Note cours Informatique Scientifique 2004

Frédéric Hecht, Olivier Pironneau
Laboratoire Jaques-Louis Lions, Université Pierre et Marie Curie

26 mars 2004

Table des matières

1 Quelques éléments de syntaxe **4**

1.1 Les déclarations du C++ 5

1.2 Comment Compile et éditer de liens 5

1.3 Quelques règles de programmation 8

1.4 Vérificateur d'allocation 10

2 Le Plan $\mathbb{R}^2$ **12**

2.1 La classe R2 14

3 Les classes tableaux **15**

3.1 Version simple d'une classe tableau 15

3.2 les classes RNM 18

3.3 Exemple d'utilisation 19

3.4 Un resolution de systéme linéaire avec le gradient conjugué 21

3.4.1 Gradient conjugué préconditionné 22

3.4.2 Test du gradient conjugué 23

3.4.3 Sortie du test 24

4 Méthodes d'éléments finis P_1 Lagrange **27**

4.1 Le problème et l'algorithme 27

4.2 Les classes de base pour les éléments finis 28

4.2.1 La classe `Label` (numéros logiques) 28

4.2.2 La classe `Vertex` (modélisation des sommets) 29

4.2.3 La classe `Triangle` (modélisation des triangles) 30

4.2.4 La classe `BoundaryEdge` (modélisation des arêtes frontières) 31

4.2.5 La classe `Mesh` (modélisation du maillage) 32

4.2.6 Le programme principale 35

4.2.7 Execution 37

5 Chaînes et Chaînages **37**

5.1 Introduction 37

5.2 Construction de l'image réciproque d'une fonction 38

5.3 Construction des arêtes d'un maillage 38

5.4 Construction des triangles contenant un sommet donné 40

5.5 Construction de la structure d'une matrice morse 41

5.5.1 Description de la structure morse 42

5.5.2 Construction de la structure morse par coloriage 42

6 Différentiation automatique **45**

6.1 Le mode direct 45

6.2 Fonctions de plusieurs variables 48

6.3 Une bibliothèque de classes pour le mode direct 48

6.4 Principe de programmation 48

6.5 Implémentation comme bibliothèque C++ 49

7 Algèbre de fonctions **53**

7.1 Version de base 53

7.2 Les fonctions C^∞ 54

1 Quelques éléments de syntaxe

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

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

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

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

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

  où (T est un type), et où `(n-args )` est la déclaration classique des n arguments. Remarque si opérateur est défini dans une classe alors le premier argument est la classe elle même et donc le nombre d'arguments est $n-1$.
- Les règles de conversion d'un type T en A par défaut qui sont génèrées à partir d'un constructeur A(T) dans la classe A ou avec l'opérateur de conversion (cast en anglais) `operator (A) ()` dans la classe T, `operator (A) (T)` hors d'une classe. De plus il ne faut pas oublier que C++fait automatiquement un au plus un niveau de conversion pour trouver la bonne fonction ou le bon opérateurs.
- Programmation générique de base (c.f. template). Exemple d'écriture de la fonction min générique suivante `template<class T> T & min(T & a,T & b){return a<b? a :b;}`
- Pour finir, connaître seulement l'existence du macro générateur et ne pas l'utiliser.

1.1 Les déclarations du C++

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

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

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

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

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

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

ou encore avec déclaration et puis allocation :

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

1.2 Comment Compile et éditer de liens

Comme en `C` ; dans les fichiers `.cpp`, il faut mettre les corps des fonctions et de les fichiers `.hpp`, il faut mettre les prototype, et le définition des classes, ainsi que les fonctions `inline` et les fonctions `template`, Voilà, un exemple complète avec trois fichiers `a.hpp`,`a.cpp`,`tt.hpp`, et un Makefile dans `http://www.ann.jussieu/~hecht/ftp/InfoSciO4/l1/a.tar.gz`

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

Listing 1 *(a.hpp)*

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

Listing 2 *(a.cpp)*

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

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

Listing 3 *(tt.cpp)*

```
#include <iostream>
#include "a.hpp"
using namespace std;
int main(int argc,char ** argv)
{
  for(int i=0;i<argc;++i)
    cout << " arg " << i << " = " << argv[i] << endl;
  A a[10];                                                    //    un tableau de 10 A
  return 0;                                                                  //    ok
}
```

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

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

Pour faire les trois choses en même temps, entrez :

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

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

```
[brochet:P6/DEA/sfemGC] hecht% ./tt aaa bb cc dd qsklhsqkfhsd " -----------
 arg 0 = ./tt
 arg 1 = aaa
 arg 2 = bb
 arg 3 = cc
 arg 4 = dd
```

```
 arg 5 = qsklhsqkfhsd
 arg 6 =  -----------
 Constructeur A par défaut 0xbfffeef0
 Constructeur A par défaut 0xbfffeef1
 Constructeur A par défaut 0xbfffeef2
 Constructeur A par défaut 0xbfffeef3
 Constructeur A par défaut 0xbfffeef4
 Constructeur A par défaut 0xbfffeef5
 Constructeur A par défaut 0xbfffeef6
 Constructeur A par défaut 0xbfffeef7
 Constructeur A par défaut 0xbfffeef8
 Constructeur A par défaut 0xbfffeef9
```

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

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

Listing 4 *(Makefile)*

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

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

Pour l'utilisation :

- pour juste voir les commandes exécutées sans rien faire :
  ```
  [brochet:P6/DEA/sfemGC] hecht% make -n tt
  g++  -c  tt.cpp
  g++  -c  a.cpp
  g++  -o tt tt.o a.o
  ```
- pour vraiment compiler
  ```
  [brochet:P6/DEA/sfemGC] hecht% make  tt
  g++  -c  tt.cpp
  g++  -c  a.cpp
  g++  -o tt tt.o a.o
  ```
- pour recompiler avec une modification du fichier `a.hpp` via la command `touch` qui change la date de modification du fichier.
  ```
  [brochet:P6/DEA/sfemGC] hecht% touch a.hpp
  [brochet:P6/DEA/sfemGC] hecht% make tt
  g++  -c  tt.cpp
  ```

```
g++  -c  a.cpp
g++  -o tt tt.o a.o
```
remarque : les deux fichiers sont bien à recompiler car il font un *include* du fichier `a.hpp`.

