C++ et éléments finis
Note de cours de DEA (version provisoire)

F. Hecht
Université Pierre et Marie Curie
http://www.ann.jussieu.fr/~
mailto:Frederic.Hecht@Inria.fr

20 décembre 2002

Résumé

Dans ce cours de C++ pour les éléments finis, nous proposons des outils pour programmer la méthode d'éléments finis. La partie syntaxique du `C++` à proprement parler n'est pas décrite. Seuls certains points fins sont discutés. La plus par des constructions syntaxiques sont décrites dans des exemples.
Tous les sources des exemples sont dans le répertoire sous INTERNET. `http://www.ann.jussieu.fr/string~hecht/ftp/cpp`

Table des matières

Chapitre 1

Quelques éléments de syntaxe

Il y a tellement de livres sur la syntaxe du C++ qu'il me paraît déraisonable 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 bien sur d'utiliser le livre The C++ , programming language [2]
Je veux décrire seulement quelques trucs et astuces qui sont généralement utiles comme les déclarations des types de bases et l'algébre 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 réference (je vous rappele 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 déréférence, ou encore donné un autre nom a 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 créér 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 régles de conversion d'un type T en A par défaut qui sont génére à partir d'un constructeur `A(T)` dans la classe A ou avec l'operateur 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 tourver 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, connaitre 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 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 langague 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 de pointeurs (débordement de tableau, destruction multiple, oublie de destruction), return de pointeur sur des variable locales.
Voilà quelques règles a respecté.

Régle 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 oublier ses operateurs, il suffit l'oublié une perluette (&) 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 detruit ici
                                                            //    l'argument est copie puis detruit
 cout << GetOk(a,1) << "\n";                            //    bug on utilise un zone memoire libérée

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

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

Régle 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.

Régle 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: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
#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 régle).
Exemple d'une petite classe qui modelise un tableau d'entier

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

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

Régle 4 *N'utiliser le macro générateur que si vous ne pouver par faire autrement, ou pour ajouter du code de verification ou test qui sera très utile lors de la mise au point.*

Régle 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.

1.3 Verificateur 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 deboguer les oublis de déallocation de pointeurs.

Remarque: 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: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 déboguer favorit (par exemple gdb).
Dans CheckPtr.cpp, il y a une macro du préprocesseur `DEBUGUNALLOC` qu'il faut definir, 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 génére 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. Executez votre programme avec un jeu de donné.
4. Réexecutez votre programme sous le debogueeur et mettez un point d'arrêt dans la fonction `debugunalloc()` (deuxiéme ligne `CheckPtr.cpp` .
5. remontez dans la pile des fonctions appelées pour voir quel pointeur n'est pas désalloué.
6. etc...

Chapitre 2

Le Plan $\mathbb{R}^2$

Mais avant toute chose voilà quelques fonctions de base, qui sont définies pour tous les types (le type de la fonction est un paramètre du patron (**template** en anglais).

```
using namespace std;                                    //    pour utilisation des objet standard
typedef double R;                                                          //    définition de ℝ
//    some usefull function
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 void Exchange (T& a,T& b)       {T c=a;a=b;b=c;}
template<class T>
   inline T Max(const T &a,const T & b,const T & c)
               {return Max(Max(a,b),c);}
template<class T>
  inline T Min(const T &a,const T & b,const T & c)
               {return Min(Min(a,b),c);}
```

Voici une modélisation de $\mathbb{R}^2$ disponible à FTP: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).

```
class R2 {                                                                    //    la classe R2
public:
  R x,y;
  R2 () :x(0),y(0) {};                                              //    constructeur par défaut
  R2 (R a,R b):x(a),y(b)  {}                                          //    constructeur standard
  R2 (R2 a,R2 b):x(b.x-a.x),y(b.y-a.y)  {}                                          //    bipoint
  R2   operator+(R2 P)  const   {return R2(x+P.x,y+P.y);}
  R2   operator+=(R2 P)        {x += P.x;y += P.y;return *this;}
  R2   operator-(R2 P)  const   {return R2(x-P.x,y-P.y);}
  R2   operator-=(R2 P)        {x -= P.x;y -= P.y;return *this;}
  R2   operator-()      const   {return R2(-x,-y);}
  R2   operator+()      const   {return *this;}
  R    operator,(R2 P)  const   {return x*P.x+y*P.y;}                  //    produit scalaire
  R    operator^ (R2 P) const   {return x*P.y-y*P.x;}                       //    determinant
  R2   operator*(R c)   const   {return R2(x*c,y*c);}
  R2   operator/(R c)   const   {return R2(x/c,y/c);}
  R2   perp()          const   {return R2(-y,x);}                  //    la perpendiculaire ⊥
};
 inline R2 operator*(R c,R2 P) {return P*c;}
 inline R Norme(const R2 &A, const R2 &B) {return sqrt((A,B));}
 inline ostream& operator <<(ostream& f, const R2 & P )
       { f << P.x << ' ' << P.y; return f; }
 inline istream& operator >>(istream& f,  R2 & P)
      { f >>  P.x >>  P.y; return f; }
```

Quelques remarques sur la syntaxe :

– Les opérateurs binaires dans une classe n'ont seulement q'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`.

Exercice 1 : Modifier la classe `R2` pour faire une classe `complexe` qui modélise le corps de complexe $\mathbb{C}$. Puis écrire un programme test, qui calcule des racines d'un polynome p avec la méthode de Newton:

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

2.1 La classe R2

Cette classe modélise le plan $\mathbb{R}^2$, de façon que les opérateurs classiques fonctionnent, c'est-à-dire:
Un point P du plan est modélisé par ces 2 coordonnés x, y, 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

cout <<  A;                                     //    imprime sur la console A.x et A.x
cint >> A;                                      //    vous devez entrer 2 double à la console
```

Chapitre 3

Les classes tableaux

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: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.

3.1 Exemple d'utilisation

```
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 × 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;
}
```

3.2 Un 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 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

$\rho = -\dfrac{(G^i, H^i)}{(H^i, AH^i)}$

$x^{i+1} = x^i + \rho H^i$

$G^{i+1} = G^i + \rho AH^i$

$\gamma = \dfrac{(G^{i+1}, G^{i+1})_C}{(G^i, G^i)_C}$

$H^{i+1} = -CG^{i+1} + \gamma H^i$

si $(G^{i+1}, G^{i+1})_C < \varepsilon$ *stop*

Voila comment écrire un gradient conjugué avec ces classes.

3.2.1 Gradient conjugué préconditionné

Listing 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);}
};
```

3.2.2 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 *(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;
}
```

3.2.3 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
```

Chapitre 4

Méthodes d'éléments finis P_1

4.1 Le problème et l'algorithme

Le problème est de résoudre numériquement l'équation de la chaleur dans un domaine Ω de $\mathbb{R}^2$.

$$\Delta u = f, \quad \text{dans } \Omega, \quad u = g \quad \text{sur } \partial\Omega, \quad u = u_0 \text{ au temps } 0 \tag{4.1}$$

Nous utiliserons la discrétisation par des éléments finis P^1 Lagrange construite sur un maillage $\mathcal{T}_h$ de Ω. Notons V_h l'espace des fonctions éléments finis et V_{0h} les fonctions de V_h nulle sur bord de Ω_h (ouvert obtenu comme l'intérieur de l'union des triangles fermés de $\mathcal{T}_h$).

$$V_h = \{v \in \mathcal{C}^0(\Omega_h)/\forall K \in \mathcal{T}_h, v|_K \in P^1(K)\} \tag{4.2}$$

Après utilisation de la formule de Green, et en multipliant par δt, le problème peut alors s'écrire: medskip
Calculer $u_h^{n+1} \in V_h$ à partir de u_h^n , où la donnée initiale u_h^0 est interpolé P^1 de u_0.

$$\int_\Omega \nabla u_h \nabla v_h = \int_\Omega f v_h, \quad \forall v_h \in V_{0h}, \tag{4.3}$$

$$u_h = g \quad \text{sur les sommets de } \mathcal{T}_h \text{ dans } \partial\Omega$$

Nous nous proposons de programmer cette méthode l'algorithme du gradient conjugue pour résoudre le problème linéaire car nous avons pas besoin de connaître explicitement la matrice, mais seulement, le produit matrice vecteur.
On utilisera la formule l'intégration sur un triangle K qui est formé avec les 3 sommets q_K^i, suivante:

$$\int_K f \approx \frac{aire_K}{3} \sum_{i=1}^{3} f(q_K^i) \tag{4.4}$$

Puis, notons, $(w^i)_{i=1,N_s}$ les fonctions de base de V_h associées aux N_s sommets de $\mathcal{T}_h$ de coordonnées $(q^i)_{i=1,N_s}$, tel que $w^i(q_j) = \delta_{ij}$. Notons, U_i^k le vecteur de $\mathbb{R}^{N_s}$ associé à u^k et tel que $u^k = \sum_{i=1,N_s} U_i^k w^i$.
Sur un triangle K formé de sommets i, j, k tournants dans le sens trigonométrique. Notons, H_K^i le vecteur hauteur pointant sur le sommet i du triangle K et de longueur l'inverse de la hauteur, alors on a:

$$\nabla w^i|_K = H_K^i = \frac{(q^j - q^k)^\perp}{2\, aire_K} \tag{4.5}$$

où l'opérateur $\perp$ de $\mathbb{R}^2$ est défini comme la rotation de $\pi/2$, ie. $(a, b)^\perp = (-b, a)$.

Algorithme 2 *Le plus simple des programmes d'éléments finis*

1. *On peut calculer $u_i = \int_\Omega w^i f$ en utilisant la formule 4.4, comme suit:*
   (a) $\forall i = 1, N_s; u_i = 0$;
   (b) $\forall K \in \mathcal{T}_h; \forall i$ *sommet de* $K; u_i = \sigma_i + f(b_K) aire_K/3$; *où* b_K *est le barycentre du triangle* K.

2. *Le produit y matrice A par un vecteur x est dans ce cas s'écrit mathématiquement:*

$$y_i = \begin{cases} \sum_{K\in\mathcal{T}_h} \int_K \nabla w^i|_K \nabla w^j|_K x_j & si\ i\ n'est\ pas\ sur\ le\ bord \\ 0 & si\ i\ est\ sur\ le\ bord \end{cases}$$

Pour finir, on initialiserons le gradient conjugué par:

$$u_i = \begin{cases} 0 & si\ i\ n'est\ pas\ sur\ le\ bord \\ g(q_i) & si\ i\ est\ sur\ le\ bord \end{cases}$$

Les sources et un maillage sont accessibles sur la toile à l'adresse suivante: FTP:sfemGC.tar.gz (version tar compressée). Pour décompresser sous Unix, dans une fenêtre shell entré:

```
tar zxvf sfemGC.tar.gz
```

4.2 Les classes de base pour les éléments finis

Nous allons définir les outils informatiques en `C++` pour programmer l'algorithme 2, pour cela il nous faut modéliser $\mathbb{R}^2$, le maillage formé de triangles qui sont définis par leurs trois sommets. Mais attention, les fonctions de bases sont associées au sommet et donc 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 trouver le numéro d'un sommet, ou du triangle
Donc 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 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.

4.2.1 La classe `Label` (numéros logiques)

Nous avons vu que le moyen le plus simple de distinguer (pour les conditions aux limites) 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 3 *(sfem.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 une étiquette, et tester si elle est nulle (pas de conditions aux limites).

Listing 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
```

4.2.2 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. Par conséquent, la classe `Vertex` va dériver des classes `R2` et `Label` tout en héritant leurs données membres et méthodes (voir le paragraphe **??**) :

Listing 5 *(sfem.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 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: 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. ∎

4.2.3 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 très utiles pour le calcul des intégrales intervenant dans les formulations variationnelles :

– `Edge(i)` qui calcule le vecteur «arête du triangle» opposée au sommet local i ;

– `H(i)` qui calcule directement ∇w^i par la formule (**??**).

Listing 7 *(sfem.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 ∇w^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: 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: 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 8 *(utilisation de la classe Triangle)*

```
                                                //    soit T un triangle de sommets A,B,C ∈ IR^2
                                                //    ----------------------------------
  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)
```

4.2.4 La classe `BoundaryEdge` (modélisation des arêtes frontières)

La classe `BoundaryEdge` va permettre l'accès rapide aux arêtes frontières pour le calcul des intégrales de bord. Techniquement parlant, cette classe reprend les idées développées pour la construction de la classe `Triangle`.
Une arête frontière est définie comme un tableau de deux pointeurs sur des sommets, plus une étiquette (*label*) donnant le numéro de la frontière. La classe contient trois fonctions :

- `set` définit effectivement l'arête ;
- `in` répond si un sommet appartient à l'arête ;
- `length` calcule la longueur de l'arête.

Listing 9 *(sfem.hpp - la classe BoundaryEdge)*

```
 class BoundaryEdge: public Label {
public:
  Vertex *vertices[2];                                          //    tableau de deux Vertex

  void set(Vertex * v0,int i0,int i1,int r)                     //    construction de l'arête
  { vertices[0]=v0+i0; vertices[1]=v0+i1; lab=r; }

  bool in(const Vertex * pv) const
  {return pv == vertices[0] || pv == vertices[1];}

  BoundaryEdge(){};                                          //    constructeur par défaut vide

  Vertex & operator[](int i) const {ASSERTION(i>=0 && i <2);
  return *vertices[i];}

  R length() const { R2 AB(*vertices[0],*vertices[1]);return sqrt((AB,AB));}

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

Remarque: Les remarques 4.2.3 et 4.2.3 restent également valables pour la classe `BoundaryEdge`. ∎

La classe `BoundaryEdge` permet les opérations suivantes :

Listing 10 *(utilisation de la classe BoundaryEdge)*

```
                                                        //    soit E une arête de sommets A,B
                                                    //    ----------------------------------
  const Vertex & V =  E[i];                                 //    le sommet i de E (i=0 ou 1)
  double a = E.lenght();                                          //    longueur de E
  (Label) E  ;                                                  //    l'étiquette de E

  BoundaryEdge E;
  E.set(v,ia,ib,lab);                                 //    construction d'une arête avec les sommets
                                                        //    v[ia],v[ib] et étiquette lab
                                                      //    (v est le tableau des sommets)
```

4.2.5 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 11 *(sfem.hpp - la classe Mesh)*

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

  Vertex *vertices;
  Triangle *triangles;
  BoundaryEdge  *bedges;

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

  inline Mesh(const char * filename);          //    constructeur (lecture du fichier ".msh")

  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 étiquettes 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 et dont le format a été décrit dans le paragraphe **??** :

Listing 12 *(sfem.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];                    //   allocation mémoire - tab des triangles
  vertices = new Vertex[nv];                          //   allocation mémoire - tab des sommets
  bedges = new BoundaryEdge[neb];             //   allocation mémoire - tab des arêtes frontières

  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 de chaque 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 13 *(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

  int ie = ...;
  BoundaryEdge & be= Th.begdes[ie];                     //    référence de l'arête frontière ie
  int ie1= Th(be[1]);                             //    numéro du sommet 1 de l'arête frontière be
  be.lab;                                                       //    label de l'arête frontière be
```

4.2.6 Le programme principale

Listing 14 *(sfemGC.cpp)*

```
#include <cassert>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include "sfem.hpp"
#include "RNM.hpp"
#include "GC.hpp"

R f(const R2 & ){return 1;}                            //    right hand side
R g(const R2 & ){return 0;}                            //    boundary condition
R u0(const R2 & ){return 0;}                           //    initialization

class Laplacien2d: VirtualMatrice<R> { public:
 const  Mesh & Th;
 typedef  VirtualMatrice<R>::plusAx plusAx;
 Laplacien2d(const  Mesh & T) : Th(T) {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {
                                                                  //
```

$$Ax_i += \sum_{K\in\mathcal{T}_h} \int_K (\nabla w^i|_K, \nabla w^j|_K)\, x_j \quad \textit{si } i \textit{ n'est pas sur le bord}$$

```
   for (int k=0;k<Th.nt;k++)
     {
      const Triangle & K(Th[k]);
      int i0(Th(K[0])),i1(Th(K[1])),i2(Th(K[2]));         //    n0 globaux des 3 sommets
      R2 H0(K.H(0)),H1(K.H(1)),H2(K.H(2));
      R2 gradx= H0*x[i0] + H1*x[i1] + H2*x[i2];
      if ( ! K[0].onGamma() ) Ax[i0] +=  (gradx,H0)*K.area;
      if ( ! K[1].onGamma() ) Ax[i1] +=  (gradx,H1)*K.area;
      if ( ! K[2].onGamma() ) Ax[i2] +=  (gradx,H2)*K.area;
     }
  }
 plusAx operator*(const KN<R> &  x) const {return plusAx(this,x);}
};

void InternalError(const char * str) {cerr << str; exit(1);}


int main(int , char** )
{
 Mesh Th("c30.msh");

 KN<R> b(Th.nv);                                          //    b[i] = ∫_Ω f w_i
 b=0;
 for (int k=0;k<Th.nt;k++)
        {
          Triangle & K(Th[k]);
          R2 A(K[0]),B(K[1]),C(K[2]),M( (A+B+C)/3.);
          R intKfwi = K.area*f(M)/3.;
          if(! K[0].onGamma()) b[Th(K[0])] += intKfwi;
          if(! K[1].onGamma()) b[Th(K[1])] += intKfwi;
          if(! K[2].onGamma()) b[Th(K[2])] += intKfwi;
        }

 KN<R> x(Th.nv);
 x=0;                              //    donne initial avec les conditions aux limites.
 for (int k=0;k<Th.nt;k++)
        {
           Triangle & K(Th[k]);
           for (int i=0;i<3;i++)
             if(K[i].onGamma())  x[Th(K[i])] = g(K[i]);
             else                   x[Th(K[i])] = u0(K[i]);
    }
 GradienConjugue(Laplacien2d(Th),MatriceIdentite<R>(), b,x,Th.nv,1e-10);
```

```
{                                                                    //    a file for gnuplot
  ofstream plot("plot");
  for(int it=0;it<Th.nt;it++)
    plot << (R2) Th[it][0] << " " << x[Th(it,0)] << endl
         << (R2) Th[it][1] << " " << x[Th(it,1)] << endl
         << (R2) Th[it][2] << " " << x[Th(it,2)] << endl
         << (R2) Th[it][0] << " " << x[Th(it,0)] << endl << endl << endl;
}
return 0;}
```

4.2.7 Execution

Pour compiler et executer le programe, il faut entrer les lignes suivantes dans un fenêtre `Shell Unix`:

```
#  la compilation et édition de lien
g++ sfemGC.cpp -o sfemGC
#  execution
./sfemGC
Read On file "c30.msh"
Nb of Vertex 109  Nb of Triangles 186
End of read: area = 12.4747
34  ro = 0.315038 ||g||^2 = 3.15864e-21
#  visualisation du resultat
gnuplot
splot "plot" w l
quit
```

Chapitre 5

Modélisation des vecteurs, matrices, tenseurs

Dans ce chapitre, nous allons nous intéresser à la construction de classes pour modéliser des tableaux à 1,2 ou 3 indices, soit $\mathbb{K}^n$, $K^{n,m}$, et $\mathbb{K}^{n,m,k}$ où $\mathbb{K}^{n,m}$ est l'espace des matrices $n \times m$ à coefficient dans $\mathbb{K}$ et ou $\mathbb{K}^{n,m,k}$ est l'espace des tenseurs d'ordre 3 à coefficient dans $\mathbb{K}$.
Donc nous allons pleinement utiliser les patrons « template »pour construire ces trois classes.
Nous commencerons sur une version didactique, nous verrons la version complète qui est dans le fichier « tar compressé » FTP:RNM.tar.gz.

5.1 Version simple

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 $+, -, *, ...$

```
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 functions 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: 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 lÕallocateur `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.

