sa^j_1}, x^{sa^j_2}]$, 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 4.2

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 T_{d,h} qui sont dans 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 4.2 Récrire l'algorithme 4.2 sans utiliser la récursivité.

4.1.8 Génération de points internes

Pour compléter l'algorithme de construction du maillage, il faut savoir générer les points internes (voir figure 4.4). 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 :

$$N_1(a,b) \quad \overset{def}{=} \quad |ab|\left(\frac{\int_a^b h(t)dt}{|ab|}\right)^{-1} = \left(\int_a^b h(t)dt\right)^{-1}. \tag{4.16}$$

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

$$l \quad \overset{def}{=} \quad \int_{t=0}^{1} \sqrt{(\gamma'(t), \gamma'(t))}dt, \tag{4.17}$$

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 $h(\mathbf{x})$) :

$$l^h \stackrel{def}{=} \int_{t=0}^{1} \frac{\sqrt{(\gamma'(t),\gamma'(t))}}{h(\gamma(t))} dt \tag{4.18}$$

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

$$N_2(a,b) \stackrel{def}{=} \int_a^b h(t)^{-1} dt. \tag{4.19}$$

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

$$N_1(0,t) = \left(\int_0^t [h_0 + (h_l - h_0)x]dx\right)^{-1} = (h_0 t + \frac{(h_l - h_0)}{2} t^2)^{-1}, \tag{4.20}$$

$$N_2(0,t) = \frac{1}{h_l - h_0} \ln\left(1 + \frac{h_l - h_0}{h_0} t\right). \tag{4.21}$$

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 (4.21) 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 (4.21), ces points seront alors distribués suivant une suite géométrique de raison $\frac{\ln(h_l - h_0)}{h_l - h_0}$.

4.2 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 (4.12). 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 4.1.
6. Retirer les parties externes du domaine en utilisant l'algorithme 4.2.
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.

4.3 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 4.1 *(R2.hpp)*

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

#include <iostream>

namespace bamg {
using namespace std;


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

[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

```
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 4.6) 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 4.2 *(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^o 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 4.3

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 4.3 Prouver que l'algorithme 4.3 précédent marche si le maillage est convexe.

Listing 4.3 *(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 4.4 *(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 4.4

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 4.5

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 (4.2)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 4.6

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 4.7

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 4.8

Écrire le générateur de maillage complet qui lit les données définies en §4.1.3 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.

Index

échange de diagonale, 64

coloriage, 67
complexité de d'algorithme, 93
composante connexe, 68
composantes connexes, 67
convexifié, 55
coordonnées entières, 70
Critère de boule vide, 60

Delaunay-Voronoï, 55
Diagramme de Voronoï, 59

erreurs d'arrondi, 70

forçage d'arête, 67
forçage de la frontière, 65

générateur de maillage, 71

$h()$, 68, 71

longueur Remmanienne, 69

maillage, 55
    frontière, 57
    triangulaire, 56
Maillage de Delaunay, 59

points internes, 68

sous-domaines, 58

taille de la maille, 71
taille de maillage, 68
tar.gz, 7