– pour nettoyer :
```
[brochet:P6/DEA/sfemGC] hecht% touch a.hpp
[brochet:P6/DEA/sfemGC] hecht% make clean
rm *.o tt
```
Remarque : Je vous conseille très vivement d'utiliser un `Makefile` pour compiler vous programme.

Exercice 1 Ecrire un Makefile pour compile et tester tous les programmes `http://www.ann.jussieu/~hecht/ftp/InfoSci04/l1/exemple_de_base`

1.3 Quelques règles de programmation

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

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

Voilà quelques règles à respecté.

Règle 1 *absolue* Dans une classe avec des pointeurs et avec un destructeur, il faut que les deux opérateurs de copie (création et affection) soient définis. Si vous considérez que ces deux opérateurs ne doivent pas exister alors les déclarez en privé sans les définir.

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

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

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

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

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

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

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

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

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

Règle 2 ‖ Dans une fonction, ne jamais retournez de référence ou le pointeur sur une variable locale

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

Il ne faut jamais faire ceci :

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

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

Règle 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 `http://www.ann.jussieu/~hecht/ftp/InfoSciO4/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 règle).
Exemple d'une petite classe qui modélise un tableau d'entier

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

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

Règle 4 N'utilisez le macro générateur que si vous ne pouvez pas faire autrement, ou pour ajoutez du code de vérification ou test qui sera très utile lors de la mise au point.

Règle 5 Une fois toutes les erreurs de compilation et d'édition des liens corrigées, il faut éditer les liens en ajoutant `CheckPtr.o` (le purify du pauvre) à la liste des objets à éditer les liens, afin de faire les vérifications des allocations.

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

1.4 Vérificateur d'allocation

L'idée est très simple, il suffit de surcharger les opérateurs `new` et `delete`, de stocker en mémoire tous les pointeurs alloués et de vérifier avant chaque déallacation s'il fut bien alloué (cf. `AllocExtern::MyNewOperator(size_t )` et `AllocExternData.MyDeleteOperator(void *)`. Le tout est d'encapsuler dans une classe `AllocExtern` pour qu'il n'y est pas de conflit de nom. De plus, on utilise `malloc` et `free` du C, pour éviter des problèmes de récurrence infinie dans l'allocateur. Pour chaque allocation, avant et après le tableau, deux petites zones mémoire de 8 octets sont utilisées pour retrouver des débordement amont et aval.

Et le tout est initialisé et terminé sans modification du code source en utilisant la variable `AllocExternData` globale qui est construite puis détruite. À la destruction la liste des pointeurs non détruits est écrite dans un fichier, qui est relue à la construction, ce qui permet de déboguer les oublis de déallocation de pointeurs.

Remarque 1 *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 `http://www.ann.jussieu/~hecht/ftp/InfoSci04/CheckPtr.cpp`. Pour l'utiliser, il suffit de compiler et d'éditer les liens avec les autres parties du programme.
Il est aussi possible de retrouver les pointeurs non désalloués, en utilisant votre débogueur favori (par exemple gdb).
Dans `CheckPtr.cpp`, il y a une macro du préprocesseur `DEBUGUNALLOC` qu'il faut définir, et qui active l'appel de la fonction `debugunalloc()` à la création de chaque pointeur non détruit. La liste des pointeurs non détruits est stockée dans le fichier `ListOfUnAllocPtr.bin`, et ce fichier est génèré par l'exécution de votre programme.
Donc pour déboguer votre programme, il suffit de faire :

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

Exercice 2 un Makefile pour compiler et tester tous les programmes du `http://www.ann.jussieu/~hecht/ftp/InfoSci04/l1/exemple_de_base` avec `CheckPtr`

Compréhension des constructeurs, destructeurs et des passages d'arguments

Exercice 3

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

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

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

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

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

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

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

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 `http://www.ann.jussieu/~hecht/ftp/InfoSci04/l1/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
//    pas de destructeur car pas de new
//    donc les deux operateurs de copies fonctionnent
  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 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 qu'un paramètre. Le premier paramètre étant la classe et le second étant le paramètre fourni ;
- si un opérateur ou une fonction membre d'une classe ne modifie pas la classe alors il est conseillé de dire au compilateur que cette fonction est « constante »en ajoutant le mots clef **const** après la définition des paramètres ;
- dans le cas d'un opérateur défini hors d'une classe le nombre de paramètres est donné par le type de l'opérateur uniare (`+ - * ! [] etc..` : 1 paramètre), binaire (`+ - * / | & || &&^ == <= >= < > etc..` 2 paramètres), n-aire (`()` : n paramètres).

– `ostream`, `istream` sont les deux types standards pour respectivement écrire et lire dans un fichier ou sur les entrées sorties standard. Ces types sont définis dans le fichier « iostream »inclue avec l'ordre `#include<iostream>` qui est mis en tête de fichier ;
– les deux opérateurs `<<` et `>>` sont les deux opérateurs qui généralement et respectivement écrivent ou lisent dans un type `ostream`, `istream`, ou `iostream`.

2.1 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
```

Exercice 4

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

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

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

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

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

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

Q4 modifier les sources `http://www.ann.jussieu/~hecht/ftp/InfoSciO4/l1/pj0.tar.gz` afin d'affiché graphique, il suffit de modifier la fonction `void Draw()`, de plus les bornes de l'écran sont entières et sont stockées dans les variables global `int Width , Height ;`.

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

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

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

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

3 Les classes tableaux

Nous commencerons sur une version didactique, nous verrons la version complète qui est dans le fichier « tar compressé » `http://www.ann.jussieu/~hecht/ftp/InfoSciO4/RNM.tar.gz`.

3.1 Version simple d'une classe tableau

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

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

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

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

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

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

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

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

};
```

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

```
                    //    version avec avec une copie du tableau au niveau du return
```

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

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

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

Pour la multiplication par un scalaire à droite on a :

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

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

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

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

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

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

```
 A a(n),b(n),c(n),d(n);
 A t1(n),t2(n),t3(n),t4(n);
 muld(2.,b),t1);                                              //    t1 = 2.*b
 add(a,t1,t2);                                               //    t2 = a+2.*b
 mulg(c,2.,t3);                                               //    t3 = c*2.
 add(t2,t3,t4);                                        //    t4 = (a+2.*b)+c*2.0;
 copy(t4,d);                                            //    d = (a+2.*b)+c*2.0;
```