5.2 Les classes tableaux

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, ...).
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.

```
tyepdef 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.

5.3 Exemple d'utilisation

```
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 × 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++)                                       //    colonne
  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;
}
```

5.4 Un 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 :

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

où $(.,.)$ est le produit scalaire classique de $\mathbb{R}^n$.

Algorithme 3 *Le gradient conjugué:*
soit $x \in \mathbb{R}^n$ *donné.*
- $g^0 = Ax - b$
- $h^0 = -g^0$
- *pour* $i = 0$ *à* n
  - $\rho = -\dfrac{(g^i, h^i)}{(h, Ah)}$
  - $x^{i+1} = x^i + \rho h^i$
  - $g^{i+1} = g^i + \rho A h^i$
  - $\gamma = \dfrac{(g^{i+1}, g^{i+1})}{(g^i, g^i)}$
  - $h^{i+1} = -g^{i+1} + gamma * h^i$
    si $(g^{i+1}, g^{i+1}) < \varepsilon$ *stop*

Voila comment écrire un gradient conjugue avec ces classes.

5.4.1 Gradient conjugué préconditionné

Listing 15 *(RNM/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);
   KN<R> cg(n);
   KN<R> h(n);
   KN<R> Ah(n); KN<R> & 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);
      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;
}

template<class R>
class MatriceIdentite: VirtualMatrice<R> { public:
 MatriceIdentite() {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {   Ax+=x; }
  VirtualMatrice<R>::plusAx operator*(const KN<R> &  x) const
    {return plusAx(*this,x);}
};
```

5.4.2 Teste 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 16 *(RNM/GradConjugue.cpp)*

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

 using namespace std;

#include <time.h>
inline double CPUtime(){
#ifdef SYSTIMES
  struct tms buf;
  if (times(&buf)!=-1)
    return ((double)buf.tms_utime+(double)buf.tms_stime)/(long) sysconf(_SC_CLK_TCK);
  else
#endif
    return ((double) clock())/CLOCKS_PER_SEC;
}

                                                                    //   #include "sfem.hpp"
#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;
}
```

5.4.3 Sortie du teste

```
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 : Utiliser les classes de `RNM.hpp` dans la programmation de `sfem.cpp`.

Exercice 3 : Ecrire la résolution par l'élément fini P_1 en utilisant le gradient conjuge préconditioné du problème suivant:

$$-\Delta u = 1, \quad \text{dans } \Omega, \quad u = 0 \quad \text{sur } \partial\Omega, \quad u = u_0 \text{ au temps } 0 \tag{5.1}$$

La solution de se problème est la limite quand $t \to \infty$ du problème résolue dans le chapitre 4.
Donc le produit y de matrice par un vecteur x est dans ce cas s'écrit mathématiquement

$$y_i = \begin{cases} \sum_{K \in \mathcal{T}_h} \nabla w^i|_K \nabla w^j|_K x_j & \text{si } i \text{ n'est pas sur le bord} \\ 0 & \text{si } i \text{ est sur le bord} \end{cases}$$

Chapitre 6

Chaînes et Chaînages

6.1 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.

6.2 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 4 *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 4 : Le pourquoi est laissé en exercice.

6.3 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` (voir **??**). Dans certaines applications, il peut être utile de construire la liste des *arêtes du maillage*, c'est-à-dire l'ensemble des arêtes de tous les éléments. La difficulté dans ce type de construction réside dans la manière d'éviter – ou d'éliminer – les doublons (le plus souvent une arête appartient à deux triangles).
Nous allons proposer deux algorithmes pour déterminer la liste des arêtes. Dans les deux cas, nous utiliserons le fait que les arêtes sont des segments de droite et sont donc définies complètement par la donnée des numéros de leurs deux sommets. On stockera donc les arêtes dans un tableau `arete[nbex][2]` où `nbex` est un majorant du nombre total d'arêtes. On pourra prendre grossièrement `nbex = 3*nt` ou bien utiliser la formule d'Euler en 2D

$$\texttt{nbe} = \texttt{nt} + \texttt{nv} + \texttt{nb\_de\_trous} - \texttt{nb\_composantes\_connexes}, \tag{6.1}$$

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 5 *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++)                                 //    on parcourt les arêtes déjà construites
        if (arete[e][0] == i && arete[e][1] == j)
          {existe=true;break;}                                 //    l'arête est déjà construite stop
      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 $3 \times nbs$.
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 4 seront simultanées.

Algorithme 6 *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 5 : 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]`.

6.4 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 **??**), `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 4, 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:

$$\mathtt{p} \longrightarrow (\mathtt{k}, \mathtt{j}) = (\mathtt{k} = \mathtt{p}/\mathtt{m}, \mathtt{j} = \mathtt{p}\%\mathtt{m}). \tag{6.2}$$

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

[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`.

Algorithme 7 *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],k=p/3,j = p % 3)
  { assert( i == Th(k,j));
                                                               //    votre code
  }
```

Exercice 6 : Optimiser le code en initialisant `p =-1` et en remplaçant `p = 3*j+k` par `p++`.

6.5 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.

6.5.1 Description de la structure morse

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

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

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

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

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

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

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

```
n=10,nbcoef=20,
ligne[-1:9] = {-1,0,1,3,6,8,11,14,17,19,20};
colonne[21] ={0, 1,  1,2,  1,2,3,  2,4,  3,4,5,  2,4,6,
              5,6,7,  4,8,  9};
a[21]                                          //    ... valeurs des 21 coefficients de la matrice
```

6.5.2 Construction de la structure morse par coloriage

Nous allons maintenant construire la structure morse d'une matrice symétrique à partir de la donnée d'un maillage d'éléments finis P^1. Pour construire la ligne i de la matrice, il faut trouver tous les sommets j tels que i, j appartiennent à un même triangle. Ainsi, pour un noeud donné i, il s'agit de lister les sommets appartenant aux triangles contenant i. Le premier ingrédient de la méthode sera donc d'utiliser l'algorithme 7 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 8 *Construction de la structure d'une matrice morse*

```
MatriceMorseSymetrique::MatriceMorseSymetrique(const Mesh & Th){
  int color=0, * mark;
  int i,j,jt,k,p,t;
  n = Th.nv;
  mark = new int [n];
                                        //    construction optimisee de l'image réciproque de Th(k,j)
  int end_list=-1,*head_s,*next_t;
  head_s = new int [Th.nv];
  next_p = new int [Th.nt*3];
  int i,j,k,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
  nbcoef = 0;
  for(i=0; i<n; i++,color++,nbcoef++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
      for(jt=0; jt< 3; jt++ )
       if ( i <= (j=Th(t,jt))  && mark[j]!= color))
        mark[j]=color,nbcoef++;                       //    nouveau coefficient => marquage + ajout
```

```
                                                                    //    2) allocations mémoires
  ligne = new int [n+1];
  ligne++;                                                    //    car le tableau commence en -1;
  colonne = new int [ nbcoef];
  a = new double [nbcoef];
                                          //    3) constructions des deux tableaux ligne et colonne
  ligne[-1] = -1;
  nbcoef =0;
  for(i=0; i<n; ligne[i++]=nbcoef, color++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
       for(jt=0; jt< 3; jt++ )
        if ( i <= (j=Th(t,jt)))  && mark[j]!= color))
         mark[j]=color, colonne[nbcoef++]=j;        //    nouveau coefficient => marquage + ajout
                                                     //    4) tri des lignes par index de colonne
  for(i=0; i<n; i++)
    HeapSort(colonne + ligne[i-1] + 1 ,ligne[i] - ligne[i-1]);
                                                                                    //    nettoyage
  delete  [] head_s;
  delete  [] next_p;
}
```

Au passage, nous avons utilisé la fonction `HeapSort` qui implémente un petit algorithme de tri, présenté dans [1], qui a la propriété d'être toujours en $nlog_2 n$ (*cf.* code ci-dessous). Noter que l'étape de tri n'est pas absolument nécessaire, mais le fait d'avoir des lignes triées par indice de colonne permet d'optimiser l'accés à un coefficient de la matrice dans la structure creuse.

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

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

Chapitre 7

Algèbre de fonctions

Le but de cette section est de montrer comment il est possible en C++ de construire et manipuler des fonctions.

7.1 Version de base

Une fonction est modélisée par une classe (Cvirt) qui a juste l'opérateur « fonction »() virtuelle. Donc une fonction sera définie comme une classe C qui contient juste un pointeur une classe Cvirt.
On dérive cette classe Cvirt pour créer des types de fonctions qui sont donc : un classe Cfonc qui contient un pointeur sur une fonction, et l'opérateur « fonction », et une classe pour construire les opérateur binaire classique qui contient 2 deux pointeurs sur des Cvirt qui correspondent au membre de droite et gauche de l'opérateur et dÕune fonction de $\mathbb{R}^2$ à valeur de $\mathbb{R}$ pour définir l'opérateur.
Les programmes sources sont accessible :
FTP:lg/fonctionsimple.cpp.

```
#include <iostream>
                                                    //    pour la fonction pow
#include <math.h>
 typedef double R;
 class Cvirt { public: virtual R operator()(R ) const =0;};

 class Cfonc : public Cvirt { public:
   R (*f)(R);
   R operator()(R x) const { return (*f)(x);}
   Cfonc( R (*ff)(R)) : f(ff) {} };

 class Cconst : public Cvirt { public:
   R a;
   R operator()(R ) const { return a;}
   Cconst( R aa) : a(aa) {} };

 class Coper : public Cvirt { public:
   const  Cvirt *g, *d;
   R (*op)(R,R);
   R operator()(R x) const { return (*op)((*g)(x),(*d)(x));}
   Coper( R (*opp)(R,R), const Cvirt *gg, const Cvirt *dd) : op(opp),g(gg),d(dd) {}
   ~Coper(){delete g,delete d;} };

                                                    //    les 5 opérateur binaire
 static  R Add(R a,R b) {return a+b;}
 static  R Sub(R a,R b) {return a-b;}
 static  R Mul(R a,R b) {return a*b;}
 static  R Div(R a,R b) {return a/b;}
 static  R Pow(R a,R b) {return pow(a,b);}

 class Fonction { public:
   const Cvirt *f;
   R operator()(const R x){ return (*f)(x);}
```

```
   Fonction(const Cvirt *ff) : f(ff) {}
   Fonction(R (*ff)(R)) : f(new Cfonc(ff)) {}
   Fonction(R a) : f(new Cconst(a)) {}
   operator const  Cvirt * () const {return f;}
   Fonction operator+(const Fonction & dd) const {return new Coper(Add,f,dd.f);}
   Fonction operator-(const Fonction & dd) const {return new Coper(Sub,f,dd.f);}
   Fonction operator*(const Fonction & dd) const {return new Coper(Mul,f,dd.f);}
   Fonction operator/(const Fonction & dd) const {return new Coper(Div,f,dd.f);}
   Fonction operator^(const Fonction & dd) const {return new Coper(Pow,f,dd.f);}
 };

using namespace std;                                    //   introduces namespace std

R Identite(R x){ return x;}
int main()
{
  Fonction Cos(cos),Sin(sin),x(Identite);
  Fonction f((Cos^2)+Sin*Sin+(x*4));                    //    attention ^ n'est prioritaire
  cout << f(2) << endl;
  cout << (Cos^2)+Sin*Sin+(x*4))(1) << endl;
  return 0;
}
```

Dans cet exemple, la fonction $f = cos^2 + sin * sin + x^4$ sera défini par un arbre de classe que l'on peut representer par

```
f.f= Coper (Add,t1,t2);                 t1=(cos^2)          +    // t2=(sin*sin + x^4)
    t1.f= Coper (Pow,t3,t4);            // t3=(cos)          ^    t4=(2)
    t2.f= Coper (Add,t5,t6);            // t5=(sin*sin)      +    t6=x^4
        t3.f= Ffonc (cos);
        t4.f= Fconst (2.0);
        t5.f= Coper (Mul,t7,t8);        // t7=(sin)          *    t8=(sin)
            t6.f= Coper (Pow,t9,t10);   // t9=(x)            ^    t10=(4)
                t7.f= Ffonc (sin);
                t8.f= Ffonc (sin);
                t9.f= Ffonc (Identite);
                t10.f= Fconst (4.0);
```

7.2 Les fonctions C^∞

Cette algèbre de fonctions réelles est bases sur les classes virtuelles.
Maintenant nous voulons aussi pouvoir dériver les fonctions. Il faut faire attention car si la classe contient la dérive de la fonction, implicitement il va y avoir une récursivité infinie car les fonctions sont indéfiniment dérivables. Donc pour que cela marche il faut un processus à deux niveaux : l'un qui peut construire la fonction dérive, et l'autre qui évalue la fonction dérive. Donc la classe `CVirt` contiendra deux fonctions virtuelle `virtual float operator () (float) = 0;` (la fonction) et `virtual CVirt *de () return zero;` (la fonction qui calcul la fonction dérivée qui retourne la fonction nulle par défaut). Bien etendu nous stockerons la fonction dérive (`CVirt`) qui ne sera construit que si l'on utilise la dérive de la fonction.
Et introduisons aussi les fonctions de $\mathbb{R}^2$ à valeur dans $\mathbb{R}$, que l'on appellera `Fonction2`.
Dans les 2 URL
– FTP:lg/fonction.cpp
– FTP:lg/fonction.hpp
Vous trouverez les programmes sources.

```
#ifndef __FONCTION__
#define __FONCTION__

struct CVirt {
  mutable CVirt *md;                                //    le pointeur sur la fonction dérivée
  CVirt () : md (0) {}
```

```
  virtual R operator () (R) = 0;
  virtual CVirt *de () {return zero;}
  CVirt *d () {if (md == 0) md = de (); return md;}
  static CVirt *zero;
};

class Fonction {                                          //    Fonction d'une variable
  CVirt *p;
public:
  operator CVirt *() const {return p;}
  Fonction (CVirt *pp) : p(pp) {}
  Fonction (R (*) (R));                                //    Creation a partir d'une fonction C
  Fonction (R x);                                             //    Fonction constante
  Fonction (const Fonction& f) : p(f.p) {}                      //    Copy constructor
  Fonction d () {return p->d ();}                                     //    Derivee
  void setd (Fonction f) {p->md = f;}
  R operator() (R x) {return (*p)(x);}                        //    Valeur en un point
  Fonction operator() (Fonction);                          //    Composition de fonctions
  friend class Fonction2;
  Fonction2 operator() (Fonction2);
  static Fonction monome (R, int);
};

struct CVirt2 {
  CVirt2 ():  md1 (0), md2 (0) {}
  virtual R operator () (R, R) = 0;
  virtual CVirt2 *de1 () {return zero2;}
  virtual CVirt2 *de2 () {return zero2;}
  CVirt2 *md1, *md2;
  CVirt2 *d1 () {if (md1 == 0) md1 = de1 (); return md1;}
  CVirt2 *d2 () {if (md2 == 0) md2 = de2 (); return md2;}
  static CVirt2 *zero2;
};

class Fonction2 {                                        //    Fonction de deux variables
  CVirt2 *p;
public:
  operator CVirt2 *() const {return p;}
  Fonction2 (CVirt2 *pp) : p(pp) {}
  Fonction2 (R (*) (R, R));                            //    Creation a partir d'une fonction C
  Fonction2 (R x);                                            //    Fonction constante
  Fonction2 (const Fonction2& f) : p(f.p) {}                    //    Copy constructor
  Fonction2 d1 () {return p->d1 ();}
  Fonction2 d2 () {return p->d2 ();}
  void setd (Fonction2 f1, Fonction2 f2) {p->md1 = f1; p->md2 = f2;}
  R operator() (R x, R y) {return (*p)(x, y);}
  friend class Fonction;
  Fonction operator() (Fonction, Fonction);                //    Composition de fonctions
  Fonction2 operator() (Fonction2, Fonction2);
  static Fonction2 monome (R, int, int);
};

extern Fonction Chs,Identity;
extern Fonction2 Add, Sub, Mul, Div, Abscisse, Ordonnee;

inline Fonction operator+ (Fonction f, Fonction g) {return Add(f, g);}
inline Fonction operator- (Fonction f, Fonction g) {return Sub(f, g);}
inline Fonction operator* (Fonction f, Fonction g) {return Mul(f, g);}
inline Fonction operator/ (Fonction f, Fonction g) {return Div(f, g);}
inline Fonction operator- (Fonction f) {return Chs(f);}

inline Fonction2 operator+ (Fonction2 f, Fonction2 g) {return Add(f, g);}
inline Fonction2 operator- (Fonction2, Fonction2 g) {return Sub(f, g);}
inline Fonction2 operator* (Fonction2 f, Fonction2 g) {return Mul(f, g);}
inline Fonction2 operator/ (Fonction2 f, Fonction2 g) {return Div(f, g);}
inline Fonction2 operator- (Fonction2 f) {return Chs(f);}

#endif
```

Chapitre 8

La méthode de éléments finis

Dans ce chapitre nous présentations de classe suffisamment générale pour pouvoir programmer tous les éléments finis classiques.
les sources sont sur la toile à l'adresse suivante FTP:fem.tar.gz pour une version tar compressée.

8.1 Présentation des éléments finis classiques

A faire

8.2 les classes du maillage

Pour cela nous allons nous servir des classes définies dans le chapitre 4 et que nous allons modifier.

8.2.1 Nouvelle classe triangles

Dans cette classe nous avons ajouté juste un membre qui une fonction `renum(Vertex *v0, long * r)` qui renumérote les sommets d'un triangle, qui dessine un triangle, et qui la transformation de $\hat{T}$ dans T du point $\hat{P}$ en $P = T(\hat{P})$.

```
class Triangle: public Label {
  Vertex *vertices[3];                                    //    an array of 3 pointer to vertex
public:
  R area;
  Triangle(){};                                              //    constructor empty for array
  Vertex & operator[](int i) const                  //    to see triangle as a array of vertex
       {return *vertices[i];}

  Vertex *& operator()(int i)                       //    to see triangle as a array of vertex
       {return vertices[i];}

  Triangle(Vertex * v0,int i0,int i1,int i2,int r,R a=0.0): Label(r)
     { R2 A=*(vertices[0]=v0+i0);
       R2 B=*(vertices[1]=v0+i1);
       R2 C=*(vertices[2]=v0+i2);
       area = a ==0? (( B-A)^(C-A))*0.5 : a;
       throwassert(area>=0);}

  R2 Edge(int i) const                                          //    opposite edge vertex i
     {return (R2) *vertices[(i+2)%3]-(R2) *vertices[(i+1)%3];}

  Vertex & Edge(int j,int i) const                              //    Vertex j of edge i
     {throwassert(j==0 || j==1 );return  *vertices[(i+j+1)%3];}

  R2 n(int i) const                                            //    unit exterior normal
     {R2 E=Edge(i);return R2(E.y,-E.x)/Norme2(E);}

  R2 H(int i)  const                                                 //    heigth (∇λ_i
     {R2 E=Edge(i);return R2(-E.y,E.x)/(2*area);}
```

```
  R lenEdge(int i) const {R2 E=Edge(i);return sqrt((E,E));}
  R h() const { return Max(lenEdge(0),lenEdge(1),lenEdge(2));}

  void Renum(Vertex   *v0, long * r)  {
    for (int i=0;i<3;i++)
      vertices[i]=v0+r[vertices[i]-v0];}

  Vertex & VerticeOfEdge(int i,int j) const                          //    vertex j of edge i
    {return  *vertices[(i+1+j)%3];}                                  //    vertex j of edge i

    R EdgeOrientation(int i) const {                                    //    return +1 or -1
     R Orient[2]={-1.,1.};
    return  Orient[vertices[ (i+1)%3] < vertices[ (i+2)%3] ];}

  bool intersect(R2 P,R2 Q) const
   {
     const R2 &A(*vertices[0]);
     const R2 &B(*vertices[1]);
     const R2 &C(*vertices[2]);
     R2 mn(Min(A.x,B.x,C.x),Min(A.y,B.y,C.y)),
        mx(Max(A.x,B.x,C.x),Max(A.y,B.y,C.y));
     assert(P.x < Q.x && P.y < Q.y );
     return (mx.x >= P.x) && (mn.x <= Q.x) &&  (mx.y >= P.y) && (mn.y <= Q.y) ;

   }

          //    const Vertex & VerticeOfEdge( int i, int j) const  return *vertices[(i+1+j)%3];
  void Draw(double shink=1) const;
  void Fill(int color) const;
  void Draw(int edge,double shink=1) const;
  void SetVertex(int j,Vertex *v){vertices[j]=v;}
  R2 operator()  (const R2 & P) const{                    //    local to Global in triangle
     return    (const R2 &) *vertices[0] * (1-P.x-P.y)
           +  (const R2 &) *vertices[1] * (P.x)
           +  (const R2 &) *vertices[2] * (P.y);}

};
```

8.2.2 Classe arête frontiere

Pour prendre en compte les conditions de type Neumann, et pour pouvoir faire des intégrales sur les bords du domaine.

