Des petites classes C++ pour tableaux avec des opérations vectorielles

Frédéric Hecht
Université Paris 6/ INRIA

mailto:Frederic.hecht@upmc.fr
http://www.ann.jussieu.fr/~hecht

31 janvier 2006

Table des matières

1 Introduction

Dans ce chapitre nous allons nous intéresser à la construction de classes pour modéliser des tableaux à 1,2 où 3 indices, soit $\mathbb{K}^n$,$\mathbb{K}^{n,m}$, et $\mathbb{K}^{n,m,k}$ où $\mathbb{K}^{n,m}$ est l'espace matrice $n \times m$ à coefficient dans $\mathbb{K}$ et ou $\mathbb{K}^{n,m,k}$ est l'espace de tenseurs d'ordre 3 à coefficient dans $\mathbb{K}$.
Donc nous allons pleinement utiliser les patrons `<<template>>` pour construire ces trois classes, où $\mathbb{K}$ sera le type paramétré. Nous n'utilisons pas les classes `valarray` du `C++` qui ne fonctionne pas encore très bien, de plus les classes définies dans le package `Blitz++` `http://www.oonumerics.org/blitz` sont trops lourde pour être utiliser pour faire l'enseignement.
Nous commencerons sur une version didactique, nous verrons la version complète accessible sur le web
où dans le fichier « tar compressé » `ftp://ann.jussieu.inria.fr/~hecht/ftp/cpp/RNM.tar.gz`.
Dans ce chapitre, nous allons nous intéresser à la construction de classes pour modéliser des tableaux à 1,2 ou 3 indices, soit $\mathbb{K}^n$,$\mathbb{K}^{n,m}$, et $\mathbb{K}^{n,m,k}$ où $\mathbb{K}^{n,m}$ est l'espace des matrices $n \times m$ à coefficient dans $\mathbb{K}$ et ou $\mathbb{K}^{n,m,k}$ est l'espace des tenseurs d'ordre 3 à coefficient dans $\mathbb{K}$.
Donc nous allons pleinement utiliser les patrons « template »pour construire ces trois classes.
Nous commencerons sur une version didactique, nous verrons la version complète qui est dans `ftp://ann.jussieu.inria.fr/~hecht/ftp/cpp/source/RNM` où dans le fichier « tar compressé » `ftp://ann.jussieu.inria.fr/~hecht/ftp/cpp/RNM.tar.gz`.

2 Version simple

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

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

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

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

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

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

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

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

};
```

Il faut faire attention dans les paramètres d'entrée et de sortie des opérateurs. Il est clair que l'on ne veut pas travailler sur des copies, mais la sortie est forcément un objet et non une référence sur un objet car il faut allouer de la mémoire

dans l'opérateur et si l'on retourne une référence aucun destructeur ne sera appelé et donc cette mémoire ne sera jamais libérée.

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

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

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

Pour la multiplication par un scalaire à droite on a :

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

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

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

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

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

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

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

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

REMARQUE 1 *Il est toujours possible de créer des classes intermédiaires pour des opérations prédéfinies afin obtenir se code générer :*

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

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

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

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

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

3 Les classes tableaux

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

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

```
A[j] = c;
```

Pour des raisons évidentes nous ne sommes pas allés plus loin que des combinaisons linéaires à plus de deux termes. Toutes ces opérations sont faites sans aucune allocation et avec une seule boucle.
De plus nous avons défini, les tableaux 1,2 ou 3 indices, il est possible extraire une partie d'un tableau, une ligne ou une colonne.

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

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

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

4 Exemple d'utilisation

```
namespace std

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


using namespace std;
//   definition des 6 types de base des tableaux a 1,2 et 3 parametres
typedef double R;
typedef KN<R> Rn;
typedef KN_<R> Rn_;
typedef KNM<R> Rnm;
typedef KNM_<R> Rnm_;
typedef KNMK<R> Rnmk;
typedef KNMK_<R> Rnmk_;
R f(int i){return i;}
R g(int i){return -i;}
int main()
{
```

```
const int n= 8;
cout << "Hello World, this is RNM use!" << endl << endl;
Rn a(n,f),b(n),c(n);
b =a;
c=5;
b *= c;