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

Remarque 2 *Il est toujours possible de créer des classes intermédiaires pour des opérations prédéfinies afin obtenir se code générer :*

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

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

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

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

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

3.2 les classes RNM

Nous définissons des classes tableaux à un, deux ou trois indices avec ou sans allocation. Ces tableaux est défini par un pointeur sur les valeurs et par la forme de l'indice (class `ShapeOfArray`) qui donne la taille, le pas entre de valeur et le tableau suivant de même type (ces deux données supplémentaire permettent extraire des colonnes ou des lignes de matrice, ...).
La version est dans le fichier « tar compressé » `http://www.ann.jussieu/~hecht/ftp/InfoSciO4/RNM.tar.gz`.
Nous voulons faire les opérations classiques sur `A, C,D` tableaux de type `KN<R>, KNM<R>`, ou `KNMK<R>` comme par exemple :

```
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.

Régle *: Toutes les classes (* `KN_<R>, KNM_<R> , KNMK_<R>` *) se terminant par un _ sont des classes sans aucune allocation mémoire (*`new` *) ni libération (*`delete[]`*, donc elle ne travaillent que sur des tableaux préexistant comme dans des passages de paramètres, ou bien comme pour donner un nom à une portion de tableau, de matrice, ... C'est a dire que ces classes sont des sortes de référence (*`&`*).*

```
#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,'.');                 //    la 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);               //    la 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.

3.3 Exemple d'utilisation

```
using namespace std;

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


         //    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;                                              //    produit matrice vecteur
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;
}
```

3.4 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 :

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

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.

3.4.1 Gradient conjugué préconditionné

Listing 5 *(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);}
};
```

3.4.2 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 6 *(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;
}
```

3.4.3 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
```

Exercice 5

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

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

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

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

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

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

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

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

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

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

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

.

Q1 ecrire l'algorithme mathematiquement, avec des matrices

Q2 Trancrire l'algorithme

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

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

4 Méthodes d'éléments finis P_1 Lagrange

4.1 Le problème et l'algorithme

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

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

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 $\mathcal{T}_h$).

$$V_h = \{v \in \mathcal{C}^0(\Omega_h)/\forall K \in \mathcal{T}_h, v|_K \in P_1(K)\} \tag{2}$$

Après utilisation de la formule de Green, et en multipliant par v, le problème peut alors s'écrire :

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.

$$\int_\Omega \nabla u_h \nabla v_h = \int_\Omega f v_h, \quad \forall v_h \in V_{0h}, \tag{3}$$

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

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 :

$$\int_K f \approx \frac{aire_K}{3} \sum_{i=1}^{3} f(q_K^i) \tag{4}$$

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 le vecteur de $\mathbb{R}^{N_s}$ associé à u et tel que $u = \sum_{i=1,N_s} U_i w^i$.
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 :

$$\nabla w^i|_K = H_K^i = \frac{(q^j - q^k)^\perp}{2\, aire_K} \tag{5}$$

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 programme d'éléments finis sans matrice**

1. *On peut calculer* $u_i = \int_\Omega w^i f$ *en utilisant la formule 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 :*

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

Pour finir, on initialiserons le gradient conjugué par :

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

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 7 — *(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 8 *(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 9 *(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 10 *(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 3 *Les trois champs* `(x,y,lab)` *d'un sommet sont initialisés par* `(0.,0.,0)` *par défaut, car les constructeurs sans paramètres des classes de base sont appelés dans ce cas.*

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 (5).

Listing 11 *(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 4 *La construction effective du triangle n'est pas réalisée par un constructeur, mais par la fonction* `set`*. Cette fonction est appelée une seule fois pour chaque triangle, au moment de la lecture du maillage (voir plus bas la classe* `Mesh`*).*

Remarque 5 *Les opérateurs d'entrée-sortie ne sont pas définis, car il y a des pointeurs dans la classe qui ne sont pas alloués dans cette classe, et qui ne sont que des liens sur les trois sommets du triangle (voir également la classe* `Mesh`*).*

Regardons maintenant comment utiliser cette classe :

Listing 12 *(utilisation de la classe Triangle)*

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

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

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 13 *(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 6 *Les remarques 4 et 5 restent également valables pour la classe* `BoundaryEdge`.

La classe `BoundaryEdge` permet les opérations suivantes :

Listing 14 *(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 15 *(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); //    lecture du fichier def. par filename

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

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

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

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

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

Listing 16 *(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];          //   alloc mémoire - tab des triangles
  vertices = new Vertex[nv];                //   alloc mémoire - tab des sommets
  bedges = new BoundaryEdge[neb]; //   alloc 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 du triangle
    area += triangles[i].area;}        //    calcul de l'aire totale du maillage

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

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

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

Listing 17 *(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 18 *(sfemMatPleine.cpp)*

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

R g(const R2 & P){return P.x*P.x+2*P.y*P.y;}            //    boundary condition
R f(const R2 & ){return -6;}                    //    right hand side  -Δg = -2 - 4

void savesplot(const Mesh & Th,char * fileplotname,KN_<R> &x)
{
  ofstream plot(fileplotname);
  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;
}

             //    un class matrice diagonale pour le preconditionneaur diagonal
template <class R>
class MatDiag: VirtualMatrice<R> { public:
  const KN_<R> d;                                   //    Adr du Vecteur diagonal
  typedef typename  VirtualMatrice<R>::plusAx plusAx;

  MatDiag(KN_<R> dd) :d(dd) {}

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

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

int main(int argc , char** argv )
{
  Mesh Th(argc>1? argv[1] : "c30.msh");
  int N=Th.nv;
  KN<R> b(Th.nv);                                   //    b[i] = ∫_Ω f w_i ~ Σ_{K∈T_h} f(G_K) ∫_K w_i
  KNM<R> A(N,N);              //    Matrice pleine stupide mais pour l'exemple
  b=0;                                                    //    second membre
```

```
  A =0;                                                          //    matrice
                                  //    Assemblage de la matrice et du second membre
  for (int k=0;k<Th.nt;k++)
    {
      Triangle & K(Th[k]);
      R2 sA(K[0]),sB(K[1]),sC(K[2]);
      R2 gw[3]={K.H(0),K.H(1),K.H(2)};
      R2 G_K((sA+sB+sC)/3.);
      R intfgkwi= f(G_K)*K.area/3;                          //     ∫_K fw_i = f(G_K)|K|/3
      for (int il=0;il<3;++il)
        {
          int i=Th(K[il]);                //    numero du sommet il du triangle K
          b[i] += intfgkwi;                     //    Assemblage du second membre
          for (int jl=0;jl<3;++jl)                 //    Assemblage de la matrice
            {
              int j=Th(K[jl]);            //    numero du sommet jl du triangle K
              R a_K_ij= (gw[il],gw[jl])*K.area;
              A(i,j) += a_K_ij;
            }
        }
    }

  R tgv = 1e30;              //    tres grand valeur pour la penalisation exacte.

  KN<R> x(N),xe(N);
  x=0;           //    donnee initial, il faut mettre les conditions aux limites.

                       //    Prise en compte des condition au limite de Dirichlet
                                                  //    par penalisation exact.
  KN<R> D1(N);        //    un vecteur pour stoker la 1/diagonale de la matrice A
  for (int i=0;i<Th.nv;i++)
    {
      Vertex & Si(Th(i));
      xe[i]=g(Si);
      if (Si.onGamma())
        {
          A(i,i) = tgv;
          b[i]= tgv*g(Si);
          x[i] = g(Si);             //    la donne initial verifie les CL (mieux)
        }
      D1[i]=1./A(i,i);                   //    Construction du vecteur 1/diagonal
    }

  MatDiag<R> C(D1);                     //    Matrice de preconditionnement (Dii)^-1
  bool converge=GradienConjugue(A,C, b,x,N,1e-10);
  if (converge)
    {                                                        //    a file for gnuplot
      savesplot(Th,"x.splot",x);
      savesplot(Th,"xe.splot",xe);
      xe -= x;
      cout << " diff x-xe :  min =" << xe.min() << " , max = " << xe.max() << endl;
      ofstream fsol("x.sol");
      fsol << x;

    }
  else
    {
      cerr << "Erreur no converge du gradient conjuge " << endl;
      return 1;                 //    pour que la commande unix retour une erreur
    }
  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++ sfemMatPleine.cpp -o sfemGC
#  execution
./sfemMatPleine
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
```