```
class BoundaryEdge: public Label {
public:
  Vertex *vertices[2];
  BoundaryEdge(Vertex * v0,int i0,int i1,int r): Label(r)
  { vertices[0]=v0+i0; vertices[1]=v0+i1; }
  bool in(const Vertex * pv) const {return pv == vertices[0] || pv == vertices[1];}
  BoundaryEdge(){};                                          //    constructor empty for array
  void Draw() const;
  Vertex & operator[](int i) const {return *vertices[i];}
  R length() const { return Norme2(R2(*vertices[0],*vertices[1]));}
  void Renum(Vertex   *v0, long * r) {
    for (int i=0;i<2;i++)
      vertices[i]=v0+r[vertices[i]-v0];}
};
```

8.2.3 Compteur de référence

Sur un maillage, il est possible de définir plusieurs espaces d'éléments finis, donc pour eviter de detruire un maillage, alors qu'il est encore utiliser, il est judicieux d'introduire un compteur de référence, le maillage ne sera effectivement détruit qui si ce compteur est nulle (plus objet utilisant ce maillage).

Le compteur est stocké dans la classe `RefCounter`, et la classe maillage dérivera de cette classe. Soit la classe `C` qui utilise une reference, si cette classe a un membre de type `TheCounter` initialiser avec maillage. Quand nous detruisons cette classe `C` le membre `TheCounter` est détruit et le compteur est décrementé automatiquement, et le maillage est détruit compteur passe à 0.
remarque: Si vous construisez un pointeur sur un maillage via l'operateur **new**, il ne faut pas detruire le pointeur sur le maillage avec l'operateur **delete** car dans tous les cas vous détruirez le maillage, il faut détruire un pointeur sur un maillage via la fonction `destroy()` qui détruit le tout si nécessaire.

```
class RefCounter {
  friend class TheCounter;
  mutable int count;
  protected:
  void destroy() const { assert(count>=0);
                         if ( count==0) delete this; else count--; }
  virtual ~RefCounter() {assert(count==0);}
  RefCounter() : count(0) {}
};

class TheCounter {
 const RefCounter * c;
 public:
 TheCounter() : c(0) {}
 TheCounter(const RefCounter * a) :c(a) { if(c) c->count++;}
 TheCounter(const RefCounter & a) :c(&a) { if(c) c->count++;}
 ~TheCounter(){ if(c) c->destroy();}
};
```

8.2.4 nouvelle classe Maillage

Dans cette classe il y a beaucoup de modification. A compléter ....

```
class FQuadTree;
class Mesh: public RefCounter { public:
   static const char magicmesh[8] ;
  int nt,nv,neb;
  R area;
  static int kthrough,kfind;
  FQuadTree *quadtree;
  Vertex *vertices;
  Triangle *triangles;
  BoundaryEdge  *bedges;
  int NbMortars,NbMortarsPaper;
  Mortar *mortars;                                                    //    list of mortar
  int    *datamortars;
  R2  * bnormalv;                                                  //    boundary vertex normal
                                                                      //   Triangle * adj;
  Triangle & operator[](int i) const {throwassert(i>=0 && i<nt);return triangles[i];}
                            //    const Triangle & operator[]( int i) const  return triangles[i];
  Vertex & operator()(int i) const {throwassert(i>=0 && i<nv);return vertices[i];}
  Mesh(const char * filename) {read(filename);}                       //    read on a file
  Mesh(const string s) {read(s.c_str());}
  Mesh( const Serialize & );
  Serialize serialize() const;
  Mesh(int nbv,int nbt,int nbeb,Vertex *v,Triangle *t,BoundaryEdge  *b);
  Mesh(const Mesh & Thold,int *split,bool WithMortar=true,int label=1);
  ~Mesh();
  int number(const Triangle & t) const {return &t - triangles;}
  int number(const Triangle * t) const {return t  - triangles;}
  int number(const Vertex & v)   const {return &v - vertices;}
  int number(const Vertex * v)   const {return v  - vertices;}
  int operator()(const Triangle & t) const {return &t - triangles;}
  int operator()(const Triangle * t) const {return t  - triangles;}
  int operator()(const Vertex & v)   const {return &v - vertices;}
  int operator()(const Vertex * v)   const {return v  - vertices;}
  int operator()(int it,int j) const {return number(triangles[it][j]);}
```

```
                                                    //    Nu vertex j of triangle it
  void BoundingBox(R2 & Pmin,R2 &Pmax) const;
  void InitDraw() const;
  void Draw(int init=2,bool fill=false) const;
  void DrawBoundary() const;
  int Contening(const Vertex * v) const{ return TriangleConteningVertex[ v  - vertices];}
  int renum();
  int gibbsv (long* ptvoi,long* vois,long* lvois,long* w,long* v);
  int TriangleAdj(int it,int &j) const
     {int i=TheAdjacencesLink[3*it+j];j=i%3;return i/3;}
  void VerticesNumberOfEdge(const Triangle & T,int j,int & j0,int & j1) const
     {j0 =  number(T[(j+1)%3]),j1=  number(T[(j+ 2)%3]);}

  int BoundaryTriangle(int be,int & edgeInT) const {
     int i= BoundaryEdgeHeadLink[be]; edgeInT = i%3;
     return i/3;}

  Triangle * Find(const R2 & P) const;
  const Triangle * Find(R2 P, R2 & Phat,bool & outside,const Triangle * tstart=0) const ;

  BoundaryEdge * TheBoundaryEdge(int i,int j)  const
   {
    int p2;
    for (int p=BoundaryAdjacencesHead[i];p>=0;p=BoundaryAdjacencesLink[p])
     {
     if ( bedges[p2=p/2].in(vertices+j) )   return bedges+p2;
     }

    return 0;}
 void destroy() {RefCounter::destroy();}
 void MakeQuadTree();
private:
  void read(const char * filename);
                                                    //    to construct the adj triangle
  int *TheAdjacencesLink;
  int *BoundaryEdgeHeadLink;
  void ConsAdjacence();
  void Buildbnormalv();
  int *BoundaryAdjacencesHead;
  int *BoundaryAdjacencesLink;
  int *TriangleConteningVertex;
};


                                                  //    2 routines to compute the caracteristic
int WalkInTriangle(const Mesh & Th,int it, double *lambda,
                   const  KN_<R> & U,const  KN_<R> & V, R & dt);
int  WalkInTriangle(const Mesh & Th,int it, double *lambda,
                    R u, R v, R & dt);

int Walk(const Mesh & Th,int & it, R *l,
        const KN_<R>  & U,const KN_<R>  & V, R dt);

void DrawMark(R2 P,R k=0.02);
```

8.3 Formule d'intégration

Voilà une classe qui défini une formule d'intégration comme un tableau de point d'intégration plus l'ordre de la formule et le degré d'exactitude et le type de support triangle ou quadrangle pour pouvoir faire quelques vérifications.
De plus les formules classiques sur des triangles et ou des quadrangles sont définis pour de polynômes. Le paramètre est le dégrée exactitude. Pour plus de détail, les définitions complètes de ces formules sont données dans le fichier `QuadratureFormular.cpp`.

```
class QuadratureFormular;
```

```
class QuadraturePoint : public R2{
  friend ostream& operator <<(ostream& , QuadraturePoint & );
public:
const R a;
                                                                    //    const R2 c;
  QuadraturePoint(R aa,R2 xx): a(aa),R2(xx) {}
  QuadraturePoint(R aa,R xx,R yy): a(aa),R2(xx,yy) {}
  operator R() const {return a;}
                                                          //    operator R2() const  return c;
};


class QuadratureFormular {
 friend  ostream& operator <<(ostream& , const QuadratureFormular & );

public:
  const int n;                                              //    nombre de point d'integration
  const int exact;                                                    //    exact
  const int on;                                          //    on = 3 => triangle, on=4 => quad
  const QuadraturePoint *p;                                 //    les point d'integration
                                                       //    -- les fonctions ------------------
  void Verification();                                                //    for verification
  QuadratureFormular (int o,int e,int NbOfNodes,QuadraturePoint *pp)
       :on(o),exact(e), n(NbOfNodes),p(pp)
       {Verification();}
  QuadratureFormular (int o,int e,int NbOfNodes,const QuadraturePoint *pp)
      :on(o),exact(e),n(NbOfNodes),p(pp)
      {Verification();}

const QuadraturePoint & operator [](int i) const  {return p[i];}
const QuadraturePoint & operator ()(int i) const {return p[i];}

private:
 QuadratureFormular(const QuadratureFormular &)
   :n(0),exact(0),on(0),p(0){throwassert(0);}
 void operator=(const QuadratureFormular &)
   {throwassert(0);}
 QuadratureFormular()
   :n(0),exact(0),on(0),p(0){throwassert(0);}


};

ostream& operator <<(ostream& , const QuadratureFormular & );
ostream& operator <<(ostream& , QuadraturePoint & );


class QuadratureFormular1d {public:
  const int n;
  class Point { public: R p,x; Point(R pp=0,R xx=0): p(pp),x(xx) {} } *p;
  QuadratureFormular1d(Point p0,Point p1,Point p2) : n(3),p(new Point[3]) { p[0]=p0,p[1]=p1,p[2]=p2;}
  QuadratureFormular1d(Point p0,Point p1) : n(2),p(new Point[2]) { p[0]=p0,p[1]=p1;}
  QuadratureFormular1d(Point p0) : n(1),p(new Point[1]) { p[0]=p0;}
  ~QuadratureFormular1d(){ delete [] p;}
 Point operator[](int i) const { return p[i];}
 private:
                                          //    pas operator de copie pour les formules d'intergation
 QuadratureFormular1d()
  : n(0){throwassert(0);};
 QuadratureFormular1d(const QuadratureFormular1d &)
  :n(0) {throwassert(0);}
 void operator=(const QuadratureFormular1d &){throwassert(0);}
};

extern const QuadratureFormular1d QF_GaussLegendre1;
extern const QuadratureFormular1d QF_GaussLegendre2;
extern const QuadratureFormular1d QF_GaussLegendre3;

extern const QuadratureFormular QuadratureFormular_T_1;
extern const QuadratureFormular QuadratureFormular_T_2;
```

```
extern const QuadratureFormular QuadratureFormular_T_5;
extern const QuadratureFormular QuadratureFormular_Q_1;
extern const QuadratureFormular QuadratureFormular_Q_3;
extern const QuadratureFormular QuadratureFormular_Q_5;
extern const QuadratureFormular QuadratureFormular_Q_7;

ostream& operator <<(ostream& f,const  QuadraturePoint & p);
ostream& operator <<(ostream& f, const QuadratureFormular & fi);
```

8.4 Définition d'un l'élément fini

A faire
Les classes qui définissent un élément fini sont les plus difficiles.

```
class ConstructDataFElement {
  friend class FESpace;
  int thecounter;
  int * counter;
  int MaxNbNodePerElement;
  int MaxNbDFPerElement;
  int *NodesOfElement;
  int *FirstNodeOfElement;
  int *FirstDfOfNode;
  int NbOfElements;
  int NbOfDF;
  int NbOfNode;
  int Nproduit;
  ConstructDataFElement(const Mesh &Th,int NbDfOnSommet,int NbDfOnEdge,
                                       int NbDfOnElement,const TypeOfMortar *tm=0,
  int nbdfv=0,const int *ndfv=0,int nbdfe=0,const int *ndfe=0);
  ConstructDataFElement(const ConstructDataFElement *,int k);
  ConstructDataFElement(const FESpace ** l,int k);
  void renum(const long *r,int l) ;
  ~ConstructDataFElement();
 void Make(const Mesh &Th,int NbDfOnSommet,int NbDfOnEdge,
                      int NbDfOnElement,const TypeOfMortar *tm=0,
   int nbdfv=0,const int *ndfv=0,int nbdfe=0,const int *ndfe=0);

};
```

La classe `TypeOfFE`, Contiendra toutes les données qui sont indépendant de l'element mais seulement du type de l'élément, toute l'éléments finis dérivera de cette classe.

```
class TypeOfFE { public:
                                                    //    The FEM is in R^N
                                          //    The FEM is compose from nb_sub_fem
                                             //    dim_which_sub_fem[N] give
  friend class FESpace;
  friend class FElement;
  friend class FEProduitConstruct;
  const int NbDfOnVertex, NbDfOnEdge, NbDfOnElement, N,nb_sub_fem;
  const int NbDoF;
  const  int NbNode;
                                                                    //    remark
  virtual void FB(const Mesh & Th,const Triangle & K,const R2 &P, RNMK_ & val
                                  ) const =0;
  virtual void FB(const bool *,const Mesh & Th,const Triangle & K,
                                   const R2 &P, RNMK_ & val) const =0;
  virtual void Pi_h(const baseFElement & K,RN_ & val, InterpolFunction f, R* v,int
                                    , void *) const=0;

                                                                    //    soit
                                  //    (U_pj)_{j=0,N-1;p=0,nbpoint_P i_h - 1} les valeurs de U au point Phat[i];
                                                //    p est le numero du point d'integration
                                                          //    j est la composante
```

```
                                                    //    l'interpole est defini par
                                                //    Pi_hu = \sum_k u_k w^k, etu_i = \sum_p jalpha_ipj U_pj
                                            //    la matrice de alpha_ipj est tres creuse
struct IPJ {
  int i,p,j;                                        //    i is DoF, p is Point, j is componante
  IPJ(int ii=0,int pp=0,int jj=0):i(ii),p(pp),j(jj) {}
};

friend KN<IPJ> Makepij_alpha(const TypeOfFE **,int );
friend KN<R2> MakeP_Pi_h(const TypeOfFE **,int );

 const KN<IPJ > & Ph_ijalpha() const {return pij_alpha;}      //    la struct de la matrice
 const KN<R2> & Pi_h_R2() const { return P_Pi_h;}                        //    les points
 virtual void Pi_h_alpha(const baseFElement & K,KN_<double> & v) const
   {  assert(coef_Pi_h_alpha);
      v=KN_<double>(coef_Pi_h_alpha,pij_alpha.N());}

                                                                                  //    ----

const int nbsubdivision;                                    //    nb of subdivision for plot
 int const * const DFOnWhat;                          //    0,1,2 vertex 3,4,5 edge 6 triangle
 int const * const DFOfNode;                                 //    numero du df on Node
 int const * const NodeOfDF;                                                    //
 int const * const fromFE;                                  //    the df come from df of FE
 int const * const fromDF;                                   //    the df come from with FE
 int const * const dim_which_sub_fem;
 KN<IPJ > pij_alpha;
 KN<R2 > P_Pi_h;
double *coef_Pi_h_alpha;

                                                                  //    if 0 no plot
public:

  TypeOfFE(const TypeOfFE & t,int k,const int * data)
  : NbDfOnVertex(t.NbDfOnVertex*k),NbDfOnEdge(t.NbDfOnEdge*k)
    ,NbDfOnElement(t.NbDfOnElement*k),N(t.N*k),nb_sub_fem(t.nb_sub_fem*k),
    NbNode(t.NbNode),
   nbsubdivision(t.nbsubdivision),NbDoF(t.NbDoF*k),
  DFOnWhat(data),
  DFOfNode(data+NbDoF),
  NodeOfDF(data+2*NbDoF),
  fromFE(data+3*NbDoF),
  fromDF(data+4*NbDoF),
  dim_which_sub_fem(data+5*NbDoF),
  pij_alpha(t.pij_alpha.N()*k),P_Pi_h(t.P_Pi_h),
  coef_Pi_h_alpha(0)

  { throwassert(dim_which_sub_fem[N-1]>=0 && dim_which_sub_fem[N-1]< nb_sub_fem);
                                                  //    Warning the componant is moving first
      for (int j=0,l=0;j<t.pij_alpha.N();j++)                    //    for all sub DF
        for(int i=0,i0=0; i<k; i++,l++)                           //    for componate
        {
        pij_alpha[l].i=t.pij_alpha[j].i*k+i;                                //    DoF number
        pij_alpha[l].p=t.pij_alpha[j].p;                          //    point of interpolation
        pij_alpha[l].j=t.pij_alpha[j].j+i*t.N;                //    componante of interpolation
        }
  }

  TypeOfFE(const TypeOfFE ** t,int k,const int * data)
  : NbDfOnVertex(sum(t,&TypeOfFE::NbDfOnVertex,k)),
    NbDfOnEdge(sum(t,&TypeOfFE::NbDfOnEdge,k)),
    NbDfOnElement(sum(t,&TypeOfFE::NbDfOnElement,k)),
    N(sum(t,&TypeOfFE::N,k)),nb_sub_fem(sum(t,&TypeOfFE::nb_sub_fem,k)),
    NbNode( (NbDfOnVertex? 3 :0) + (NbDfOnEdge? 3 :0 ) +(NbDfOnElement? 1 :0) ),
    nbsubdivision(max(t,&TypeOfFE::nbsubdivision,k)),
    NbDoF(sum(t,&TypeOfFE::NbDoF,k)),
  DFOnWhat(data),
  DFOfNode(data+NbDoF),
  NodeOfDF(data+2*NbDoF),
  fromFE(data+3*NbDoF),
  fromDF(data+4*NbDoF),
```

```
    dim_which_sub_fem(data+5*NbDoF),
    pij_alpha(Makepij_alpha(t,k)),
    P_Pi_h(MakeP_Pi_h(t,k)),
    coef_Pi_h_alpha(0)


   { throwassert(dim_which_sub_fem[N-1]>=0 && dim_which_sub_fem[N-1]< nb_sub_fem);}

  TypeOfFE(const int i,const int j,const int k,const int NN,const  int  *   data,
           int nsub,int nbsubfem,int kPi,int npPi,double * coef_Pi_h_a=0)
   : NbDfOnVertex(i),NbDfOnEdge(j),NbDfOnElement(k),N(NN),nb_sub_fem(nbsubfem),
     NbNode( (NbDfOnVertex? 3 :0) + (NbDfOnEdge? 3 :0 ) +(NbDfOnElement? 1 :0) ),
    nbsubdivision(nsub),NbDoF(3*(i+j)+k),
    DFOnWhat(data),
    DFOfNode(data+NbDoF),
    NodeOfDF(data+2*NbDoF),
    fromFE(data+3*NbDoF),
    fromDF(data+4*NbDoF),
    dim_which_sub_fem(data+5*NbDoF),
    pij_alpha(kPi),P_Pi_h(npPi),
    coef_Pi_h_alpha(coef_Pi_h_a)

     { throwassert(dim_which_sub_fem[N-1]>=0 && dim_which_sub_fem[N-1]< nb_sub_fem);}

  virtual ~TypeOfFE() { }
  virtual R operator()(const FElement & K,const  R2 & PHat,const KN_<R> & u,
                       int componante,int op) const;

};
```

Donc, pour construire un nouvel élément fini, il faudra utiliser le constructeur `TypeOfFE(i ,j ,k, NN, data, nsub)` où `i,j,k` sont respectivement le nombre de dégrées de liberté sur les sommets, sur les arêtes, dans le triangle, où `NN` est la dimension de l'espace d'arrivé des fonctions de base (1 dans le cas scalaire,3 pour un problème de Stokes bidimensionnel u, v, p), où`data` est un tableau contenant les données des 3 tableaux `DFOnWhat, DFOfNode, NodeOfDF` concaténées , et où `nsub` est le nombre de subdivision du triangle utiliser pour le tracé.