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

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


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

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

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

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

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


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

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

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

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

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


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

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

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

 //  copie du sous tableaux

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

  return 0;
}
```

5 Un resolution de systéme linéaire avec le gradient conjugué

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

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

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

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

Voila comment écrire un gradient conjugue avec ces classes.

5.1 Gradient conjugué préconditionné

Listing 1 *(GC.hpp)*

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

```
        x += ro *h;
        g += ro *Ah; //  plus besoin de Ah, on utilise avec Cg optimisation
        Cg = C*g;
        R g2p=g2;
        g2 = (Cg,g);
        if (g2 < reps2) {
           cout << iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
           return 1;//  ok
           }
        R gamma = g2/g2p;
        h *= gamma;
        h -= Cg;  //   h = -Cg * gamma* h
      }
   cout << " Non convergence de la méthode du gradient conjugue " <<endl;
   return 0;
}
template <class R>
class MatriceIdentite: VirtualMatrice<R> { public:
 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);}
};
```

5.2 Teste du gradient conjugué

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

Listing 2 *(GradConjugue.cpp)*

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


#include "gmres.hpp"

 using namespace std;

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

// #include "sfem.hpp"
#define KN_CHECK
#include "RNM.hpp"
#include "GC.hpp"
```

```
typedef double R;

class MatriceLaplacien1D: VirtualMatrice<R> { public:
 typedef  VirtualMatrice<R>::plusAx plusAx;
 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];
  }

// R dot(const KN<double> &a,const KN<double> & b) {return (a,b);}
// R norm(KN<double> &a) { return sqrt( (a,a) );}

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;
  R eps = 1e-6;
  int i50=50;
  Rnm H(i50,i50);
```

```
  x=0;
//   int res=GMRES<Rnm,Rn,Rnm,Rnm,R>(A,x,b,C,H,i50,n,eps);
  int res=GMRES(A,x,b,C,H,i50,n,eps);
  cout << " GMRES " << n << " " << eps << " " << res << endl;
  s = A*x;
  cout << " solution : A*x= "  << s << endl;
  }
  {
  cout << "GradienConjugue laplacien 1D  par la identity  " << endl;
  int N=100;
  Rn b(N),x(N);
  R h= 1./(N-1);
  b= h;
  b[0]=0;
  b[N-1]=0;
  x=0;
  R t0=CPUtime();
  GradienConjugue(MatriceLaplacien1D(),MatriceIdentite<R>() ,b,x,N,1e-5);
  cout << " Temps cpu = " << CPUtime() - t0<< "s" << endl;
  R err=0;
  for (int i=0;i<N;i++)
    {
    R xx=i*h;
    err= max(fabs(x[i]- (xx*(1-xx)/2)),err);
    }
  cout << "Fin err=" << err << endl;
  }
  return 0;
}
```

5.3 Sortie du teste

```
10x10   :
          10  -1  -1  -1  -1  -1  -1  -1  -1  -1
          -1  11  -1  -1  -1  -1  -1  -1  -1  -1
          -1  -1  18  -1  -1  -1  -1  -1  -1  -1
          -1  -1  -1  37  -1  -1  -1  -1  -1  -1
          -1  -1  -1  -1  74  -1  -1  -1  -1  -1
          -1  -1  -1  -1  -1 135  -1  -1  -1  -1
          -1  -1  -1  -1  -1  -1 226  -1  -1  -1
          -1  -1  -1  -1  -1  -1  -1 353  -1  -1
          -1  -1  -1  -1  -1  -1  -1  -1 522  -1
          -1  -1  -1  -1  -1  -1  -1  -1  -1 739
        GradienConjugue preconditionne par la diagonale
6  ro = 0.990712 ||g||^2 = 1.4253e-24
 solution : A*x= 10     :
1.60635e-15       1       2       3       4       5       6       7       8       9

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

 GMRES 5 4.11771e-08 0
 solution : A*x= 10     :
-3.60463e-10      1       2       3       4       5       6       7       8       9
```

```
GradienConjugue laplacien 1D  par la identity
48  ro = 0.00505051 ||g||^2 = 1.55006e-32
 Temps cpu = 0s
Fin err=5.55112e-17
```

Références

[1] G. Buzzi-Ferrari : Scientific C++ Addison-Wesley 1993

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

[3] B. Lucquin, O. Pironneau : Introduction au calcul scientifique. Masson (1996).

[4] J. Shapiro : A C++ Toolkit, Prentice Hall 1991.

[5] B. Stroustrup : The C++ Programming Language Addison Wesley 1991