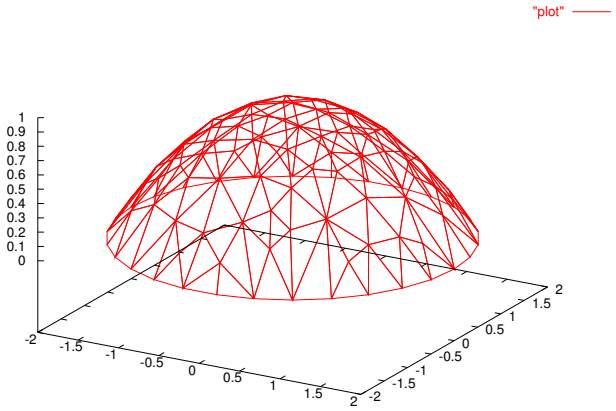

FIG. 1 – le dessin fait par gnuplot

5 Chaînes et Chaînages

5.1 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.

5.2 Construction de l'image réciproque d'une fonction

On va montrer comment construire l'image réciproque d'une fonction F. Pour simplifier l'exposé, nous supposerons que F est une fonction entière de $[0,n]$ dans $[0,m]$ et que ses valeurs sont stockées dans un tableau. Le lecteur pourra changer les bornes du domaine de définition ou de l'image sans grand problème.
Voici une méthode simple et efficace pour construire $F^{-1}(i)$ pour de nombreux i dans $[0,n]$, quand n et m sont des entiers raisonnables. Pour chaque valeur $j \in Im\, F \subset [0,m]$, nous allons construire la liste de ses antécédents. Pour cela nous utiliserons deux tableaux : `int head_F[m]` contenant les "têtes de listes" et `int next_F[n]` contenant la liste des éléments des $F^{-1}(i)$. Plus précisément, si i_1, i_2,...,$i_p \in [0,n]$, avec $p \geq 1$, sont les antécédents de j, `head_F[j]`$=i_p$, `next_F[`i_p`]`$=i_{p-1}$, `next_F[`i_{p-1}`]`$=i_{p-2}$, ..., `next_F[`i_2`]`$=i_1$ et `next_F[`i_1`]`$=$ -1 (pour terminer la chaîne).
L'algorithme est découpé en deux parties : l'une décrivant la construction des tableaux `next_F` et `head_F`, l'autre décrivant la manière de parcourir la liste des antécédents.

Algorithme 3 — **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 6 *Le pourquoi est laissé en exercice.*

5.3 Construction des arêtes d'un maillage

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

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

}
```

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

$$\texttt{nbe} = \texttt{nt} + \texttt{nv} + \texttt{nb\_de\_trous} - \texttt{nb\_composantes\_connexes}, \tag{6}$$

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

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

Cet algorithme trivial est bien trop cher dès que le maillage a plus de 500 sommets (plus de 1.000.000 opérations). Pour le rendre de l'ordre du nombre d'arêtes, on va remplacer la boucle sur l'ensemble des arêtes construites par une boucle sur l'ensemble des arêtes ayant le même plus petit numéro de sommet. Dans un maillage raisonnable, le nombre d'arêtes incidentes sur un sommet est petit, disons de l'ordre de six, le gain est donc important : nous obtiendrons ainsi un algorithme en $3 \times nt$.
Pour mettre cette idée en oeuvre, nous allons utiliser l'algorithme de parcours de l'image réciproque de la fonction qui à une arête associe le plus petit numéro de ses sommets. Autrement dit, avec les notations de la section précédente, l'image par l'application F d'une arête sera le minimum des numéros de ses deux sommets. De plus, la construction et l'utilisation des listes, c'est-à-dire les étapes 1 et 2 de l'algorithme 3 seront simultanées.

Algorithme 5 — **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 7 *Il est possible de modifier l'algorithme précédent en supprimant le tableau* `next_edge` *et en stockant les chaînages dans* `arete[i][0]`*, mais à la fin, il faut faire une boucle de plus sur les sommets pour reconstruire* `arete[.][0]`*.*

Exercice 8 *Construire le tableau* `adj` *d'entier de taille* `3 nt` *qui donne pour l'arete* `i` *du triangle* `k`*.*

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

5.4 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 5.3), `i=Th(k,j)` est le numéro global du sommet `j` $\in [0,3[$ de l'élément `k`. L'application F qu'on va considérer associe à un couple (k,j) la valeur `i=Th(k,j)`. Ainsi, l'ensemble des numéros des triangles contenant un sommet `i` sera donné par les premières composantes des antécédents de `i`.
On va utiliser à nouveau l'algorithme 3, 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 :

$$\texttt{p} \longrightarrow (\texttt{k},\texttt{j}) = (\texttt{k} = \texttt{p}/\texttt{m}, \texttt{j} = \texttt{p}\%\texttt{m}). \tag{7}$$

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

Algorithme 6

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

5.5 Construction de la structure d'une matrice morse

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

[1] 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`.

5.5.1 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 :

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

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
```