```
class FElement;

class baseFElement { public:
  const FESpace &Vh;
  const Triangle &T;
  const TypeOfFE * tfe;
  const int N;
  const int number;
  baseFElement(  const FESpace &aVh, int k);
  baseFElement(const   baseFElement & K,  const TypeOfFE & atfe);
  R EdgeOrientation(int i) const {return T.EdgeOrientation(i);}
};

class FElement : public baseFElement { public:
  typedef const KN<TypeOfFE::IPJ> &  aIPJ;
  typedef  TypeOfFE::IPJ   IPJ;
  typedef const KN<R2> &  aR2;


   friend class FESpace;
  const int *p;
  const int nb;
  FElement(const FESpace * VVh,int k);
  public:
  int NbOfNodes()const {return nb;}
  int  operator[](int i) const;                                    //    Numero du noeud
  int  NbDoF(int i) const;                                           //    number of DF
  int  operator()(int i,int df) const;            //    Nu du DoF du noeud i de df local df
  int  operator()(int df) const { return operator()(NodeOfDF(df),DFOfNode(df));}
  void Draw(const KN_<R> & U, const  KN_<R> & VIso,int j=0) const;
  void Drawfill(const KN_<R> & U, const  KN_<R> & VIso,int j=0) const;
```

```
  void Draw(const RN_& U,const RN_& V, const  KN_<R> & Viso,R coef,int i0,int i1) const;
  R2   MinMax(const RN_& U,const RN_& V,int i0,int i1) const ;
  R2   MinMax(const RN_& U,int i0) const ;
  void BF(const R2 & P,RNMK_ & val) const;                    //     tfe->FB(Vh.Th,T,P,val);
  void BF(const bool * whatd, const R2 & P,RNMK_ & val) const;
 void Pi_h(RN_ val,InterpolFunction f,R *v,  void * arg) const;                          //
{tfe->Pi_h(Vh.Th,T,val,f,v);}
  aIPJ Pi_h_ipj() const { return tfe->Ph_ijalpha();}
  aR2 Pi_h_R2() const { return tfe->Pi_h_R2();}
  void Pi_h(KN_<R> & v) const { return tfe->Pi_h_alpha(*this,v);}

  int NbDoF() const { return tfe->NbDoF;}
  int DFOnWhat(int i) const { return tfe->DFOnWhat[i];}
  int FromDF(int i) const { return tfe->fromDF[i];}
  int FromFE(int i) const { return tfe->fromFE[i];}
                                                          //    df is the df in element
  int NodeOfDF(int df) const { return tfe->NodeOfDF[df];}                    //    a node
  int DFOfNode(int df) const { return tfe->DFOfNode[df];}     //    the df number on the node

  R operator()(const R2 & PHat,const KN_<R> & u,int i,int op)  const;
 private:
  int nbsubdivision() const { return tfe->nbsubdivision;}                    //    for draw
  };
```

Le problème de la définition d'un élément fini est entre autre la description des dégres des libertés de l'element.
Soit un élément K qui est de type `FElement` alors on a

`K[n]` n° global du noeud local `n` de l'element K,

`K(df)` n° global du degrées de liberté `df` de l'element K,

`K(n,i)` n° global du degrées de liberté `df` du i^{e}degrée de liberté du noeud n de l'élément K,

`K.NbOfNodes()` nombre de noeuds de l'élément de K,

`K.NbDoF()` nombre de degrées de liberté de K,

`K.NbDoF(n)` nombre de degrées de liberté du noeud `n`,

`K.DFOnWhat(df)` n° du support du degrées de liberté `df`,

`K.NodeOfDF(df)` n° local du noeud supportant le degrées de liberté`df`,

`K.DFOfNode(df)` n° local du degrées de liberté par rapport au noeud supportant le degrées de liberté`df`,

`K.BF(`$\hat{P}$`,val)` retourne les valeurs des fonctions et des derivées premieres des fonctions base au point $\hat{P}$ de l'élément de référence. Le tableau val est $val(i,j,k)$ valeur (k=0), dérivée partielle en x (k=1), dérivée partielle en y (k=2) de j^{e}composante de i^{e}fonction de base de l'element.

`K.D2_BF(`$\hat{P}$`,val)` retourne les valeurs des trois derivées secondes.

La classe `FElement` contient toutes les données utile a un élément fini. Mais attention un `FElement` est une classe qui est construit pour chaque élément, elle n'est pas stoker dans un tableau. Donc pour l'utiliser, il ne faut pas utiliser de référence pour définir un `FElement`.

```
  FESpace Vh(Th);                              //    construction d'un élément fini P1 Lagrange
  FElement K=Vh[k];                                                          //    ok
  FElement & K=Vh[k];                                                        //    mauvais
```

La classe `FESpace` permet de définir un espace d'élément fini. Les constructeur de cette classe sont

```
  FESpace Vh(Th);                                                  //    construction d'un élément
                                                                   //    fini P1 Lagrange scalaire

  FESpace Vh3(Vh,3);                                                    //    construction de Vh^3
  FESpace Uh(Th,P2Lagrange);                                           //    P2 lagrange scalaire
  nb=Vh(k);                                                  //    nombre de noeud de l'élément k
  n=Vh(k,i);                                                    //    n° du noeud i de l'élément k
  Vh.NbOfDF;                                                 //    Nombre de degrées de libertées
```

La définition de la classe est :

```
class FESpace : public RefCounter {
  ConstructDataFElement * cdef;                                    //    juste pour les constantes
  public:
  CountPointer<const Mesh> cmesh;
  const int N;                                                                //    dim espace d'arrive
  const int Nproduit;                                  //    dim de l'espace produit generalement 1
  const int NbOfDF;
  const int NbOfElements;
  const int NbOfNodes;
  const Mesh &Th;
   TypeOfFE const * const ptrTFE;
  KN<const TypeOfFE *>  TFE;
  const int nb_sub_fem;      //    nb de sous elements finis tensorise (independe au niveau des
composantes)
  int const* const dim_which_sub_fem; //    donne les dependant des composantes liee a un meme
sous element fini
                                                                          //     exemple si N=5,
                                                                    //    dim_which_sub_fem[0]=0;
                                                                   //    dim_which_sub_fem[1] =1;
                                                                    //    dim_which_sub_fem[2]= 2
                                                                   //    dim_which_sub_fem[3] =2
                                                                    //    dim_which_sub_fem[4] =3
                                                                                        //    =>
                                                 //    le sous elements fini 0 est lie a la composante 0
                                                 //    le sous elements fini 1 est lie a la composante 1
                                               //    le sous elements fini 2 est lie aux composantes 2,3
                                                 //    le sous elements fini 3 est lie a la composante 4
    //    donc pour les CL. les composante 2 et 3 sont lie car elle sont utiliser pour definir un
                                                                          //    meme degre de liberte.

  int const*const NodesOfElement;
  int const*const FirstNodeOfElement;
  int const*const FirstDfOfNodeData;
  const TypeOfMortar * tom;
  const int MaxNbNodePerElement;
  const int MaxNbDFPerElement;
  int FirstDFOfNode(int i) const
              {return FirstDfOfNodeData? FirstDfOfNodeData[i] : i*Nproduit;}
  int LastDFOfNode(int i)  const
              {return FirstDfOfNodeData? FirstDfOfNodeData[i+1] : (i+1)*Nproduit;}
  int NbDFOfNode(int i)  const
              {return FirstDfOfNodeData? FirstDfOfNodeData[i+1]-FirstDfOfNodeData[i] : Nproduit;}
  int MaximalNbOfNodes() const {return MaxNbNodePerElement;};
  int MaximalNbOfDF() const {return MaxNbDFPerElement;};
  const int * PtrFirstNodeOfElement(int k) const {
     return NodesOfElement
             ? NodesOfElement + (FirstNodeOfElement? FirstNodeOfElement[k] : k*MaxNbNodePerElement)
              : 0;}

  int SizeToStoreAllNodeofElement() const {
      return  FirstNodeOfElement
              ?  FirstNodeOfElement[NbOfElements]
               : MaxNbNodePerElement*NbOfElements;}

  int NbOfNodesInElement(int k)   const {
     return FirstNodeOfElement
             ?  FirstNodeOfElement[k+1] - FirstNodeOfElement[k]
              : MaxNbNodePerElement;}
  int  esize() const { return  MaxNbDFPerElement*N*last_operatortype;}
                                                                                        //    par defaut P1
  FESpace(const Mesh & TTh)
   :  cmesh(TTh),cdef(0),N(1),Nproduit(1),Th(TTh),NbOfDF(TTh.nv),
     NbOfElements(TTh.nt),NbOfNodes(TTh.nv),
     FirstDfOfNodeData(0),
     MaxNbNodePerElement(3),MaxNbDFPerElement(3*Nproduit),
     NodesOfElement(0),FirstNodeOfElement(0),
     ptrTFE(0),
     TFE(1,0,&P1Lagrange),tom(0),
     nb_sub_fem(TFE[0]->nb_sub_fem),
     dim_which_sub_fem(TFE[0]->dim_which_sub_fem){}
```

```
    FESpace(const FESpace &,int k );
    FESpace(const FESpace **,int k );
    FESpace(const Mesh & TTh,const TypeOfFE **,int k );
    FESpace(const Mesh & TTh,const TypeOfFE & ,int nbdfv=0,
            const int *ndfv=0,int nbdfe=0,const int *ndfe=0);
    FESpace(const Mesh & TTh,const TypeOfFE &,const TypeOfMortar & );
    ~FESpace();

  int  renum();

  FElement operator[](int k) const { return FElement(this,k);}
  FElement operator[](const Triangle & K) const { return FElement(this,Th.number(K));}

  int  operator()(int k)const {return NbOfNodesInElement(k);}
  int  operator()(int k,int i) const {                          //    the node i of element k
     return NodesOfElement?  *(PtrFirstNodeOfElement(k) + i)  : Th(k,i) ;}

  void Draw(const KN_<R>& U,const KN_<R>& Viso,int j=0) const;          //    Draw iso line
  void Drawfill(const KN_<R>& U,const KN_<R>& Viso,int j=0) const;      //    Draw iso line
  void Draw(const KN_<R>& U,const KN_<R> & Viso,
            R coef,int j0=0,int j1=1) const ;                             //    Arrow
  void Draw(const KN_<R>& U,const KN_<R>& V,const KN_<R> & Viso,
            R coef,int iu=0,int iv=0) const ;                             //    Arrow

  R2 MinMax(const KN_<R>& U,const KN_<R>& V,int j0,int j1,bool bb=true) const;
  R2 MinMax(const KN_<R>& U,int j0, bool bb=true) const;

  bool isFEMesh() const { return!cdef && ( N==1);}                      //    to make optim
private:                                                          //    for gibbs renumbering
  int gibbsv (long* ptvoi,long* vois,long* lvois,long* w,long* v);
};
```

8.4.1 Définition de l'élément P^1 Lagrange

La classe `TypeOfFE_P1Lagrange` dérive de la classe purement virtuelle `TypeOfFE`, qui a pour constructeur `TypeOfFE(i,j,k,NN,data,nsub)` défini en 8.4

```
class TypeOfFE_P1Lagrange : public  TypeOfFE { public:
  static int Data[];
  static double Pi_h_coef[];
   TypeOfFE_P1Lagrange(): TypeOfFE(1,0,0,1,Data,1,1,3,3,Pi_h_coef)
    { const R2 Pt[] = { R2(0,0), R2(1,0), R2(0,1) };
      for (int i=0;i<NbDoF;i++) {
       pij_alpha[i]= IPJ(i,i,0);
       P_Pi_h[i]=Pt[i]; }
     }

   void FB(const Mesh & Th,const Triangle & K,const R2 &P, RNMK_ & val) const;    //    old
version
   void FB(const bool * whatd,const Mesh & Th,const Triangle & K,const R2 &P, RNMK_ & val)
    const;

   void Pi_h(const baseFElement & K,RN_ & val, InterpolFunction f, R* v,int, void *)  const;
   virtual R operator()(const FElement & K,const  R2 & PHat,const KN_<R> & u,
                        int componante,int op) const;

};
int TypeOfFE_P1Lagrange::Data[]={0,1,2,     0,0,0,     0,1,2,     0,0,0,     0,1,2,     0};
double TypeOfFE_P1Lagrange::Pi_h_coef[]={1.,1.,1.};
```

Dans le tableau `data` de taille (ici 9) 3 fois le nombre de dégres de liberte de l'élément (ici 3) qui contient les 3 tableaux concaténés suivant pour chaque degrées de liberté df

- n° du support du degre de liberté df, avec le support défini par $(0, 1, 2)$ pour les sommets, $(3, 4, 5)$ sur l'arête ,et 6 pour le triangle,
- n° du dégre du liberte local au noeud supportant ce degre de liberté df,
- n° du noeud supportant ce degre de liberté

Définition des fonctions de base et de l'interpolation.
La fonction `FB(`$\hat{P}$`,tab` défini calcule : les valeurs des fonctions de bases sur l'élément fini et leurs dérivées partielle premières qui sont retourné dans le tableau `KNMK_<double> tab` évalué en P image de $\hat{P}$(ie. $P = T(\hat{P})$) . Le tableau $tab(i, j, k)$ est défini par $fb_{i,j}$ si $k = 1$, $\frac{\partial fb_{i,j}}{x}$ si $k = 1$, $\frac{\partial fb_{i,j}}{y}$ si $k = 2$; où $fb_{i,j}$ est la j^{e}composante de la fonction fb_i de base n$^{\text{o}}$ i de l'élément K.

```
R TypeOfFE_P1Lagrange::operator()(const FElement & K,const  R2 & PHat,
                                  const KN_<R> & u,int componante,int op) const
{
   R u0(u(K(0))), u1(u(K(1))), u2(u(K(2)));
   R r=0;
   if (op==0)
    {
      R l0=1-PHat.x-PHat.y,l1=PHat.x,l2=PHat.y;
      r = u0*l0+u1*l1+l2*u2;
    }
   else
    {
       const Triangle & T=K.T;
       R2 D0 = T.H(0) , D1 = T.H(1)  , D2 = T.H(2);
       if (op==1)
         r =  D0.x*u0 + D1.x*u1 + D2.x*u2;
        else
         r =  D0.y*u0 + D1.y*u1 + D2.y*u2;
    }
   return r;
}
```

Numerotation de valeurs calculable: `op_id` valeur de la fonction de base, `op_dx` valeur de la derivée en x la fonction de base , etc...

```
                                                          //    numbering of derivative
enum operatortype { op_id=0,
   op_dx=1,op_dy=2,
   op_dxx=3,op_dyy=4,
   op_dyx=5,op_dxy=5,
   op_dz=6,
   op_dzz=7,
   op_dzx=8,op_dxz=8,
   op_dzy=9,op_dyz=9
   };

const int last_operatortype=10;
```

Calcul effectif des fonctions de basee, où whatd est un tableau de `bool` de l`last_operatortype` valeurs qui dit ce qu'il faut à calculer, ce tableau peut etre initialiser avec la fonction `initwhatd`.

```
 void TypeOfFE_P1Lagrange::FB(const bool *whatd,const Mesh &
                              const Triangle & K,const R2 & P,RNMK_ & val) const
{
                                                          //    const Triangle & K(FE.T);
  R2 A(K[0]), B(K[1]),C(K[2]);
  R l0=1-P.x-P.y,l1=P.x,l2=P.y;

  if (val.N() <3)
   throwassert(val.N() >=3);
  throwassert(val.M()==1 );
                                                          //    throwassert(val.K()==3 );

  val=0;
  RN_ f0(val('.',0,op_id));

  if (whatd[op_id])
   {
    f0[0] = l0;
    f0[1] = l1;
    f0[2] = l2;}
 if (whatd[op_dx] || whatd[op_dy])
  {
```

```
  R2 Dl0(K.H(0)), Dl1(K.H(1)), Dl2(K.H(2));

  if (whatd[op_dx])
   {
    RN_ f0x(val('.',0,op_dx));
   f0x[0] = Dl0.x;
   f0x[1] = Dl1.x;
   f0x[2] = Dl2.x;
  }

  if (whatd[op_dy]) {
    RN_ f0y(val('.',0,op_dy));
   f0y[0] = Dl0.y;
   f0y[1] = Dl1.y;
   f0y[2] = Dl2.y;
  }
  }
}
```

Vielle forme, à ne plus utiliser

```
void TypeOfFE_P1Lagrange::FB(const Mesh & ,const Triangle & K,const R2 & P,RNMK_ & val) const;

void TypeOfFE_P1Lagrange::Pi_h(const baseFElement & K,RN_ & val, InterpolFunction f, R* v,int j,
void * arg) const
{
  const R2 Pt[] = { R2(0,0), R2(1,0), R2(0,1) };
   for (int i=0;i<3;i++)
     {
     f(v,K.T(Pt[i]),K,i,Pt[i],arg),val[i]=*(v+j);}

}

static TypeOfFE_P1Lagrange P1LagrangeP1;
TypeOfFE  & P1Lagrange(P1LagrangeP1);
```

8.5 Matrices

Pour des raisons de simplicité, nous ne prenons en compte que des matrices élémentaires et de matrices creuses de type profile qui permettent utiliser les méthodes directes de Cholesky, Crout ou LU.

8.5.1 matrice élémentaire

Une matrice élémentaire qui est soit symétrique ou soit pleine, sera donc juste défini par les deux interpolations et deux pointeurs sur la liste de degré de liberté de l'élément courant, plus d'un pointeur sur la fonction élément et 2 fonctions virtuelles. Ces 2 fonctions qui dépendentent donc du type de la matrice, sont `call` qui va effectivement construire la matrice élémentaire de l'élément k du maillage, `operator()(int i,int j)` qui permet de voir une matrices élémentaire comme un tableau.
Remarque : `operator()(int K)` va construire la matrice élémentaire de l'élément K en appelant la fonction `call`.