5.5.2 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 6 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 7

Construction de la structure d'une matrice morse

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

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

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

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

Au passage, nous avons utilisé la fonction `HeapSort` qui implémente un petit algorithme de tri, présenté dans [Knuth-1975], qui a la propriété d'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.

6 Différentiation automatique

Les dérivées d'une fonction décrite par son implémentation numérique peuvent être calculées automatiquement et exactement par ordinateur, en utilisant la différentiation automatique (DA). La fonctionnalité de DA est très utile dans les programmes qui calculent la sensibilité par rapport aux paramètres et, en particulier, dans les programmes d'optimisation et de design.

6.1 Le mode direct

L'idée de la méthode directe est de différencier chaque ligne du code qui définit la fonction. Les différentes méthodes de DA se distinguent essentiellement par l'implémentation de ce principe de base (cf. [**?**]).
L'exemple suivant illustre la mise en œuvre de cette idée simple.

Exemple : Considérons la fonction $J(u)$, donnée par l'expression analytique suivante :

$$J(u) \quad \text{avec} \quad \begin{aligned} x &= u - 1/u \\ y &= x + \log(u) \\ J &= x + y. \end{aligned}$$

Nous nous proposons de calculer automatiquement sa dérivée $J'(u)$ par rapport à la variable u, au point $u = 2.3$.

Méthode : Dans le programme de calcul de $J(u)$, chaque ligne de code sera précédée par son expression différentiée (avant et non après à cause des instructions du type $x = 2 * x + 1$) :

$$
\begin{aligned}
dx &= du + du/(u * u) \\
\mathbf{x} &= \mathbf{u} - \mathbf{1/u} \\
dy &= dx + du/u \\
\mathbf{y} &= \mathbf{x} + \log(\mathbf{u}) \\
dJ &= dx + dy \\
\mathbf{J} &= \mathbf{x} + \mathbf{y}
\end{aligned}
$$

Ainsi avons nous associé à toute variable (par exemple x) une variable supplémentaire, sa différentielle (dx). La différentielle devient la dérivée seulement une fois qu'on a spécifié la variable de dérivation. La dérivée (dx/du) est obtenue en initialisant toutes les différentielles à zéro en début de programme sauf la différentielle de la variable de dérivation (ex du) que l'on initialise à 1.
La valeur de la dérivée $J'(u)|_{u=2.3}$ est donc obtenue en exécutant le programme ci-dessus avec les valeurs initiales suivantes : $u = 2.3, du = 1$ et $dx = dy = 0$.

Remarque 7

Les langages C,C++, FORTRAN... ont la notion de constante. *Donc si l'on sait que, par exemple, $a = 2$ dans tous le programme et que a ne changera pas, on n'est pas obligé de lui associer une différentielle. Par exemple, la fonction C*

```
float mul(const int a, float u)
{  float x ; x=a*u ; return x ;}
```

se dérive comme suit :

```
float dmul(const int a, float u, float du)
{  float x,dx ; dx = a*du ; x=a*u ; return dx ;}
```

Les structures de contrôle (boucles et tests) présentes dans le code de définition de la fonction seront traitées de manière similaire. En effet, une instruction de test de type *if* où a est prédéfini,

```
y = a ;
if ( u>0) x = u ;
  else    x = 2*u ;
J=x+y ;
```

peut être vue comme deux programmes distincts :
• le premier calcule

```
y=a ; x=u ; J=x+y ;
```

et avec la DA, il doit retourner

```
dy=0 ; y=a ; dx=du ; x=u ; dJ=dx+dy ; J=x+y ;
```

• le deuxième calcule

```
y=a ; x=2*u ; J=x+y ;
```

et avec la DA, il doit retourner

```
dy=0 ; y=a ; dx=2*du ; x=2*u ; dJ=dx+dy ; J=x+y ;
```

Les deux programmes sont réunis naturellement sous la forme d'un unique programme

```
dy=0 ; y=a ;
if (u>0) {dx=du ; x=u ;}
    else {dx=2*du ; x=2*u ;}
dJ=dx+dy ; J=x+y ;
```

Le même traitement est appliqué à une structure de type boucle :

```
x=0 ;
for( int i=1 ; i<= 3 ; i++) x=x+i/u ;
cout << x << endl ;
```

qui, en fait, calcule

```
x=0 ; x=x+1/u ; x=x+2/u ; x=x+3/u ; cout << x << endl ;
```

Pour la DA, il va falloir calculer

```
dx=0 ;x=0 ;
dx=dx-du/(u*u) ; x=x+1/u ;
dx=dx-2*du/(u*u) ; x=x+2/u ;
dx=dx-3*du/(u*u) ; x=x+3/u ;
cout << x <<'\t'<< dx << endl ;
```

ce qui est réalisé simplement par l'instruction :

```
dx=0 ;x=0 ;
for( int i=1 ; i<= 3 ; i++)
    { dx=dx-i*du/(u*u) ; x=x+i/u ;}
cout << x <<'\t'<< dx << endl ;
```

Limitations :
• Si dans les exemples précédents la variable booléene qui sert de test dans l'instruction `if` et/ou les limites de variation du compteur de la boucle `for` dépendent de u, l'implémentation décrite plus haut n'est plus adaptée. Il faut remarquer que dans ces cas, la fonction définie par ce type de programme n'est plus différentiable par rapport à la variable u, mais est différentiable à droite et à gauche et les dérivées calculées comme ci-dessus sont justes.

Exercice 10 ‖ Ecrire le programme qui dérive une boucle `while`.

• Il existe des fonctions non-différentiables pour des valeurs particulières de la variable (par exemple, $\sqrt{u}$ pour $u = 0$). Dans ce cas, toute tentative de différentiation automatique pour ces valeurs conduit à des erreurs d'exécution du type *overflow* ou *NaN* (not a number).

6.2 Fonctions de plusieurs variables

La méthode de DA reste essentiellement la même quand la fonction dépend de plusieurs variables. Considérons l'application $(u_1, u_2) \rightarrow J(u_1, u_2)$ définie par le programme suivant :

$$y_1 = l_1(u_1, u_2) \quad y_2 = l_2(u_1, u_2, y_1) \quad J = l_3(u_1, u_2, y_1, y_2) \tag{8}$$

En utilisant la méthode de DA décrite précédemment, nous obtenons :

$$\begin{aligned}
&dy_1 = \partial_{u_1} l_1(u_1, u_2)dx_1 + \partial_{u_2} l_1(u_1, u_2)dx_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2}) \\
&dy_2 = \partial_{u_1} l_2 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_2 dy_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}) \\
&dJ = \partial_{u_1} l_3 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_3 dy_1 + \partial_{y_2} l_3 dy_2 \\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2}).
\end{aligned}$$

Pour obtenir dJ pour (u_1, u_2) donnés, il faut exécuter le programme deux fois : une première fois avec $dx_1 = 1, dx_2 = 0$, ensuite, une deuxième fois, avec $dx_1 = 0, dx_2 = 1$.
Une meilleure solution est de dupliquer les lignes $dy_i = \cdots$ et les évaluer successivement pour $dx_i = \delta_{ij}$. Le programme correspondant :

$$\begin{aligned}
&d1y_1 = \partial_{u_1} l_1(u_1, u_2)d1x_1 + \partial_{u_2} l_1(u_1, u_2)d1x_2 \\
&d2y_1 = \partial_{u_1} l_1(u_1, u_2)d2x_1 + \partial_{u_2} l_1(u_1, u_2)d2x_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2}) \\
&d1y_2 = \partial_{u_1} l_2 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_2 d1y_1 \\
&d2y_2 = \partial_{u_1} l_2 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_2 d2y_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}) \\
&d1J = \partial_{u_1} l_3 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_3 d1y_1 + \partial_{y_2} l_3 d1y_2 \\
&d2J = \partial_{u_1} l_3 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_3 d2y_1 + \partial_{y_2} l_3 d2y_2 \\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2})
\end{aligned}$$

sera exécuté pour les valeurs initiales : $d1x_1 = 1,\ d1x2 = 0,\ d2x_1 = 0,\ d2x_2 = 1.$

6.3 Une bibliothèque de classes pour le mode direct

Il existe plusieurs implémentations de la différentiation automatique, les plus connues étant **Adol-C, ADIFOR** et **Odyssee**. Leur utilisation implique une période d'apprentissage importante ce qui les rend peu accessibles aux programmeurs débutants. C'est la raison pour laquelle nous présentons ici une implémentation utilisant le mode direct qui est très simple d'utilisation, quoique moins performante que le mode inverse.
On utilise la technique de surcharge d'opérateurs (voir chapitre **??**). Toutefois, cette technique n'est efficace pour le calcul de dérivées partielles que si le nombre de variables de dérivation est inférieur à la cinquantaine. Pour une implémentation similaire en FORTRAN 90, voir Makinen [**?**].

6.4 Principe de programmation

Considérons la même fonction

$$J(u) \quad \text{avec} \quad x = 2u(u+1)$$

$$y = x + \sin(u)$$
$$J = x * y.$$

Par rapport à la méthode décrite précédemment, nous allons remplacer chaque variable par un tableau de dimension deux, qui va stocker la valeur de la variable et la valeur de sa différentielle. Le programme modifié s'écrit :

```
float y[2],x[2],u[2];
                                                    //    dx = 2 u du + 2 du (u+1)
      x[1] = 2 * u[0] * u[1] + 2 * u[1] * (u[0] + 1);
                                                          //    x = 2 u (u+1)
      x[0] = 2 * u[0] * (u[0] + 1);
      y[1] = x[1] + cos(u[0])*u[1];
      y[0] = x[0] + sin(u[0]);
      J[1] = x[1] * y[0] + x[0] * y[1];
                                                                //    J = x * y
      J[0] = x[0] * y[0];
```

L'étape suivante de l'implémentation (voir [**?**], chapitre 4) consiste à créer une classe C++ qui contient comme données membres les tableaux introduits plus haut. Les opérateurs d'algèbre linéaire classiques seront redéfinis à l'intérieur de la classe pour prendre en compte la structure particulière des données membres. Par exemple, les opérateurs d'addition et multiplication sont définis comme suit :

6.5 Implémentation comme bibliothèque C++

Afin de rendre possible le calcul des dérivées partielles (N variables), l'implémentation C++ de la DA va utiliser la classe suivante :

Listing 19 *(la classe `ddouble`)*

```
  #include <iostream>
using namespace std;


struct ddouble {
 double val,dval;
 ddouble(double x,double dx): val(x),dval(dx) {}
 ddouble(double x): val(x),dval(0) {}

};


inline ostream &  operator<<(ostream & f,const  ddouble & a)
{ f << a.val << " ( d = "<< a.dval << ") "; return f;}

inline ddouble operator+(const ddouble & a,const ddouble & b)
   { return ddouble(a.val+b.val,a.dval+b.dval);}

inline ddouble operator+(const double & a,const ddouble & b)
   { return ddouble(a+b.val,b.dval);}

inline ddouble operator*(const ddouble & a,const ddouble & b)
   { return ddouble(a.val*b.val,a.dval*b.val+a.val*b.dval);}
inline ddouble operator*(const double & a,const ddouble & b)
   { return ddouble(a*b.val,a*b.dval);}
```

```
inline ddouble operator/(const ddouble & a,const ddouble & b)
   { return ddouble(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
inline ddouble operator/(const double & a,const ddouble & b)
   { return ddouble(a/b.val,(-a*b.dval)/(b.val*b.val));}

inline ddouble operator-(const ddouble & a,const ddouble & b)
   { return ddouble(a.val-b.val,a.dval-b.dval);}

inline ddouble operator-(const ddouble & a)
   { return ddouble(-a.val,-a.dval);}
inline ddouble sin(const ddouble & a)
   { return ddouble(sin(a.val),a.dval*cos(a.val));}
inline ddouble cos(const ddouble & a)
   { return ddouble(cos(a.val),-a.dval*sin(a.val));}
inline ddouble exp(const ddouble & a)
   { return ddouble(exp(a.val),a.dval*exp(a.val));}
inline ddouble fabs(const ddouble & a)
   { return (a.val > 0)? a : -a;}
inline  bool operator<(const ddouble & a ,const ddouble & b)
   { return a.val < b.val;}
```