```
#ifndef MatriceCreuse_h_
#define MatriceCreuse_h_

template<class T>
 T Square(const T & r){return r*r;}

#include "RNM.hpp"
#include "fem.hpp"
#include "FESpace.hpp"
#include "DOperator.hpp"
#include "QuadratureFormular.hpp"
                                                    //    il faut definir
```

```
                          //      typedef LinearComb<pair<MGauche,MDroit>,C_F0> BilinearForm;
                                                                                        //
template<class R>
class BilinearForm : public  LinearComb<pair<MGauche,MDroit>,R>
{};

using  Fem2D::FESpace;
using Fem2D::FElement;
using Fem2D::baseFElement;
using Fem2D::FMortar;
using Fem2D::TypeOfMortar;
using Fem2D::QuadratureFormular;
using Fem2D::QuadratureFormular1d;
using Fem2D::QuadratureFormular_T_5;
using Fem2D::QF_GaussLegendre3;
const double  EPSILON=1e-20;

                  //    #define APROGRAMMER(a) {cerr << "A PROGRAMMER "  #a << endl; exit (1) ;}
#define ERREUR(a,b) {cerr << "ERREUR " #a<< b <<endl; throw(ErrorExec("FATAL ERREUR dans " __FILE__
"\n" #a  " line: ",__LINE__));}

template <class R> class MatriceCreuse;
template <class R> class MatriceElementaire;
template <class R> class MatriceElementaireSymetrique;
template <class R> class MatriceElementairePleine;



template <class R>
class MatriceElementaire {
public:
  enum TypeOfMatriceElementaire {Full=1,Symetric=2};
  CountPointer<const FESpace> cUh,cVh;
  const FESpace &Uh;
  const FESpace &Vh;
  const QuadratureFormular & FIT;
  const QuadratureFormular1d & FIE;
  KN<R> data;
  MatriceElementaire(const FESpace & UUh,const FESpace & VVh,
                     R* aa,int *nnj,int * nni,TypeOfMatriceElementaire t=Full,
                              const QuadratureFormular & fit=QuadratureFormular_T_5,
                              const QuadratureFormular1d & fie =QF_GaussLegendre3)

       :cUh(UUh),cVh(VVh),Uh(UUh),Vh(VVh),a(aa),ni(nni),nj(nnj),n(0),m(0),mtype(t),data(UUh.esize()+VVh.esize
        FIT(fit),FIE(fie) {}

  MatriceElementaire(const FESpace & UUh,R* aa,int *nni,
                     TypeOfMatriceElementaire t=Symetric,
                     const QuadratureFormular & fit=QuadratureFormular_T_5,
                     const QuadratureFormular1d & fie =QF_GaussLegendre3)
      :cUh(UUh),cVh(UUh),Uh(UUh),Vh(UUh),a(aa),ni(nni),nj(nni),n(0),m(0),mtype(t),data((UUh.esize()))
,
       FIT(fit),FIE(fie){}

  R* a;                                                          //    array coef --
  int *ni,*nj;                                                      //    list of df
  int n,m;                                                       //    n,m number of df
  const TypeOfMatriceElementaire mtype;
  virtual ~MatriceElementaire() {
    if(ni != nj)
      delete [] nj;
    delete [] ni;
    delete [] a;
      }
  virtual R & operator() (int i,int j) =0;
  virtual void call(int ,int ie,int label,void * data) =0;                          //
  const BilinearForm<R> * bilinearform;

  MatriceElementaire & operator()(int k,int ie,int label,void * s=0) {
      call(k,ie,label,s);
      return *this;}
```

```
};


template <class R>
class MatriceElementairePleine:public MatriceElementaire<R> {

  /* --- stockage --
                                                                    //    n = 4 m = 5
                                                                      //    0 1 2 3 4
                                                                      //    5 6 7 8 9
                                                                //    10 11 12 13 14
                                                                //    15 16 17 18 19
    ------------------*/

public:
  R & operator() (int i,int j) {return a[i*m+j];}
                                                          //    MatPleineElementFunc element;
  void  (* element)(MatriceElementairePleine &,const FElement &,const FElement &, R*,int ie,int
label,void *);
  void call(int k,int ie,int label,void *);
  MatriceElementairePleine & operator()(int k,int ie,int label,void * stack=0)
  {call(k,ie,label,stack);return *this;}
  MatriceElementairePleine(const FESpace & VVh)
    :MatriceElementaire<R>(VVh,
new R[VVh.MaximalNbOfDF()*VVh.MaximalNbOfDF()],
new int[VVh.MaximalNbOfDF()],Full),
    element(0) {}
  MatriceElementairePleine(const FESpace & UUh,const FESpace & VVh,
                           const QuadratureFormular & fit=QuadratureFormular_T_5,
                           const QuadratureFormular1d & fie =QF_GaussLegendre3)
    :MatriceElementaire<R>(UUh,VVh,
new R[UUh.MaximalNbOfDF()*VVh.MaximalNbOfDF()],
new int[UUh.MaximalNbOfDF()],
new int[VVh.MaximalNbOfDF()],Full,fit,fie),
    element(0) {}

};

template <class R>
class MatriceElementaireSymetrique:public MatriceElementaire<R> {

                                                                //    --- stockage --
                                                                        //    0
                                                                      //    1 2
                                                                    //    3 4 5
                                                                  //    6 7 8 9
                                                                //    10 . . . .
                                                                        //

public:
  R & operator()(int i,int j)
  {return j < i? a[(i*(i+1))/2 + j] : a[(j*(j+1))/2 + i];}
  void (* element)(MatriceElementaireSymetrique &,const FElement &, R*,int ie,int label,void *);
  void (* mortar)(MatriceElementaireSymetrique &,const FMortar &,void *);
  void call(int k,int ie,int label,void * stack);
  MatriceElementaireSymetrique(const FESpace & VVh,
                           const QuadratureFormular & fit=QuadratureFormular_T_5,
                           const QuadratureFormular1d & fie =QF_GaussLegendre3)
    :MatriceElementaire<R>(
          VVh,
   new R[int(VVh.MaximalNbOfDF()*(VVh.MaximalNbOfDF()+1)/2)],
   new int[VVh.MaximalNbOfDF()],Symetric,
       fit,fie),
       element(0),mortar(0) {}
  MatriceElementaireSymetrique & operator()(int k,int ie,int label,void * stack=0)
  {call(k,ie,label,stack);return *this;};
};


template <class R>
class MatriceProfile;
```

```
                                                    //    classe modele pour matrice creuse
                                                    //    -------------------------------
template <class R>
class MatriceCreuse : public RefCounter,public VirtualMatrice<R> {
public:
  MatriceCreuse(int NbOfDF,int mm,int ddummy)
        : n(NbOfDF),m(mm),dummy(ddummy){}
  MatriceCreuse(int NbOfDF)
        : n(NbOfDF),m(NbOfDF),dummy(1){}
  int n,m,dummy;
  virtual int size() const =0;
  virtual MatriceCreuse & operator +=(MatriceElementaire<R> & )=0;
  virtual void operator=(const R & v) =0;                          //    Mise a zero
  KN_<R> & MatMul(KN_<R> &ax,const KN_<R> &x) const {
    ax= R();
    addMatMul(x,ax);
    return ax;}
  virtual ostream& dump (ostream&)  const =0;
  virtual void Solve(KN_<R> & x,const KN_<R> & b) const =0;
  virtual ~MatriceCreuse(){}
  virtual R & diag(int i)=0;
};

template <class R>
inline ostream& operator <<(ostream& f,const MatriceCreuse<R> & m)
    {return m.dump(f);}

template <class R>
KN_<R> & operator/=(KN_<R> & x ,const MatriceProfile<R> & a);


enum FactorizationType {
    FactorizationNO=0,
    FactorizationCholeski=1,
    FactorizationCrout=2,
    FactorizationLU=3};

template <class R>
class MatriceProfile:public MatriceCreuse<R> {
public:
  mutable R *L;                                                    //    lower
  mutable R *U;                                                    //    upper
  mutable R *D;                                                   //    diagonal
  int *pL;                                                        //    profile L
  int *pU;                                                        //    profile U
  mutable  FactorizationType typefac;
  FactorizationType typesolver;
  ostream& dump (ostream&) const;
  MatriceProfile(const int  n,const R *a);
  MatriceProfile(const FESpace &);
  MatriceProfile(int NbOfDF,R* d,
 R* u, int * pu,
 R* l, int * pl,
 FactorizationType tf=FactorizationNO)
    : D(d),U(u),L(l),pL(pl),pU(pu),
      MatriceCreuse<R>(NbOfDF),typefac(tf),typesolver(FactorizationNO){}

  const MatriceProfile t() const
     {return MatriceProfile(n,D,L,pL,U,pU,typefac);}
  const MatriceProfile lt() const


     {return MatriceProfile(n,0,L,pL,0,0);}
  const MatriceProfile l() const
     {return MatriceProfile(n,0,0,0,L,pL);}
  const MatriceProfile d() const
     {return MatriceProfile(n,D,0,0,0,0);}
  const MatriceProfile ld() const
     {return MatriceProfile(n,D,0,0,L,pL);}
  const MatriceProfile ldt() const
```

```
    {return MatriceProfile(n,D,L,pL,0,0);}
  const MatriceProfile du() const
    {return MatriceProfile(n,D,U,pU,0,0);}
  const MatriceProfile u() const
    {return MatriceProfile(n,0,U,pU,0,0);}
  const MatriceProfile ut() const


    {return MatriceProfile(n,0,0,0,U,pU);}
  void Solve(KN_<R> &x,const KN_<R> &b) const {
     if (typefac==0)
       switch(typefac) {
         FactorizationCholeski: cholesky(); break;
          FactorizationCrout:   crout(); break;
          FactorizationLU:      LU(); break;
       }
    if (&x != &b) x=b;x/=*this;}

  int size() const;
  ~MatriceProfile();
  void addMatMul(const KN_<R> &x,KN_<R> &ax) const;
  void addMatTransMul(const KN_<R> &x,KN_<R> &ax) const
    { this->t().addMatMul(x,ax);}

  MatriceCreuse<R> & operator +=(MatriceElementaire<R> &);
  void operator=(const R & v);                                    //    Mise a zero
  void cholesky(R = EPSILON/8.) const;                                    //
  void crout(R = EPSILON/8.) const;                                       //
  void LU(R = EPSILON/8.) const;                                          //
  R & diag(int i) { return D[i];}
  /*-----------------------------------------------------------------
    D[i] = A[ii]
    L[k] = A[ij]  j < i avec:   pL[i]<= k < pL[i+1] et j = pL[i+1]-k
    U[k] = A[ij]  i < j avec:   pU[j]<= k < pU[j+1] et i = pU[i+1]-k
    remarque pL = pU generalement
    si L = U => la matrice est symetrique
    -----------------------------------------------------------------
  */
};

template <class R>
class MatriceMorse:public MatriceCreuse<R> {
  int nbcoef;
  bool symetrique;
  R * a;
  int * lg;
  int * cl;
public:

 class VirtualSolver :public RefCounter {
   friend class MatriceMorse;
   virtual void Solver(const MatriceMorse<R> &a,KN_<R> &x,const KN_<R> &b) const  =0;
};

  MatriceMorse():MatriceCreuse<R>(0),a(0),lg(0),cl(0),nbcoef(0),solver(0) {};
  MatriceMorse(const int  n,const R *a);
  MatriceMorse(const FESpace & Uh,bool sym)
                :MatriceCreuse<R>(Uh.NbOfDF),solver(0) {Build(Uh,Uh,sym);}
  MatriceMorse(const FESpace & Uh,const FESpace & Vh)
                :MatriceCreuse<R>(Uh.NbOfDF,Vh.NbOfDF,0),solver(0) {Build(Uh,Vh,false);}
  const MatriceMorse t() const;
  void Solve(KN_<R> &x,const KN_<R> &b) const;
  int size() const;
  void addMatMul(const KN_<R> &x,KN_<R> &ax) const;
  void addMatTransMul(const KN_<R> &x,KN_<R> &ax) const;
  MatriceMorse & operator +=(MatriceElementaire<R> &);
  void operator=(const R & v)
    { for (int i=0;i< nbcoef;i++) a[i]=v;}
  ~MatriceMorse(){ if (!dummy) { delete [] a; delete [] cl;delete [] lg;}}
  ostream&  dump(ostream & f) const;
  R * pij(int i,int j) const;
```

```
  R  operator()(int i,int j) const {R * p= pij(i,j);throwassert(p); return *p;}
  R & operator()(int i,int j)  {R * p= pij(i,j);throwassert(p); return *p;}
  R & diag(int i)  {R * p= pij(i,i);throwassert(p); return *p;}
  void SetSolver(const VirtualSolver & s){solver=&s;}
  void SetSolverMaster(const VirtualSolver * s){solver.master(s);}
  bool sym() const {return symetrique;}
  private:
    CountPointer<const VirtualSolver> solver;
    MatriceMorse(const MatriceMorse & );
    void operator=(const MatriceMorse & );
  void  Build(const FESpace & Uh,const FESpace & Vh,bool sym);
};


template<class R,class M,class P>
int ConjuguedGradient(const M & A,const P & C,const KN_<R> &b,KN_<R> &x,
                      const int nbitermax, double &eps,long kprint=1000000000)
{
                    //    ConjuguedGradient lineare A*x est appelé avec des conditions au limites
                                        //    non-homogene puis homogene pour calculer le gradient
   if (verbosity>50)
     kprint=2;
   if (verbosity>99)  cout << A << endl;
   throwassert(&x && &b && &A && &C);
   typedef KN<R> Rn;
   int n=b.N();
   throwassert(n==x.N());
   Rn g(n), h(n), Ah(n), & Cg(Ah);                              //    on utilise Ah pour stocke Cg
   g = A*x;
   R xx= (x,x);
   double epsold=eps;
   g -= b;                                                                    //    g = Ax-b
   Cg = C*g;                                                        //    gradient preconditionne
   h =-Cg;
   R g2 = (Cg,g);
   if (g2 < 1e-30)
    { if(verbosity>1)
       cout << "GC  g^2 =" << g2 << " < 1.e-30  Nothing to do " << endl;
     return 2;  }
   R reps2 =eps >0?  eps*eps*g2 : -eps;                                    //    epsilon relatif
   eps = reps2;
   for (int iter=0;iter<=nbitermax;iter++)
     {
       Ah = A*h;
       R hAh =(h,Ah);
                           //    if (Abs(hAh)<1e-30) ExecError("CG2: Matrix non defined, sorry ");
       R ro =  - (g,h)/ hAh;                                //    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);
       if ( !(iter%kprint) && iter && (verbosity>3) )
         cout << "CG:" <<iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
       if (g2 < reps2) {
         if ( !iter &&!xx && g2 && epsold >0 ) {
                                                 //    change fo eps converge to fast due to the
                                                 //    penalization of boundary condition.
            eps =  epsold*epsold*g2;
            if (verbosity>3 )
            cout << "CG converge to fast (pb of BC)  restart: " << iter <<  "  ro = "
                 << ro << " ||g||^2 = " << g2 << " <= " << reps2 << "  new eps2 =" <<  eps <<endl;
             reps2=eps;
          }
         else
          {
           if (verbosity>1 )
            cout << "CG converge: " << 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 =" << nbitermax
       <<  ", xx= "  << xx<< endl;
  return 0;
}

template<class R,class M,class P>
int ConjuguedGradient2(const M & A,const P & C,KN_<R> &x,const int nbitermax, double &eps,long
kprint=1000000000)
{
       //    ConjuguedGradient2 affine A*x = 0 est toujours appele avec les condition aux limites
                                                                                //    -------------
  throwassert(&x  && &A && &C);
  typedef KN<R> Rn;
  int n=x.N();
  if (verbosity>99) kprint=1;
  R ro=1;
  Rn g(n),h(n),Ah(n), & Cg(Ah);                              //    on utilise Ah pour stocke Cg
  g = A*x;
  Cg = C*g;                                                  //    gradient preconditionne
  h =-Cg;
  R g2 = (Cg,g);
  if (g2 < 1e-30)
   { if(verbosity>1)
      cout << "GC  g^2 =" << g2 << " < 1.e-30  Nothing to do " << endl;
    return 2;  }
  if (verbosity>5 )
    cout << " 0 GC  g^2 =" << g2 << endl;
  R reps2 =eps >0?  eps*eps*g2 : -eps;                          //    epsilon relatif
  eps = reps2;
  for (int iter=0;iter<=nbitermax;iter++)
    {
      R rop = ro;
      x += rop*h;                                            //    x+ rop*h , g=Ax (x old)
      Ah = A*x;                                            //    Ax + rop*Ah = rop*Ah + g =
      Ah -= g;                                                    //    Ah*rop
      R hAh =(h,Ah);
      if (Abs(hAh)<1e-60) ExecError("CG2: Matrix is not defined (/0), sorry ");
      ro =  - (g,h)*rop/hAh;                              //    ro optimal (produit scalaire usuel)
      x += (ro-rop) *h;
      g += (ro/rop) *Ah;                 //    plus besoin de Ah, on utilise avec Cg optimisation
      Cg = C*g;
      R g2p=g2;
      g2 = (Cg,g);
      if ( ( (iter%kprint) == kprint-1)  &&  verbosity >1 )
        cout << "CG:" <<iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
      if (g2 < reps2) {
        if (verbosity )
           cout << "CG converge: " << iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
         return 1;                                                                  //    ok
         }
      R gamma = g2/g2p;
      h *= gamma;
      h -= Cg;                                                    //    h = -Cg * gamma* h
    }
    if (verbosity )
     cout << "CG does'nt converge: " << nbitermax <<   " ||g||^2 = " << g2 << " reps2= " << reps2
<< endl;
  return 0;
}

template <class R>
class MatriceIdentite: VirtualMatrice<R> { public:
 typedef typename VirtualMatrice<R>::plusAx plusAx;
 MatriceIdentite() {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {
    assert(x.N()==Ax.N());
  Ax+=x; }
```

```
 plusAx operator*(const KN<R> &  x) const {return plusAx(this,x);}
};

template<class R>
class SolveGCDiag :   public MatriceMorse<R>::VirtualSolver , public VirtualMatrice<R>{
  int n;
  int nbitermax;
  double eps;
  mutable double  epsr;
  KN<R> D1;
  public:
  SolveGCDiag(const MatriceMorse<R> &A,double epsilon=1e-6) :
    n(A.n),nbitermax(Max(n,100)),D1(n),eps(epsilon),epsr(0) { throwassert(A.sym());
    for (int i=0;i<n;i++)
      D1[i] = 1/A(i,i);}
   void Solver(const MatriceMorse<R> &a,KN_<R> &x,const KN_<R> &b) const  {
     epsr = (eps < 0)? (epsr >0? -epsr : -eps ) : eps;
                                                    //    cout << " epsr = " << epsr << endl;
     ConjuguedGradient<R,MatriceMorse<R>,SolveGCDiag<R> >(a,*this,b,x,nbitermax,epsr);
   }
typename VirtualMatrice<R>::plusAx operator*(const KN_<R> &  x) const
      {return plusAx(this,x);}


 void addMatMul(const KN_<R> & x, KN_<R> & Ax) const
  {  assert(x.N()==Ax.N());
     for (int i=0;i<n;i++)
     Ax[i]+= D1[i]*x[i];}


};
#endif
```

Chapitre 9

Graphique

9.1 Interface Graphique

Cette interface graphique fonctionne sur les 3 systèmes graphiques suivantes:
- X11R6 (système graphique existant sur les ordinateur de type unix, WindowXX), il faut compiler `Xrgraph.cpp` et édité de lien avec la bibliothéque X11R6.
- Window95, Window98, WindowNT,Window2000, il faut compiler avec le compilateur Metrowerks CodeWarrior `PCrgraph.cpp`.
- MacOS, il faut compiler avec le compilateur Metrowerks CodeWarrior `Macrgraph.cpp`.

```
#ifndef RGRAPH_H_
#define RGRAPH_H_
#ifdef __cplusplus
extern "C" {
#endif
  void initgraphique();
  void closegraphique();
  void showgraphic();
  void rattente(int waitm);
  void cadre(float xmin,float xmax, float ymin, float ymax);
  void getcadre(float &xmin,float &xmax, float &ymin, float &ymax);
  void cadreortho(float centrex, float centrey, float rayon);
  void couleur(int c);
  int  LaCouleur();
  void pointe(float x, float y);
  int  InPtScreen(float x, float y);
  int  InRecScreen(float x1, float y1,float x2, float y2);
  void plotstring(const char *s);
  void rmoveto(float x, float y);
  void rlineto(float x, float y);
  void penthickness(int );
  void cercle(float centrex, float centrey, float rayon);
  void execute(const char* s);
  void reffecran();
  void raffpoly(int n, float *poly);
  char Getxyc(float &x,float &y);
  void SetColorTable(int nb);
  void GetScreenSize(int &ix,int &iy);
  void openPS(const char * );
  void closePS(void);
  void myexit(int err=0);
#ifdef __cplusplus
          }
#endif

#endif
```

9.2 Tracer des isovaleurs

```
void plot(int i)
{
  char buf[24];
  snprintf(buf,24,"%d",i);
  plotstring(buf);
}
void plot(double i)
{
  char buf[24];
  snprintf(buf,24,"%g",i);
  plotstring(buf);
}

void DrawIsoT(const R2 Pt[3],const R ff[3],const RN_ & Viso);

void     FillRect(float x0,float y0, float x1, float y1)
 {
     float r[8];
     r[0]=x0;r[1]=y0;
     r[2]=x1;r[3]=y0;
     r[4]=x1;r[5]=y1;
     r[6]=x0;r[7]=y1;
     fillpoly(4,r);
 }

void PlotValue(const RN_ & Viso)
{
   float xmin,xmax,ymin,ymax;
   getcadre(xmin,xmax,ymin,ymax);
   float dx=(xmax-xmin);
   float dy=(ymax-ymin);
   float h=dy/80;
   float x0=xmin+dx*0.8;
   float y= ymin+dy*0.95;
   for (int i=0;i<Viso.N();i++)
    {
       couleur(i+4);
       FillRect(x0,y,x0+h,y+h);
       rmoveto(x0+1.2*h,y);
       y -=  h*1.4;
       plot(Viso[i]);
    }
}

void DrawCommentaire(const char * cm,float x,float y)
{
   float xmin,xmax,ymin,ymax;
   getcadre(xmin,xmax,ymin,ymax);
   float dx=(xmax-xmin);
   float dy=(ymax-ymin);

   rmoveto(xmin+dx*x,ymin+dy*y);
   plotstring(cm);
}

void Mesh::InitDraw() const
   {
     R2 Pmin,Pmax;
     BoundingBox(Pmin,Pmax);
     R2 O((Pmin+Pmax)/2);
     R r =  (Max(Pmax.x-Pmin.x,Pmax.y-Pmin.y)*0.55);
     showgraphic();
     cadreortho((float)O.x,(float)(O.y+r*0.05),(float) r);
    }
inline  R Square(R x){return x*x;}
```

```
void  SetDefaultIsoValue(const RN_& U,RN_ & Viso)
 {
   R umn = U.min();
   R umx = U.max();
   int N = Viso.N();
   R d = (umx-umn)/(N+1);
   R x = umn+d/2;
   for (int i = 0;i < N;i++)
     {Viso[i]=x;x +=d; }
   SetColorTable(N+4);
 }


void FElement::Draw(const RN_& U,const RN_ & Viso,int composante) const
{
  int nsb = nbsubdivision();
  int nsb2 = nsb*nsb;
  int nbdf=NbDoF();
  RN fk(nbdf);
  for (int i=0;i<nbdf;i++)                                    //    get the local value
    fk[i] = U[operator()(i)];
  RNMK fb(nbdf,N,3);                                  //    the value for basic fonction
  RN ff(3);
  R2 Pt,P[3],A(T[0]),B(T[1]),C(T[2]);

  for (int k=0;k<nsb2;k++)
   {
                                                                          //    ff=0.0;
     for (int j=0;j<3;j++)
      {
                                                                //    cout << nbdf << endl;
        Pt=SubTriangle(nsb,k,j);                                    //    current point
        BF(Pt,fb);
        ff[j] = (fb('.',composante,0),fk);
        P[j] = A*(1-Pt.x-Pt.y)+B*Pt.x + C*Pt.y;
       }
     DrawIsoT(P,ff,Viso);
   }
}


void DrawIsoT(const R2 Pt[3],const R ff[3],const RN_ & Viso)
{
  R2 PQ[5];
  int NbIso = Viso.N();
  for(int l=0;l< NbIso;l++)  /*    loop on the level curves */
    {
      R xf = Viso[l];
      int im=0;
      for(int i=0;i<3;i++)                                    //    for the 3 edges
{
  int j = (i+1)%3;
  R fi=(ff[i]);
  R fj=(ff[j]);

  if(((fi<=xf)&&(fj>=xf))||((fi>=xf)&&(fj<=xf)))
    {
      if (Abs(fi-fj)<=0.1e-10)  /* one side must be drawn */
{
  couleur(l+4);
  MoveTo(Pt[i]);
  LineTo(Pt[j]);
}
     else
{
  R  xlam=(fi-xf)/(fi-fj);
  PQ[im++]   = Pt[i] * (1.F-xlam)  +  Pt[j]* xlam;
}
    }
}
```

```
      if (im>=2) /*    draw one segment */
{
  couleur(l+4);
  MoveTo(PQ[0]);
  LineTo(PQ[1]);
}
    }

}

void  FESpace::Draw(const RN_& U,const RN_ & Viso,int j) const
{
  showgraphic();
  SetColorTable(2+6);
  Th.DrawBoundary();
  SetColorTable(Viso.N()+4);
  for (int k=0;k<Th.nt;k++)
    (*this)[k].Draw( U,Viso,j);
}
```

Chapitre 10

Problèmes physique

Nous avons défini des outils pour construire des éléments finis P^1 pour l'instant. Nous allons maintenant utiliser ces classes pour résoudre trois équation au dérivées partiels: une èquation de Laplace, un problème de Stokes et pour finir une équation de Navier-Stokes.
Tous les exemples sont dans
FTP:fem.tar.gz
pour une version tar compressée et sous le répertoire
FTP:sources/fem.

10.1 Un laplacien P1

Le but de cette section est de resoudre l'équation suivant

$$-\Delta U = f \quad \text{dans} \Omega$$
$$f = g \quad \text{sur} \Gamma = \partial\Omega$$

avec $g(x,y) = 3x + 5y$ et $f = 1$.

```
int verbosity =2;
#include <cmath>
#include <cstdlib>
#include <cassert>
#include <iostream>
#include <fstream>
#include "rgraph.hpp"
using namespace std;

#include "throwassert.hpp"
#include "Expression.hpp"
#include "MatriceCreuse_tpl.hpp"
#include "QuadratureFormular.hpp"
using namespace Fem2D;

void GradGrad(MatriceElementaireSymetrique<R> & mat,const FElement & K, double *p,int,
              int, void*)
 {                                                                          //
    const Triangle & T = K.T;                                         //    the triangle
    R * aa=mat.a;
     for(int i=0;i<3;i++) {
       Vertex & vi = T[i];
       R2 Hi = T.H(i);                                              //    some optimisation
   for(int j=0;j<=i;j++) {
     Vertex & vj = T[j];
     R2 Hj = T.H(j);                                              //    some optimisation
     *aa++ = (Hi,Hj)* T.area;}}
 }
```

```
 R g( R2 & P) { return  3*P.x+5*P.y;}
 R f( R2 & P) { return  1;}

int main(int argc, char **argv)
{
  cout << " begin" << endl;
  initgraphique();
  const char  * fn= (argc<2)?"c30x30.msh":argv[1];
  Mesh Th(fn);
  Th.renum();
  Th.Draw();
  FESpace Vh(Th);
  MatriceProfile<R> A(Vh);
  MatriceElementaireSymetrique<R> mate(Vh);
  mate.element = GradGrad;
  for (int i=0;i< Th.nt; i++)
    A += mate(i,-1,0,0);
  const R tgv = 1e30;
  RN U(Th.nv);
  U = 0;

  for (int i=0;i< Th.nt; i++)
    {
                                                    //    set the current Intergration formular
      const QuadratureFormular &FI(QuadratureFormular_T_2);
      assert(FI.on == 3);                                                    //    verification
      const Triangle & K(Th[i]);
      const FElement  S(Vh[i]);
      R area = K.area;
      int i0 = S[0];                                                  //    the 3 vertex number
      int i1 = S[1];
      int i2 = S[2];
      for (int j=0;j<FI.n;j++)                                 //    loop on integration's points
{
  QuadraturePoint PI = FI[j];
                                                  //    the value of the 3 basics function
  R w1(PI.x), w2(PI.y), w0(1-w1-w2);
  R2 P = K[0]*w0 + K[1]*w1 + K[2]*w2;
  R F = f(P)*PI.a;
  U[i0] += F*w0*area;
  U[i1] += F*w1*area;
  U[i2] += F*w2*area;
}
    }
  for (int i=0;i< Th.nv; i++)
    if (!!(Label) Th(i)  )                                    //    just for boundary vertex
      U[i] =  g(Th(i)) * (A.D[i] = tgv);
  A.cholesky();
  U /= A;

  RN Viso(20);
  Th.InitDraw();
  SetDefaultIsoValue( U,Viso);
  reffecran();
  openPS("laplace");
  PlotValue(Viso,0,"Isovalue");
  Vh.Draw(U,Viso);
  closePS();
  rattente(1);
  closegraphique();

  return 0;
}
```

10.2 Problème de Stokes

Le problème de Stokes pour un fluide visqueux de vélocity u et de pression p dans un domaine Ω s'écrit :

$$-\Delta\vec{u} + \vec{\nabla}p = 0,\ \nabla\cdot\vec{u} = 0,\quad \text{in}\ \ \Omega,\ \vec{u} = \vec{u}_\Gamma\ \ \text{on}\ \ \Gamma = \partial\Omega$$

où u_Γ est la vitesse sur le bord.
Des termes de régularisation sont ajouer à l'équation de divergence [11].

```
int verbosity =2;
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <cassert>
#include <iostream>
#include <fstream>
#include "rgraph.hpp"
using namespace std;

#include "throwassert.hpp"
#include "Expression.hpp"
#include "MatriceCreuse_tpl.hpp"
#include "QuadratureFormular.hpp"
using namespace Fem2D;


void MatStokes(MatriceElementaireSymetrique<R> & mat,const FElement & K, double * p,int,
               int, void* mydata)
 {                                                                                  //

                                                                                 //    matrix 3x3
                                                                                     //    e 0 gx
                                                                                     //    0 e gy
                                                                                    //    gx gy d
    const R eps1 = 0.001;
    const Triangle & T =K.T;                                                 //    the current triangle

     for (int i=0,i3=0;i<3;i++, i3+=3)
       {
        R2 Gi = T.H(i);                                                        //    some optimisation
        for (int j=0,j3=0;j<=i;j++, j3+=3)                                      //    the 3 sub matrix
          {
        R2 Gj = T.H(j);                                                        //    some optimisation
        R gg = (Gi,Gj)* T.area;


                                                                                   //    diagonal
            mat(i3  ,j3   ) =  gg;
            mat(i3+1,j3+1)  =  gg;
            mat(i3+2,j3+2)  = -gg*eps1;
            if(j<i)
             {
                                                                                //    Upper part 3x3
               mat(i3  ,j3+1)  = 0;
               mat(i3  ,j3+2)  = T.area*Gj.x/3;
               mat(i3+1,j3+2)  = T.area*Gj.y/3;
             }
                                                                                //    Lower part 3x3
             mat(i3+1,j3  )  = 0;
             mat(i3+2,j3  )  = T.area*Gi.x/3;
             mat(i3+2,j3+1)  = T.area*Gi.y/3;
          }

        }

 }
void ForDebug()
{
  cout << " ForDebug  " << endl;
```

```
}

int main(int argc,char **argv)
{
  cout << " begin" << endl;
  int i;
  atexit(ForDebug);
  initgraphique();
  const char  * fn= (argc<2)?"c10x10.msh":argv[1];
  Mesh Th(fn);
  Th.renum();
  Th.Draw();
  RN Viso(20),Varrow(20);

  FESpace Vh(Th,3);                                                    //    u,v,p
  MatriceProfile<R> A(Vh);
  MatriceElementaireSymetrique<R> mate(Vh);
  mate.element = MatStokes;
  for ( i=0;i< Th.nt; i++) A += mate(i,-1,0,0);
   const R tgv = 1e30;
  RNM U(3,Th.nv);
  U = 0;
  RNM_ D3(A.D,3,Th.nv);                                  //    le vecteur diagonal de la matrice
  for (i=0;i< Th.nv; i++)
    {
      const Vertex & v = Th(i);
      const Label & r(v);
      double x = v.x;
      double y = v.y;
     if (!! r )                                                  //    just for boundary vertex
      {
        D3(0,i) = tgv;
        D3(1,i) = tgv;
        U(0,i)  = ((r.lab==1)? 1 :0) *tgv;
        U(1,i)  =0;
      }
     }


  D3(2,0) = tgv;
  A.crout();
  U /= A;

    cout << " ---------- " << endl;
    cout << "u min " << U(0,'.').min() << "  max =" << U(0,'.').max() << endl;
    cout << "v min " << U(1,'.').min() << "  max =" << U(1,'.').max() << endl;
    cout << "p min " << U(2,'.').min() << "  max =" << U(2,'.').max() << endl;
    cout << " ---------- " << endl;

  Th.InitDraw();
  reffecran();
  SetDefaultIsoValue( U(2,'.'),Viso);
  SetDefaultIsoValue(U(0,'.'), U(1,'.'),Varrow);

  Vh.Draw(U,Viso,2);                                                  //    Draw p
  Vh.Draw(U,Varrow,0.2,0,1);                                        //    Draw u,v

  rattente(1);

  closegraphique();

  return 0;
}
```

10.3 Problème de Navier-Stokes

Soit un fluide incompressible de vitesse $u \in H^1(\Omega)^N$ et pression $p \in L^2(\Omega)$, solution des équations de Navier-Stokes incompressible.

$$\frac{\partial u}{\partial t} + u.\nabla u - \nu\Delta u + \nabla p = 0$$

$$\nabla.u = 0$$

plus des conditions aux limites de type Dirichelet $u|_\Gamma = u_\Gamma$.
Un utilisant la dérivée particulaire pour approcher le terme $\frac{\partial u}{\partial t} + u.\nabla u$, on obtient le schéma suivant:

$$\frac{u^{n+1} - u^n \circ \chi^n}{\Delta t} \approx \frac{\partial u}{\partial t} + u.\nabla u$$

où $\chi^n(x)$ est la fonction de transport qui donne la position de la particule qui est en x au temps $t^{n+1} = t^n + \Delta t$ et qui était en $\chi^n(x)$ au temps t^n. Donc, on a $\chi^n(x) = \zeta_x(t^n)$ où $\zeta_x(t)$ est solution de l'équation différentielle rétrograde suivante $d\zeta_x/dt = u(\zeta_x(t))$ avec condition finale $\zeta_x(t^{n+1}) = x$.
Le domaine de calcul pourra être un carrée.
Nous obtenons le schéma temporeille, suivant d'ordre 1.

$$u^{n+1} - \nu\Delta t\, \Delta u^{n+1} + \Delta t\, \nabla p^{n+1} = u^n \circ \chi^n$$

$$\nabla.u = 0$$

Donc, il ne reste plus qu'à discrétiser le problème de Stokes en utilisant la méthode précédente.
Le programme donne:

```
int verbosity =2;
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <cassert>
#include <iostream>
#include <fstream>
#include "rgraph.hpp"
using namespace std;

#include "throwassert.hpp"
#include "Expression.hpp"
#include "MatriceCreuse_tpl.hpp"
#include "QuadratureFormular.hpp"
using namespace Fem2D;

class Domaine{ public:
  virtual void f()=0;
};

  class tab {
    int n,nx;
    Domaine ** d;
    tab(int nnx) : nx(nnx),d(new Domaine*[nnx]),n(0) {}
    };

#include <ctime>
inline double CPUtime(){
#ifdef SYSTIMES
  struct tms buf;
  if (times(&buf)!=-1)
    return ((double)buf.tms_utime+(double)buf.tms_stime)/(long) sysconf(_SC_CLK_TCK);
  else
#endif
    return ((double) clock())/CLOCKS_PER_SEC;
}

static R KStokes,KUV, KVgradP, KPena;
void MatStokes(MatriceElementaireSymetrique<R> & mat,const FElement & K,R * p,int,
```

```
            int, void* mydata)
{                                                                                    //

                                                                                  //    matrix 3x3
                                                                                      //    e 0 gx
                                                                                      //    0 e gy
                                                                                    //    gx gy d
    const Triangle & T = K.T;                                              //    the current triangle

    for (int i=0,i3=0;i<3;i++, i3+=3)
{
    R2 Gi = T.H(i);                                                      //    some optimisation
    for (int j=0,j3=0;j<=i;j++, j3+=3)                                    //    the 3 sub matrix
{
    R2 Gj = T.H(j);                                                      //    some optimisation
    R cStokes = T.area* KStokes;
    R cVgradP = T.area* KVgradP;
    R cPena   = T.area* KPena;
    R cUV     = T.area *  KUV ;
    R gg  = (Gi,Gj);
    R ggs = gg * cStokes;
    R uv = cUV/12;
    R pq = 0;                                                                  //   1./12.;
    if ( i3==j3) {uv += uv;pq += pq;}
                                                                             //    diagonal
    mat(i3  ,j3   ) =  ggs  + uv;
    mat(i3+1,j3+1)  =  ggs  + uv;
    mat(i3+2,j3+2)  = -(gg+pq)*cPena;
    if(j<i)
{
                                                                          //    Upper part 3x3
    mat(i3  ,j3+1)  = 0;
    mat(i3  ,j3+2)  = -cVgradP*Gi.x/3;
    mat(i3+1,j3+2)  = -cVgradP*Gi.y/3;
}
                                                                          //    Lower part 3x3
    mat(i3+1,j3  )  = 0;
    mat(i3+2,j3  )  = -cVgradP*Gj.x/3;
    mat(i3+2,j3+1)  = -cVgradP*Gj.y/3;
}

}
 if ( K.Vh.Th(T)==0)
  for (int i=0;i<9;i++)
  { cout << i << ":";
   for (int j=0;j<=i;j++)
    cout << mat(i,j) << " ";
   cout << endl;}

}
void ForDebug()
{
    cout << " ForDebug  " << endl;
}

int main(int argc,char **argv)
{

    initgraphique();
    int i;
    atexit(ForDebug);
    cout << " begin" << endl;
    const char  * fn= (argc<2)?"c30x30.msh":argv[1];
    Mesh Th(fn);
    Th.renum();
    Th.Draw();
    RN Viso(20);
    FESpace vh(Th);                                                            //    u,v,p
    FESpace Vh(vh,3);                                                        //    u ou v ou p
    MatriceProfile<R> A(Vh);
    MatriceElementaireSymetrique<R> mate(Vh);
```

```
    KStokes = 1;
    KVgradP = 1;
    KUV     = 0;
    KPena   = 0.001;

   mate.element = MatStokes;
    R T0=CPUtime();
    for ( i=0;i< Th.nt; i++)
      A += mate(i,-1,0,0);
     R T1=CPUtime();
    cout << " CPU assemble "<< T1-T0 << "s" << endl;
    const R tgv = 1e30;
    RNM U(3,Th.nv);
    U = 0;
                                                                                  //
    {  RNM_ D3(A.D,3,Th.nv);                                          //    le vecteur diagonal
    for (i=0;i< Th.nv; i++)
{
    const Vertex & v = Th(i);
    const Label & r(v);
    double x = v.x;
    double y = v.y;
    if (!! r )                                                      //    just for boundary vertex
{
    D3(0,i) = tgv;
    D3(1,i) = tgv;
    U(0,i)  = ((r.lab==1)? 1 :0) *tgv;
    U(1,i)  =0;
}
}