---

voila un petit exemple d'utilisation de ces classes

```
template<typename R> R f(R x)
{
   R y=(x*x+1.);
   return y*y;
}



int main()
{
        ddouble x(2,1);
        cout << f(2.0) << " x = 2.0,  (x*x+1)^2 " << endl;
        cout << f(ddouble(2,1)) << "2 (2x) (x*x+1)   " << endl;
        return 0;
}
```

Mais de faite l'utilisation des (templates) permet de faire une utilisation recursive.

```
#include <iostream>
using namespace std;


template<class R> struct Diff {
 R val,dval;
 Diff(R x,R dx): val(x),dval(dx) {}
 Diff(R x): val(x),dval(0) {}
};


template<class R>  ostream &  operator<<(ostream & f,const  Diff<R> & a)
{ f << a.val << " ( d = "<< a.dval << ") "; return f;}
```

```
template<class R> Diff<R> operator+(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val+b.val,a.dval+b.dval);}

template<class R> Diff<R> operator+(const R & a,const Diff<R> & b)
   { return Diff<R>(a+b.val,b.dval);}

template<class R> Diff<R> operator*(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val*b.val,a.dval*b.val+a.val*b.dval);}
template<class R> Diff<R> operator*(const R & a,const Diff<R> & b)
   { return Diff<R>(a*b.val,a*b.dval);}

template<class R> Diff<R> operator/(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
template<class R> Diff<R> operator/(const R & a,const Diff<R> & b)
   { return Diff<R>(a/b.val,(-a*b.dval)/(b.val*b.val));}

template<class R> Diff<R> operator-(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val-b.val,a.dval-b.dval);}

template<class R> Diff<R> operator-(const Diff<R> & a)
   { return Diff<R>(-a.val,-a.dval);}
template<class R>Diff<R> sin(const Diff<R> & a)
   { return Diff<R>(sin(a.val),a.dval*cos(a.val));}
template<class R> Diff<R> cos(const Diff<R> & a)
   { return Diff<R>(cos(a.val),-a.dval*sin(a.val));}
template<class R> Diff<R> exp(const Diff<R> & a)
   { return Diff<R>(exp(a.val),a.dval*exp(a.val));}
template<class R> Diff<R> fabs(const Diff<R> & a)
   { return (a.val > 0)? a : -a;}
template<class R> bool operator<(const Diff<R> & a ,const Diff<R> & b)
   { return a.val < b.val;}
template<class R> bool operator<(const Diff<R> & a ,const R & b)
   { return a.val < b;}
```

Si l'on veut calculer l'a racine d'une equation $f(x) = y$

```
template<typename R> R f(R x)
{
   return x*x;
}

template<typename R> R Newtow(R y,R x)
{
                                                    //    solve: f(x) -y = 0
                                                  //    x -= (f(x)-y) / df(x);
   while (1) {
   Diff<R> fff=f(Diff<R>(x,1));
   R ff=fff.val;
   R dff=fff.dval;
   cout << ff << endl;
   x = x - (ff-y)/dff;
   if (fabs(ff-y) < 1e-10) break;
   }
   return x;
}
```

Maintenant, il est aussi possible de re-différencier automatiquement l'algorithme de Newtow. Pour cela ; il suffit d'ecrit

```
int main()
{
    typedef double R;
    cout << " -- newtow (2)  = " << Newtow(2.0,1.) << endl;
    Diff<R> y(2.,1) ,  x0(1.,0.);                                    //    donne
    Diff<R> xe(sqrt(2.), 0.5/sqrt(2.));                        //    solution exact
    cout << "\n -- x = Newton " << y << " ," << x0 << ")"<<  endl;
    Diff<R> x= Newtow(y,x0);
    cout << " x = " << x << " ==  " <<  xe << " = xe    "  << endl;
    return 0;
}
```

Les resultats sont

```
[guest-rocq-135177:~/work/coursCEA/autodiff] hecht% ./a.out
 -- newtow (2)  = 1
2.25
2.00694
2.00001
2
1.41421

 -- x = Newton 2 ( d = 1)  ,1 ( d = 0) )
1 ( d = 0)
2.25 ( d = 1.5)
2.00694 ( d = 1.02315)
2.00001 ( d = 1.00004)
2 ( d = 1)
 x = 1.41421 ( d = 0.353553)  ==  1.41421 ( d = 0.353553)  = xe
```

7 Algèbre de fonctions

Bien souvent, nous aimerions pouvoir considérer des fonctions comme des données classiques, afin de construire une nouvelle fonction en utilisant les opérations classiques `+,-,*,/` ... Par exemple, la fonction $f(x,y) = cos(x) + sin(y)$.
Dans un premier temps, nous allons voir comment l'on peut construire et manipuler en `C++`l'algèbre des fonctions $\mathcal{C}^\infty$, définies sur $\mathbb{R}$ à valeurs dans $\mathbb{R}$.

7.1 Version de base

Une fonction est modélisée par une classe (`Cvirt`) qui a juste l'opérateur `()()` virtuel.
Pour chaque type de fonction, on construira une classe qui dérivera de la classe `Cvirt`. Comme tous les opérateurs doivent être définis dans une classe, une fonction sera définie par une classe `Cfonc` qui contient un pointeur sur une fonction virtuelle `Cvirt`. Cette classe contiendra l'opérateur « fonction » d'évaluation, ainsi qu'une classe pour construire les opérateurs binaires classiques. Cette dernière classe contiendra deux pointeurs sur des `Cvirt` qui correspondent aux membres de droite et gauche de l'opérateur et la fonction de $\mathbb{R}^2$ à valeur de $\mathbb{R}$ pour définir l'opérateur.