    D3(2,0) = tgv;}

    A.crout();
    U /= A;


    cout << " ---------- " << endl;
    cout << "u min " << U(0,'.').min() << "  max =" << U(0,'.').max() << endl;
    cout << "v min " << U(1,'.').min() << "  max =" << U(1,'.').max() << endl;
    cout << "p min " << U(2,'.').min() << "  max =" << U(2,'.').max() << endl;
    cout << " ---------- " << endl;

    Th.InitDraw();
    reffecran();
    RN VisoU(20);
    SetDefaultIsoValue( U(2,'.'),Viso);
    SetDefaultIsoValue(U(0,'.'), U(1,'.'),VisoU);

    Vh.Draw(U,Viso,2);                                                          //    Draw p
    Vh.Draw(U,VisoU,0.2,0,1);                                                  //    Draw u,v
    rattente(1);

                                                                          //    Navier - Stokes


    R dt = 0.05;
    KStokes = 0.001;
    KUV     = 1/dt;
    KVgradP = 1;
    KPena   = 0.001/dt;

    A = 0;
    for ( i=0;i< Th.nt; i++)
A += mate(i,-1,0,0);

    {
RNM_ D3(A.D,3,Th.nv);                                                 //    le vecteur diagonal
for (i=0;i< Th.nv; i++)
    {
```

```
const Vertex & v = Th(i);
const Label & r(v);
double x = v.x;
double y = v.y;
if (!! r )                                                    //    just for boundary vertex
    {
D3(0,i) = tgv;
D3(1,i) = tgv;
    }
    }


D3(2,0) = tgv;
    }
    A.crout();
    R temps=0;
    RNM U1(U);

    RN_ U1u(U1(0,'.'));
    RN_ U1v(U1(1,'.'));
    RN_ U1p(U1(2,'.'));

    RN_ Uu(U(0,'.'));
    RN_ Uv(U(1,'.'));
    RN_ Up(U(2,'.'));

    for (int itemps=0;itemps< 500;itemps++)
{
    temps += dt;
    cout << "----- Iteration " << itemps << " temps = " << temps << endl;
    U1=0;

    for ( i=0;i< Th.nt; i++)
{
    const QuadratureFormular &FI(QuadratureFormular_T_2);     //    set the current Intergration
formular
    assert(FI.on == 3);                                                    //    verification
    const Triangle & K(Th[i]);
    const FElement  S(Vh[i]);
    int i0 = S[0];                                                  //    the 3 nodes number
    int i1 = S[1];
    int i2 = S[2];
    for (int j=0;j<FI.n;j++)                                  //    loop on integration's points
{
    QuadraturePoint PI = FI[j];
    R cc = (R) PI * K.area/dt;
    R w1(PI.x), w2(PI.y), w0(1-w1-w2);                  //    the value of the 3 basics function
    R l[3];
    l[0]=w0;
    l[1]=w1;
    l[2]=w2;
    int iX=i;
    Walk(Th,iX,l,Uu,Uv,-dt);
    const Triangle & KX(Th[iX]);
    const FElement  SX(Vh[iX]);
    int ix0 = SX[0];                                    //    the 3 node number of triangle iX;
    int ix1 = SX[1];
    int ix2 = SX[2];

    R UoX = Uu[ix0]*l[0] + Uu[ix1]*l[1] + Uu[ix2]*l[2];
    R VoX = Uv[ix0]*l[0] + Uv[ix1]*l[1] + Uv[ix2]*l[2];

    U1u[i0] += UoX*w0*cc;
    U1u[i1] += UoX*w1*cc;
    U1u[i2] += UoX*w2*cc;

    U1v[i0] += VoX*w0*cc;
    U1v[i1] += VoX*w1*cc;
    U1v[i2] += VoX*w2*cc;
}
}
```

```
                                                                    //    take account Boundary condition
                                                                                    //    rattente(1);

    for (i=0;i< Th.nv; i++)
{
    const Vertex & v = Th(i);
    const Label & r(v);
    double x = v.x;
    double y = v.y;
    if ( !! r )                                                      //    just for boundary vertex
{
    U1u[i]  = ((r.lab==1)? 1 :0) *tgv;
    U1v[i]  =0;
}
}

    U1 /= A;
    U = U1;
    cout << " u  " <<  U(0,'.').min() << "  " <<  U(0,'.').max();
    cout << " v  " <<  U(1,'.').min() << "  " <<  U(1,'.').max();
    cout << " p  " <<  U(2,'.').min() << "  " <<  U(2,'.').max() << endl;
          if(itemps % 5 == 0) {
    reffecran();
    SetDefaultIsoValue(U(2,'.'),Viso);
    vh.Draw(Up,Viso);                                                                    //    Draw p
        Vh.Draw(U,VisoU,0.2,0,1);                                                       //    Draw u,v
    }
    if(itemps%100==99)  rattente(1);

}
      cout << " On a fini " << endl;
    reffecran();
    SetDefaultIsoValue(  U(2,'.'),Viso);
   vh.Draw(Up,Viso);
       Vh.Draw(U,VisoU,0.2,0,1);                                                        //    Draw u,v

       rattente(1);
    closegraphique();

    return 0;
}
```

Chapitre 11

Triangulation Automatique

Notation

1. Le convexifié d'un ensemble S de point de $\mathbb{R}^2$ est noté $\mathcal{C}(S)$
2. Le segment extrémités a, b d'un espace vectoriel fermé (resp. ouvert) est noté $[a, b]$ (resp. $]a, b[$).
3. Un ouvert Ω est polygonal si le bord $\partial\Omega$ de cet ouvert est formé d'un nombre fini de segments.
4. l'adhérence de l'ensemble O est notée $\overline{O}$
5. l'intérieur de l'ensemble F est notée $\overset{\circ}{F}$

11.1 Introduction

Nous nous proposons de donner tous les moyens (théorique et pratique) pour écrire un mailleur 2d de types Delaunay Voronoï simple et rapide.
Pour cela, nous définissons un maillage triangulaire $\mathcal{T}_h$ d'un ouvert polygonal Ω_h comme un ensemble de triangles T^k de $\mathbb{R}^2$ pour $k = 1, N_t$ tel que l'intersection des 2 triangles $\overline{T}^i, \overline{T}^j$ de $\mathcal{T}_h$ soit l'ensemble vide, un sommet commun aux 2 triangles, une arête commune aux 2 triangles, ou que $i = j$. Le maillage $\mathcal{T}_h$ maille l'ouvert défini par:

$$\Omega_h \overset{def}{=} \overline{\overset{\circ}{\bigcup_{T\in\mathcal{T}_h} T}} \tag{11.1}$$

De plus $\mathcal{S}_h$ désignera l'ensemble des sommets de $\mathcal{T}_h$ et $\mathcal{A}_h$ désignera l'ensemble des arêtes de $\mathcal{T}_h$. Le bord $\partial\mathcal{T}_h$ du maillage $\mathcal{T}_h$, défini comme l'ensemble des arêtes de $\mathcal{A}_h$ n'appartenant qu'à un triangle de $\mathcal{T}_h$, est un maillage du bord $\partial\Omega_h$ de Ω_h. Par abus de langage, nous confondrons une arête d'extrémités (a, b) et le segment ouvert $]a, b[$ ou fermé $[a, b]$.
Par même nous avons :

Remarque: les triangles sont les composantes connexes de

$$\Omega_h \setminus \bigcup_{(a,b)\in\mathcal{A}_h} [a, b] \tag{11.2}$$

∎

Commençons par donner un théorème fondamental en dimension 2.

Théorème 1 *Pour tout ouvert polygonal de $\mathbb{R}^2$, il existe un maillage de cet ouvert sans sommets interne.*

Les sommets de ce maillage sont les points anguleux du bord $\partial\Omega_h$.
La démonstration de ce théorème utilise le lemme suivant:

Lemme 1 *Dans un ouvert polygonal $\mathcal{O}$ connexe qui n'est pas un triangle, il existe deux points anguleux a, b tels que $]a, b[\subset \mathcal{O}$.*

Preuve : Il existe trois points anguleux a, b, c consécutifs du bord $\partial\mathcal{O}$ tel que l'ouvert soit localement du coté gauche de $]a, b[$ et tels que l'angle $\widehat{abc} < \pi$. Il y a 2 cas :
- soit $]ac[\subset \mathcal{O}$ et nous avons fini dans ce cas.
- sinon $]ab[$ n'est pas inclus dans $\mathcal{O}$, donc l'intersection du bord $\partial\mathcal{O}$ avec le triangle ouvert abc n'est pas vide car $\mathcal{O}$ n'est pas un triangle. Soit $\mathcal{C}$ le convexifié de cette intersection. Par construction ce convexifié $\mathcal{C}$ ne touche pas $]a, b[$ et $]b, c[$ et il est inclus dans le triangle ouvert abc. Soit le points anguleux c' du bord du convexifié le plus proche b, il est tel que $]b, c'[\cup \mathcal{C} = \emptyset$ et par la même $]b, c'[$ est inclus $\mathcal{O}$.

∎

Preuve du théorème 1:
Construisons une suite d'ouverts $\mathcal{O}^i, i = 0, ..., k$ par récurrence avec $\mathcal{O}^0 \stackrel{def}{=} \mathcal{O}$.
Retirons à l'ouvert $\mathcal{O}^i$ un segment $]a_i, b_i[$ joignant deux sommets a_i, b_i et tel que $]a_i, b_i[\subset \mathcal{O}^i$ tant qu'il existe un tel segment

$$\mathcal{O}^{i+1} \stackrel{def}{=} \mathcal{O}^i \setminus]a_i, b_i[\tag{11.3}$$

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

Remarque: 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. ∎

Les données du mailleur sont:
- un ensemble de points

$$\mathcal{S} \stackrel{def}{=} \{x^i \in \mathbb{R}^2 / i \in \{1, ..., N_p\}\} \tag{11.4}$$

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

$$\mathcal{A} \stackrel{def}{=} \{(sa_1^j, sa_2^j) \in \{1, ..., N_p\}^2 / j \in \{1, ..., N_a\}\} \tag{11.5}$$

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

$$\mathcal{SD} \stackrel{def}{=} \{(a^i, sens^i) \in \{1, ..., N_a\} \times \{-1, 1\} / i = 1, N_{sd}\} \tag{11.6}$$

  qui peut être vide (cas par défaut)

La méthode est basée sur les diagrammes de Voronoï, que l'on défini comme suit: les diagrammes de Voronoï sont formés des polygones convexes $V^i, i = 1, N_p$ qui sont les ensembles des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j $i \neq j$ ((c.f. figure 11.1). C'est-à-dire formellement:

$$V^i \stackrel{def}{=} \{x \in \mathbb{R}^2 / ||x - x^i|| \leq ||x - x^j||, j \in \{1, ..., N_p\} \text{ et } j \neq i\} \tag{11.7}$$

Effectivement, ces polygones qui sont des intersections finies de demi-espaces, sont convexes, de plus les sommets v^k de ces polygones sont à égales distances des points $\{x^{i_j^k} / j = 1, ..., n_k\}$ de $\mathcal{S}$ et le nombre n_k est généralement égal à 3 (le cas standard) ou supérieur. A chacun de ces sommets v^k, nous pouvons associer le polygone convexe construit avec les points $\{x^{i_j^k}, j = 1, ..., n_k\}$ en tournant dans le sens trigonométrique. Ce maillage est généralement formé de triangles, sauf si il y a des points cocycliques (cf. figure 11.2 où $n_k > 3$) . Nous pouvons construire ce maillage formellement comme le maillage dual des diagrammes de Voronoï, en reliant deux points x^i et x^j, si les diagrammes V^i et V^j ont un segment en commun (maillage de Delaunay strict). 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: Le domaine d'un maillage de Delaunay d'un ensemble de point $\mathcal{S}$ est l'intérieur du convexifié $\mathcal{C}(\mathcal{S})$ de l'ensemble de points $\mathcal{S}$. ∎

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

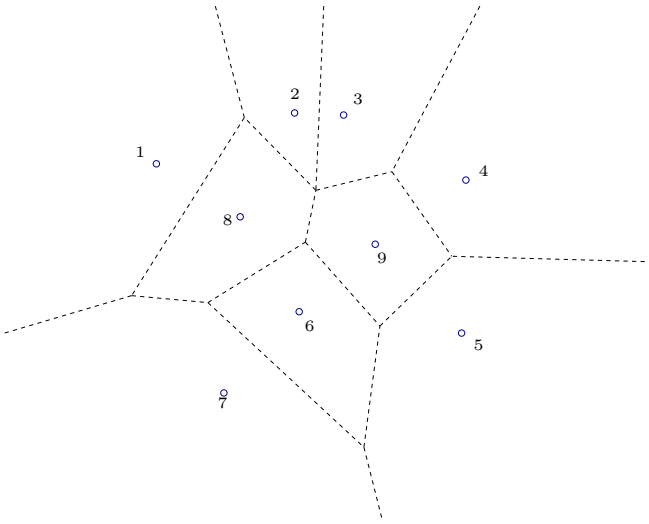

FIG. 11.1 – Diagramme de Voronoï: les ensembles des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j

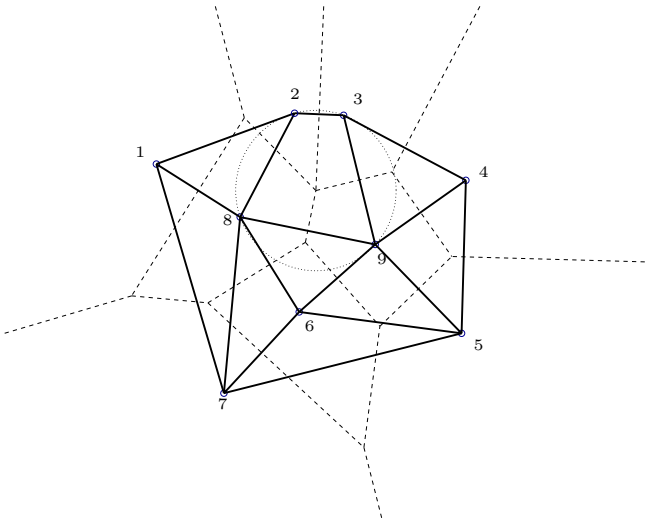

FIG. 11.2 – Maillage de Delaunay: nous relions deux points x^i et x^j, si les diagrammes V^i et V^j ont une ligne en commun

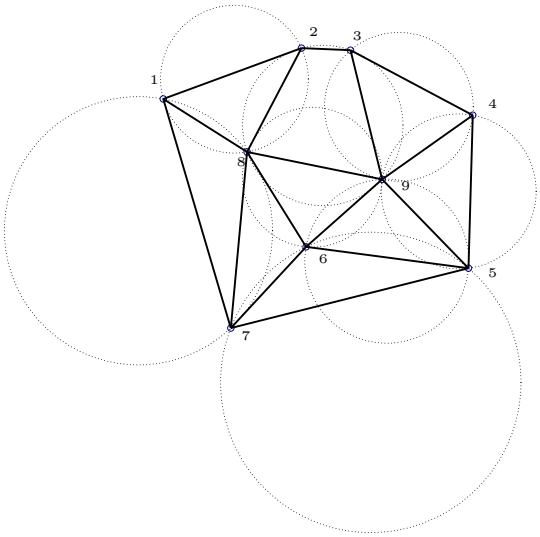

FIG. 11.3 – Propriété de la boule vide: le maillage de Delaunay et les cercles circonscrits aux triangles

Théorème 2 *Un maillage $\mathcal{T}_h$ 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:*

$$D(T) \cap \mathcal{S}_h = \emptyset \tag{11.8}$$

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

Preuve : Il est clair que si le maillage est de Delaunay, il vérifie propriété de la boule vide.
Réciproquement: soit un maillage vérifiant (11.8)
• Montrons que toutes les arêtes (x_i, x_j) du maillage sont telles que l'intersection $V^i \cap V^j$ contienne les centres des cercles circonscrits aux triangles contenant $]x_i, x_j[$.
Soit une arête (x_i, x_j) , cette arête appartient d'au moins 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 $||c - x^k|| < ||c - x^i||$ après (11.7), 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, il y va de même pour V^j ce qui montre •.
Il y a 2 cas: l'arête frontière ou 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 suffisant 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.
– Sinon l'arête (x_i, x_j) est interne, elle est contenu dans 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 convexe, 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 non
– soit le segment $[c, c']$ n'est pas réduit à un point alors l'arête (x_i, x_j) est dans le maillage Delaunay ;
– soit 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 2 triangles T, T' contenant l'arête (x_i, x_j) sont cocycliques et l'arête n'existe pas dans le maillage Delaunay strict.
Pour finir, les arêtes qui ne sont pas dans de le maillage de Delaunay sont entre des triangles cocycliques. Il suffit de remarquer que les classes équivalences des triangles cocycliques d'un même cercle forment un maillage triangulaire des polygones du maillage de Delaunay strict. ∎

Cette démonstration est encore valide en dimension d quelconque, pour cela il faut remplacer les arêtes par des hyperfaces qui sont de codimension 1.

Remarque: Il est possible obtenir un maillage de Delaunay strict en changeant propriété de la boule vide défini en (11.8) par la propriété de la boule vide strict défini comme suit:

$$\overline{D(T)} \cap \mathcal{S}_h = \overline{T} \cap \mathcal{S}_h \tag{11.9}$$

Où $\overline{D(T)}$ est le disque fermé correspondant au cercle circonscrit à un triangle T et où $\mathcal{S}_h$ est l'ensemble de sommets du maillage. La différence entre les deux propriétés (11.9) et (11.8) est qu'il peut exister dans (11.8) d'autre point de $\mathcal{S}_h$ 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 formée de deux triangles adjacents. La démonstration de ce lemme est base sur le propriété des cercles suivantes:

Lemme 2 (Delaunay) *Si le maillage $\mathcal{T}_h$ d'un domaine convexe est tel que tout sous maillage formé de deux triangles adjacents par une arête vérifie la propriété de la boule , alors le maillage $\mathcal{T}_h$ vérifie la propriété de la boule vide global et est de Delaunay.*

La démonstration de ce lemme est basé sur

Alternative 1 *Si deux cercles C_1 et C_2 intersectent sur la droite D coupant le plan les deux demi plan 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 contenue dans le cercle C_i pour $i = 1, 2$.

Exercice 7 : la démonstration de l'alternative est laisse en exercice au lecteur.

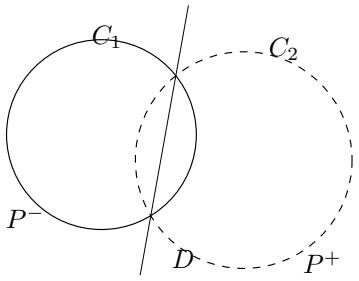

FIG. 11.4 – Représentation graphique de l'alternative 1

Preuve du lemme de Delaunay: Nous faisons une démonstration par l'absurde qui est quelque peu technique. Supposons que le maillage ne vérifie pas la propriété de la boule vide. Alors il existe un triangle $T \in \mathcal{T}_h$ et un point $x^i \in \mathcal{S}_h$, tel que $x^i \in D(T)$
Comme le domaine est convexe, soit a un point interne au triangle T. Nous allons lui associer l'ensemble des triangles $T_a^j, j = 0, .., k_a$ intersectent le segment$]a, x^i[$ qui est contenu dans le domaine convexe (i.e. $\overline{T_a^j} \cap]a, x^i[\neq \emptyset$). Cette ensemble généralement est une une chaîne de triangles T_a^j pour $j = 0, .., k_a$ c'est à dire que les triangles T_a^j et T_a^{j+1} sont adjacents par une arêtes, sauf si par un mauvais hasard le segment $]a, x^i[$ passent par un autre point x^k de $\mathcal{S}_h$. Le lecteur montrera facilement que le le point interne a tel que l'ensemble $T_a^j, j = 0, .., k_a$ de cardinal minimal n'est pas dans ce cas (i.e $]a, x^i[\cap \mathcal{S}_h = \emptyset$) car il est toujours possible de déplacer un petit peut le point a telle que il ne passe plus par un point.
Nous pouvons toujours choisir le couple T et x^i tel que $x^i \in D(T)$ et $x^i \notin T$ telle que le cardinal k de la chaîne droite soit minimal. Le lemme ce résume à montre que $k = 1$.
Soit x^{i_1} le sommet de T^1 opposé à l'arête $T^0 \cup T^1$.
Soit x^{i_0} le sommet de 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 défini par l'intersection des 2 triangles T^0 et T^1, qui est aussi la droite d'intersection des deux cercles $C(T^0)$ et $C(T^1)$. Pour finir, il suffit d'utiliser l'alternative 1 avec les cercles $C(T^0)$ et $C(T^1)$, et comme x^{i_0} n'est dans $D(T^1)$ donc $D(T^0)$ contient $D(T^1)$ dans le demi plan contenant x^{i_0}. Comme $x^i \in C(T^0)$ par hypothèse et comme x^i n'est pas dans le demi plan de contenant x^{i_0} car $]a, x^i[$ 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.

∎

De faite, dans la démonstration, nous montrons que nous pouvons réduire cette propriété au seule motif formée de deux triangles adjacents. De plus comme un quadrilatère non convexe maillé en 2 triangle vérifie la propriété de la boule vide, il suffit quelle soit vérifiée que pour toutes les paires de triangles adjacents formant un quadrilatère convexe.
La démonstration du lemme de Delaunay est encore valide en dimension n, un suffit de changer cercle par sphère et droite par hyperplan, arête par hyperface.
Mais attention malheureusement, en dimension 3, il existe des configurations de deux tétraèdres adjacentes par une faces non convexe qui ne vérifie la propriété de la boule vide.

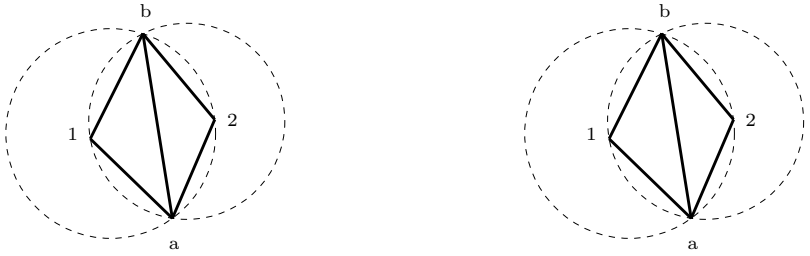

FIG. 11.5 – Échange de diagonale d'un quadrilatère convexe selon le critère de la boule vide

Nous ferons un d'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 11.5.
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):

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

Comme la cotangente est un fonction strictement décroissant 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 $\mathbb{R}^2$, où $det(.,.)$ est le déterminant de la matrice formé avec les 2 vecteurs de $\mathbb{R}^2$.
Ou encore, si l'on ne veut pas diviser et si l'on utilise les aires des triangles $aire^{1ab}$ et $aire^{1ab}$, comme nous avons $det(s^1 - s^a, s^b - s^a) = 2 * aire^{1ab}$ et $det(s^1 - s^2, s^b - s^2) = 2 * aire^{12b}$, le critère d'échange de diagonale optimisé est

$$aire^{12b} \quad (s^1 - s^a, s^b - s^a) \quad > \quad aire^{1ab} \quad (s^1 - s^2, s^b - s^2) \tag{11.10}$$

Maintenant, nous avons théoriquement les moyens de faire un maillage, passant par des points donnés, mais généralement nous ne disposons que les points de la frontière, il nous faudra donc générer des points internes.
Le maillage de Delaunay par construction n'impose rien sur les arêtes, il n'est donc pas moral que ce maillage respecte la discrétisation la frontière comme nous pouvons le remarqué sur la figure 11.7.
D'où deux piste: modifier le maillage afin qu'il respecte la frontière ou bien encore modifier la discrétisation de la frontière afin quel soit contenu dans le maillage de Delaunay. Pour des raisons des compatibilités avec d'autres méthodes nous ne modifierons pas le maillage de la frontière.

FIG. 11.6 – Exemple de maillage d'un polygone sans point interne

FIG. 11.7 – Exemple de maillage de Delaunay ne respectant pas la frontière

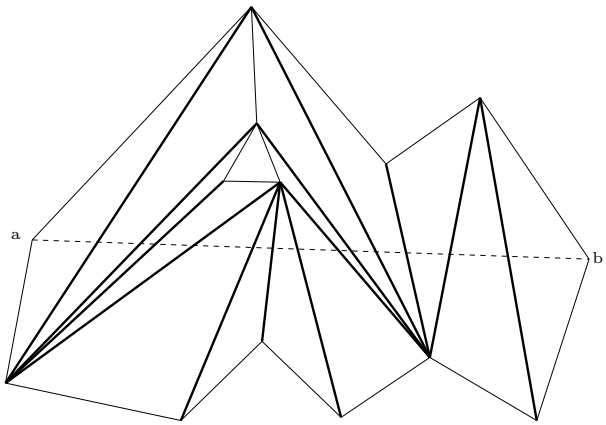

FIG. 11.8 – exemple d'arête $]a, b[$ manquante dans maillage compliqué

11.1.1 Forçage de la frontière

Comme nous avons déjà vue, les arêtes de la frontière données ne sont pas toujours dans le maillage de Delaunay des points frontières (c.f. figure 11.7 et 11.8).

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

Théorème 3 *Soit $\mathcal{T}_h$ une triangulation, soit a et b deux sommets différent de $\mathcal{T}_h$ (donc dans $\mathcal{S}_h$) tel que*

$$]a,b[\cap\mathcal{S}_h=\emptyset \quad \text{et} \quad]a,b[\subset\Omega_h \tag{11.11}$$

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

Preuve : Nous allons faire un démonstration par récurrence sur le nombre $m_{ab}(\mathcal{T}_h)$ arête dans le maillage $\mathcal{T}_h$ coupant l'arête (a,b).
Soit T^i pour $i=0,...,m_{ab}(\mathcal{T}_h)$ 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 $]a,b[\cap\mathcal{S}_h=\emptyset$, l'intersection de $\overline{T}^{i-1}$ et $\overline{T}^{i}$ est une arête noté $[\alpha_i,\beta_j]$ et tel que

$$[\alpha_i,\beta_j] \overset{def}{=} \overline{T}^{i-1}\cap\overline{T}^{i}, \quad \text{avec} \quad \alpha_i\in P_{ab}^{+} \quad \text{et} \quad \beta_i\in P_{ab}^{-} \tag{11.12}$$

où P_{ab}^{+} et P_{ab}^{-} sont les deux demi plans ouverts défini par la droite passant par a et b.
Nous nous placerons dans le maillage restreint $\mathcal{T}_h^{r_{a,b}}$ formé seulement de triangles T^i pour $i=0,...,m_{ab}(\mathcal{T}_h)=m_{ab}(\mathcal{T}_h^{r_{a,b}})$ ce qui nous permet assurer le propriété de localité. De plus le nombre de triangle N_t^{ab} de $\mathcal{T}_h^{a,b}$ est égale à $m_{ab}(\mathcal{T}_h^{r_{a,b}})+1$.
- si $m_{ab}(\mathcal{T}_h^{r_{ab}})=1$ alors il suffit de remarquer de quadrilataire formé avec deux triangles contenant l'unique arête coupant (a,b) est convexe, et donc il suffit d'échanger les arêtes du quadrilataire.
- Nous supposons vraie la propriété pour toutes les arêtes (a,b) vérifiant (11.11) et tel que l'on est $m_{ab}(\mathcal{T}_h^{r_{ab}})<n$ et pour tous les maillages.

Soit une arête (a,b) vérifiant (11.11) et tel que $m_{ab}(\mathcal{T}_h^{r_{a,b}})=n$. Soit α_{i^+} le sommet α_i pour $i=1,...,n$ le plus proche du segment $[a,b]$, nous remarquerons que nous avons les deux inclusions suivantes:

$$]a,\alpha_{i^+}[\subset\overset{\overset{\circ}{\overbrace{\quad}}}{\bigcup_{i=0}^{i^+-1}\overline{T}^{i}}, \quad \text{et} \quad]\alpha_{i^+},b[\subset\overset{\overset{\circ}{\overbrace{\quad}}}{\bigcup_{i=i^+}^{n}\overline{T}^{i}} \tag{11.13}$$

Les 2 arêtes $]a,\alpha_{i^+}[$ et $]\alpha_{i^+},b[$ vérifient les hypothèses de récurrence donc nous pouvons les forcer par échange de diagonale car elles ont des supports disjoints. Nommons $\mathcal{T}_h^{r_{a,b+}}$ le maillage obtenu après forçage de ces deux arêtes, Le nombre de triangle de $\mathcal{T}_h^{r_{a,b+}}$ est égale à $n+1$. D'où, nous avons $m_{ab}(\mathcal{T}_h^{r_{a,b+}})\leq n$.
- Si $m_{ab}(\mathcal{T}_h^{r_{a,b+}})<n$,pour finir, appliquons l'hypothèse de récurrence.
- Sinon $m_{ab}(\mathcal{T}_h^{r_{a,b+}})=n$, nous allons forcer (a,b) dans le maillage $\mathcal{T}_h^{r_{a,b+}}$, nous utilisons la même méthode. Les T^i seront maintenant les triangles de $\mathcal{T}_h^{r_{a,b+}}$. Les α^+ en β^+ sont définis par équation (11.12). Nous avions traité le demi plan supérieur. Nous traitons la partie inférieure. Nous forçons les deux arêtes $]a,\beta_{i^-}[$ et $]\beta_{i^-},b[$ où i^- est tel que le sommet $\beta_{i^+}^{+}$ soit le plus proche du segment $]a,b[$ des sommets β_i^{+} en utilisant les mêmes arguments que précédemment.

Nous avons donc un maillage local du quadrilataire $a,\alpha_{i^+},b,\beta_{i^-}^{+}$ qui contient l'arête $]a,b[$ et qui ne contient aucun autre point du maillage, il est donc formé de 2 triangles T,T' tel que tel que $]a,b[\subset\overline{T}\cup\overline{T}'$ ce qui nous permet de utiliser une dernière fois l'hypothèse de récurrence (n=1) pour achever la démonstration. ∎

On en déduit facilement autre démonstration de théorème 1: Il suffit de prendre un maillage de Delaunay de l'ensemble des sommets de l'ouvert, de forcer tout les segments frontières de l'ouvert, et de retirer les triangles qui ne sont pas dans l'ouvert.
Du théorème 3, Il découle évidement:

Théorème 4 *Soit deux maillages $\mathcal{T}_h$ et $\mathcal{T}'_h$ ayant même sommet ($\mathcal{S}_h = \mathcal{S}'_h$) et même maillage de bord $\partial\mathcal{T}_h = \partial\mathcal{T}'_h$ alors il existe une suite de échange de diagonale de quadrilataire convexe qui permet de passer du maillage $\mathcal{T}_h$ au maillage $\mathcal{T}'_h$.*

Preuve : Il suffit de forcer toutes les arêtes du maillage $\mathcal{T}_h$ dans $\mathcal{T}'_h$. ∎

Et nous pouvons donné un algorithme très simple dû à Borouchaki de forçage d'arête.

Algorithme 9 *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 2 triangles dont la diagonale $]s^\alpha, s^\beta[$ coupe $]s^a, s^b[$ en utilisant les critères suivant:*
– *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 y a une solution au problème, le faite de faire des échanges de diagonale aléatoire va permettre de converger car statistiquement tous les maillages possibles sont parcourus.

11.1.2 Recherches des sous domaines

Il faut repérer les parties qui sont les composantes connexe par arc de $\Omega_h \setminus \cup_{j=1}^{N_a}[x^{sa_1^j}, x^{sa_2^j}]$, ce qui revient à définir les composantes connexe du graphe des triangles adjacents où l'on a supprimer 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 10 coloriage de sous domaines