Listing 20 *(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 cinq opérateurs binaires
 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 définie par un arbre de classes qui peut être représenté 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^∞

Maintenant, nous voulons aussi pouvoir dériver les fonctions. Il faut faire attention, car si la classe contient la dérivée de la fonction, il va y avoir implicitement une récursivité infinie, les fonctions étant indéfiniment dérivables. Donc, pour que la classe fonctionne, il faut prévoir un processus à deux niveaux : l'un qui peut construire la fonction dérivée, et l'autre qui évalue la fonction dérivée et qui la construira si nécessaire.
Donc, la classe `CVirt` contient deux fonctions virtuelles :

```
virtual float operator () (float) = 0;
virtual CVirt *de () {return zero;}
```

La première est la fonction d'évaluation et la seconde calcule la fonction dérivée et retourne la fonction nulle par défaut. Bien entendu, nous stockons la fonction dérivée (`CVirt`) qui ne sera construite que si l'on utilise la dérivée de la fonction.
Nous introduisons aussi les fonctions de $\mathbb{R}^2$ à valeurs dans $\mathbb{R}$, qui seront modélisées dans la classe `Fonction2`.
Toutes ces classe sont définies dans le fichier d'en-tête suivant :

Listing 21 *(fonction.hpp)*

```
#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));                  //    Création à partir d'une fonction C
  Fonction (R x);                                          //    Fonction constante
  Fonction (const Fonction& f) : p(f.p) {}                 //   Constructeur par
copie
  Fonction d () {return p->d ();}                                    //    Dérivée
  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));              //    Création à partir d'une fonction C
  Fonction2 (R x);                                         //    Fonction constante
  Fonction2 (const Fonction2& f) : p(f.p) {}          //   Constructeur par copie
```

```
  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
```

Maintenant, prenons l'exemple d'une fonction `Monome` qui sera utilisée pour construire les fonctions polynômes. Pour construire effectivement ces fonctions, nous définissons la classe `CMonome` pour modéliser $x \mapsto cx^n$ et la classe `CMonome2` pour modéliser $x \mapsto cx^{n_1}y^{n_2}$.

Listing 22 *(les classes CMonome et CMonome2)*

```
class CMonome : public CVirt {
R c; int n;
public:
CMonome (R cc = 0, int k = 0) : c (cc), n (k) {}
R operator () (R x);                                          //    return cx^n
CVirt *de () {return n? new CMonome (c * n, n - 1) : zero;}
};
Function Function::monome (R x, int k) {return new CMonome (x, k);}

class CMonome2 : public CVirt2 {
R c; int n1, n2;
public:
CMonome2 (R cc = 0, int k1 = 0, int k2 = 0) : c (cc), n1 (k1), n2 (k2) {}
R operator () (R x, R y);                                     //    return cx^{n1}y^{n2}
CVirt2 *dex () {return n1? new CMonome2 (c * n1, n1 - 1, n2) : zero;}
CVirt2 *dey () {return n2? new CMonome2 (c * n2, n1, n2 - 1) : zero;}
};
Function2 Function2::monome (R x, int k, int l)
{return new CMonome2 (x, k, l);}
```

En utilisant exactement la même technique, nous pouvons construire les classes suivantes :

CFunc une fonction C++de prototype `R (*)(R)` .

CComp la composition de deux fonctions f, g comme $(x) \to f(g(x))$.

CComb la composition de trois fonctions f, g, h comme $(x) \to f(g(x), h(x))$.

CFunc2 une fonction C++de prototype `R (*)(R,R)`.

CComp2 la composition de f, g comme $(x, y) \to f(g(x, y))$.

CComb2 la composition de trois fonctions f, g, h comme $(x, y) \to f(g(x, y), h(x, y))$

Pour finir, nous indiquons juste la définition des fonctions globales usuelles :

```
CVirt *CVirt::zero = new CMonome;
CVirt2 *CVirt2::zero = new CMonome2;

Function::Function (R (*f) (R)) : p(new CFunc(f)) {}
Function::Function (R x) : p(new CMonome(x)) {}
Function Function::operator() (Function f) {return new CComp (p, f.p);}
Function2 Function::operator() (Function2 f) {return new CComp2 (p, f.p);}

Function2::Function2 (R (*f) (R, R)) : p(new CFunc2(f)) {}
Function2::Function2 (R x) : p(new CMonome2(x)) {}
Function Function2::operator() (Function f, Function g)
{return new CComb (f.p, g.p, p);}
Function2 Function2::operator() (Function2 f, Function2 g)
{return new CComb2 (f.p, g.p, p);}

static R add (R x, R y) {return x + y;}
static R sub (R x, R y) {return x - y;}

Function Log = new CFunc1(log,new CMonome1(1,-1));
Function Chs = new CMonome (-1., 1);
Function Identity = new CMonome (-1., 1);

Function2 CoordinateX = new CMonome2 (1., 1, 0);                    //    x
Function2 CoordinateY = new CMonome2 (1., 0, 1);                    //    y
Function2 One2 = new CMonome2 (1., 0, 0);                           //    1
Function2 Add = new CFunc2 (add, Function2(1.), Function2(1.));
Function2 Sub = new CFunc2 (sub, Function2(1.), Function2(-1.));
Function2 Mul = new CMonome2 (1., 1, 1);
Function2 Div = new CMonome2 (1., 1, -1);
                                                    //     pow(x,y) = x^y = e^{log(x)y}
Function2 Pow = new CFunc2 (pow, CoordinateY*Pow(CoordinateX,
CoordinateY-One2)),Log(CoordinateX)*Pow);
```

Avec ces définitions, la construction des fonctions classiques devient très simple ; par exemple, pour construire les fonctions sin, cos il suffit d'écrire :

```
Function Cos(cos),Sin(sin);
Cos.setd(-Sin);                              //    définit la dérivée de cos
Sin.setd(Cos);                               //    définit la dérivée de sin
Function d4thCos=Cos.d().d().d().d();      //    construit la dérivée quatrième
```

Références

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

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

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

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

[1] I. Danaila, F. hecht, O. Pironneau : *Simulation numérique en* `C++` Dunod, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Campione et Walrath-1996] M. Campione and K. Walrath The Java Tutorial : Object-Oriented Programming for the Internet, Addison-Wesley, 1996.
Voir aussi Integrating Native Code and Java Programs. `http://java.sun.com/nav/read/Tutorial/native1.1/index.html`.

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

[Casteyde-2003] Christian Casteyde Cours de C/C++ `http://casteyde.christian.free.fr/cpp/cours`