```
Coloriage(T) {
 Si T n'est pas Colorié {
  Pour tous les triangles T' adjacents à T par une arête non marqué
   Coloriage(T') }}
marquer tous les arêtes T_h qui sont dans A
Pour tous les Triangles T non coloriés {
 Changer de couleur
  Coloriage(T) }
```

A chaque couleur correspond une composante connexe de $\Omega_h \setminus \cup_{j=1}^{N_a}[x^{sa_1^j}, x^{sa_2^j}]$ et la complexité de l'algorithme est en $3 * N_t$ (où N_t est nombre de triangles).
Attention, si l'on utilise simplement la récurcivité du langage, nous allons devant de gros problèmes car la profondeur de la pile est généralement de l'ordre du nombre d'éléments du maillage.

Exercice 8 : Réécrire l'algorithme 10 sans utiliser la récurcivité.

11.1.3 Génération des points internes

Comme nous pouvons le remarquer sur la figure 11.3, il nous faut générer des points internes.
Par exemple nous pouvons connaître une fonction qui donne le pas de maillage $h(x)$ que nous voulons en tous points x de $\mathbb{R}^2$.
Mais généralement nous avons pas d'autre information, il nous faudra donc construire un fonction $h(x)$ qui défini le pas de maillage, à partir des données.
Pour ce faire, par exemple nous pouvons utiliser la méthode suivante:

1. En chaque sommet s de $\mathcal{S}_h$ nous pouvons associer un pas de maillage qui est la moyenne de longueurs des arêtes ayant comme sommet s, Si un sommet de $\mathcal{S}_h$ qui n'est contenu dans aucune arête alors nous lui affectons une valeur par défaut par exemple de pas de maillage moyen.
2. Puis faire le maillage de Delaunay de l'ensemble de points
3. Pour finir, interpoler P^1 les h dans tous les triangles de Delaunay.

Dans la pratique nous aimerions disposer d'une fonction $N(a, b)$ de $\mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ qui nous donne le nombre de mailles qu'il doit y avoir entre les points a, b. Nous pouvons construire différent type de fonction N à partir de h.

Méthode utilisant d'un moyenne simple: $|ab|^{-1}\int_a^b h(t)dt$

$$N_1(a,b) \stackrel{def}{=} |ab|(\frac{\int_a^b h(t)dt}{|ab|})^{-1} = (\int_a^b h(t)dt)^{-1} \tag{11.14}$$

Nous pouvons aussi utiliser la géométrie différentiel. La longueur l d'une courbe γ paramétré par $t \in [0,1]$ est défini par

$$l \stackrel{def}{=} \int_{t=0}^{1} \sqrt{(\gamma'(t),\gamma'(t))}dt. \tag{11.15}$$

Si l'on veut que la longueur dans le plan tangent en x d'un segment donne le nombre de pas (i.e. nous divisons par $h(x)$) Il suffit utiliser d'utiliser la longueur Remmanienne suivante:

$$l^h \stackrel{def}{=} \int_{t=0}^{1} \frac{\sqrt{(\gamma'(t),\gamma'(t))}}{h(\gamma(t))}dt \tag{11.16}$$

Ce qui nous donne comme autre définition

$$N_2(a,b) \stackrel{def}{=} \int_a^b h(t)^{-1}dt. \tag{11.17}$$

Ces deux méthodes de construction sont équivalentes dans le cas où $h(x)$ indépendant de x. Regardons ce qu'il va changer de le cas affine, sur le segment $[0,l]$. La fonction $h(t)$ est défini par $h_0 + (h_l - h_0)t$.

$$N_1(0,t) = (\int_0^t h_0 + (h_l - h_0)xdx)^{-1} = (h_0 t + \frac{(h_l - h_0)}{2}t^2)^{-1} \tag{11.18}$$

$$N_2(0,t) = \frac{\ln(\frac{h_0 + h_1\, t - h_0\, t}{h_0})}{h_1 - h_0} \tag{11.19}$$

Par construction, N_2 verifie 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 verifie clairement pas cette relation.
Voilà une de raison pour les quelques nous préfèrons la methode N_2 et de plus si nous voulons mettre de points optimaux tels que $N_2(0,t) = i, \forall i \in 1,2,..$ dans 11.19. ces points sont alors distribués suivant une suite géométrique de raison $\frac{\ln(h1-h0)}{h1-h0}$.

11.2 Algorithme de Maillage

1. Prendre pour origine le point de coordonné minimal $O = (\min_{i=1,N_p} x_k^i)_{k=1,2}$ où les $(x_k^i)_{k=1,2}$ sont les coordonnées des points x^i.
2. Trier les x^i avec l'ordre lexicographique par rapport à la norme des $x^i - O$ et puis par rapport x_2^i. Cette ordre est telle que le point courant ne soit jamais dans le convexifié des points précédents.
3. Ajouter les points un à un suivant l'ordre pré-défini par l'étape 2. Les points ajoutes sont toujours à l'extérieur du maillage courant. Donc on peut créer un nouveau maillage en reliant tous les arêtes frontières du maillage précédent qui voit les points (du bon cotés).
4. Pour rendre le maillage de Delaunay, il suffit d'appliquer la méthode **optimisation** du maillage autour du points x^k en rendant toutes les motifs forment de 2 triangles contenant le point x^k de Delaunay, et cela pour tous les points x^k du maillage.
5. Il faut forcer la frontière dans le maillage de Delaunay en utilisant algorithme 9.
6. Il faut retirer les parties du domaine externes en utilisant l'algorithme 10.
7. Il faut créer des points internes en utilisant la génération des points internes. Si il y a des points internes nous recommençons en 1) avec un ensemble de point auquel nous avons ajouté les points internes.
8. Régularisation, optimisation du maillage

Bibliographie

[1] D. KNUTH The art of programming, tome 3,...

[2] The C++ , programming language, Third edition, Bjarne Stroustrup, Addison-Wesley 1997.

[3] D. Bernardi, F.Hecht, K. Ohtsuka, O. Pironneau: *freefem+ documentation*, on the web at on the web at http://www-rocq.inria.fr/Frederic.Hecht/FreeFemPlus.htm

[4] D. Bernardi, F.Hecht, O. Pironneau, C. Prud'homme: *freefem documentation*, on the web at http://www-rocq.inria.fr/gamma/cdrom/www/freefem/fraold.htm

[5] P.L. George, H. Borouchaki : *Automatic triangulation*, Wiley 1996.

[6] P.J. Frey, P.L. George: *Maillages*, Hermes 1999.

[7] F. Hecht: The mesh adapting software: bamg. INRIA report 1998. on the web at http://www-rocq.inria.fr/gamma/cdrom/www/freefem/../bamg/fra.htm

[8] J.L. Lions, O. Pironneau: Parallel Algorithms for boundary value problems, Note CRAS. Dec 1998. Also : Superpositions for composite domains (to appear)

[9] B. Lucquin, O. Pironneau: *Scientific Computing for Engineers* Wiley 1998.

[10] F. Preparata, M. Shamos ; *Computational Geometry* Springer series in Computer sciences, 1984.

[11] R. Rannacher: On Chorin's projection method for the incompressible Navier-Stokes equations, in "Navier-Stokes Equations: Theory and Numerical Methods" (R. Rautmann, et al., eds.), Proc. Oberwolfach Conf., August 19-23, 1991, Springer, 1992

[12] J.L. Steger: The Chimera method of flow simulation, Workshop on applied CFD, Univ of Tennessee Space Institute, August 1991.