Algorithmes et techniques avancées de programmation

extrait du livre : *I. Danaila, F. Hecht, O. Pironneau, Simulation numérique en C++, Dunod, 2003*

1.1 La méthode du gradient conjugué

Nous allons montrer dans cette section comment utiliser les classes `RNM` pour programmer l'algorithme du gradient conjugué préconditionné (GCP) pour résoudre le système linéaire $Ax = b$, avec $A \in \mathbb{R}^{n\times n}$ une matrice symétrique, définie positive.

Rappelons d'abord le principe et les principales caractéristiques de l'algorithme du gradient conjugué (GC) (pour les développements théoriques de la méthode, nous renvoyons à [**?**, **?**] par exemple).

Si $a, b \in \mathbb{R}^n$ sont deux vecteurs colonnes, on va noter $(a, b) = a^T b$ leur produit scalaire euclidien. Considérons la fonctionnelle quadratique $E : \mathbb{R}^n \to \mathbb{R}$

$$E(x) = \frac{1}{2}(Ax, x) - (b, x), \tag{1.1}$$

dont le gradient vaut $\nabla E(x) = (Ax - b)$. Le minimum de cette fonctionnelle est atteint pour le point $\bar{x} \in \mathbb{R}^n$ qui annule $\nabla E(x)$, donc vérifiant $A\bar{x} = b$.

La méthode du gradient conjugué fait partie des *méthodes de descente*, qui ont comme principe commun la recherche de $\bar{x}$ suivant le procédé itératif

$$x^0 \text{ donné}, \;\; x^{i+1} = x^i + \rho^i d^i, \tag{1.2}$$

avec $d^i \in \mathbb{R}^n$ la *direction de descente* et $\rho^i \in \mathbb{R}$ le *pas de descente*. Comment définir les paramètres de la méthode de descente ? La réponse est simple pour le choix du pas de descente, car, une fois d^i et x^i fixés, on peut calculer facilement un pas de descente *optimal*

$$\rho^i = -\frac{(Ax^i - b, d^i)}{(Ad^i, d^i)} \tag{1.3}$$

qui réalise le minimum de la fonctionnelle E dans la direction choisie, ou, plus formellement, le minimum de la fonction réelle, de variable réelle : $\rho \mapsto E(x^i + \rho d^i)$.

Quant à la direction de descente, plusieurs choix sont possibles, ce qui nous permet de distinguer :

- la méthode de relaxation : $d^i = e^i$, avec $(e^i)_{1\le i\le n}$ la base canonique de $\mathbb{R}^n$ (on peut démontrer qu'il s'agit en fait de la méthode itérative de Gauss-Seidel) ;

- la méthode du gradient : $d^i = g^i = \nabla E(x^i)$ (basée sur l'observation que la valeur de la fonctionnelle diminue le plus rapidement suivant la direction du gradient) ;
- la méthode du gradient conjugué : $d^i = h^i$, avec les directions de descente *conjuguées* par rapport à la matrice A, c'est-à-dire $(Ah^i, h^j) = 0, i \neq j$.

L'efficacité de la méthode du gradient conjugué réside dans ses propriétés remarquables, énoncées ici pour l'itération i :

- les directions de descente h^i sont construites de telle manière que les gradients $g^i = Ax^i - b$ soient tous orthogonaux entre eux, *i.e.* $(g^k, g^j) = 0, \forall 0 \leq k < j \leq i$ (ce n'est pas le cas dans la méthode du gradient, où seulement deux gradients successifs sont orthogonaux) ;
- x^i réalise le minimum de la fonctionnelle E sur l'espace vectoriel $x^0 + Vect\,\{g^0, g^1, \ldots, g^{i-1}\}$, de dimension i ; quand $i = n$, cet espace vectoriel sera exactement $\mathbb{R}^n$, ce qui prouve que l'algorithme converge en n itérations au plus ;
- d'un point de vue pratique, les directions de descente h^i sont faciles à calculer, à partir des gradients g^i, et la méthode nécessite le stockage de seulement trois vecteurs supplémentaires (voir programme plus loin).

Une formulation *optimisée* de l'algorithme (GC) est la suivante :

Algorithme 1.1

Le gradient conjugué pour résoudre le système linéaire $Ax = b$:
soient $x^0 \in \mathbb{R}^n$ la valeur initiale (donnée) et ε la précision (fixée)
$g^0 = Ax^0 - b$
$h^0 = -g^0$
– pour $i = 0$ à n

$\rho = -\dfrac{(g^i, h^i)}{(h^i, Ah^i)}$	*(pas optimal, cf. 1.3)*
$x^{i+1} = x^i + \rho h^i$	*(avancement suivant h^i, cf. 1.2)*
$g^{i+1} = g^i + \rho Ah^i$	*(calcul itératif du gradient)*
$\gamma = \dfrac{(g^{i+1}, g^{i+1})}{(g^i, g^i)}$	*(paramètre assurant $(h^{i+1}, h^i) = 0$)*
$h^{i+1} = -g^{i+1} + \gamma h^i$	*(calcul itératif du la direction de descente)*
si $(g^{i+1}, g^{i+1}) < \varepsilon$ *stop*	

Si l'algorithme (GC) peut être considéré comme une méthode exacte (la convergence étant assurée en n itérations au plus), il est généralement utilisé comme méthode itérative en s'attendant à ce qu'il converge plus rapidement. Pour accélérer la vitesse de convergence, on peut *préconditionner* le système linéaire. L'idée ici est de multiplier le système initial par une matrice C, dite de *préconditionnement*, et résoudre le nouveau système ($CAx = Cb$) par le (GC) classique.

En toute rigueur, pour appliquer le (GC), la matrice du système doit être symétrique, définie positive, condition qui n'est pas assurée même si la matrice C est symétrique, définie positive (le produit de deux matrices symétriques, définie positives, n'est pas nécessairement une matrice symétrique, définie positive). Cette observation nous amène à faire les manipulations algébriques suivantes pour trouver une formulation simple de l'algorithme (GCP).

Soit $U \in \mathbb{R}^{n\times n}$ une matrice inversible et $z = Ux$ un changement de variable. La fonctionnelle E donnée par (1.1) devient :

$$E(z) = \frac{1}{2}(AU^{-1}z, U^{-1}z) - (b, U^{-1}z) = \frac{1}{2}(U^{-T}AU^{-1}z, z) - (U^{-T}b, z), \qquad (1.4)$$

avec $U^{-T} = (U^{-1})^T = (U^T)^{-1}$. Cette fonctionnelle correspond au système linéaire $A^*z = b^*$ de matrice $A^* = U^{-T}AU^{-1}$ symétrique, définie positive, et de second membre $b^* = U^{-T}b$. Nous pouvons appliquer le (GC) pour ce système, et les principales quantités à calculer deviennent (cf. algorithme 1.1) :

$$
\begin{array}{lcl}
g^0 = U^{-T}AU^{-1}z^0 - U^{-T}b & \Leftrightarrow & \underbrace{U^T g^0}_{G^0} = Ax^0 - b \\
g^{i+1} = g^i + \rho U^{-T}AU^{-1}h^i & \Leftrightarrow & G^{i+1} = G^i + \rho A \underbrace{U^{-1}h^i}_{H^i} = G^i + \rho AH^i \\
h^0 = -g^0 & \Leftrightarrow & H^0 = -\underbrace{U^{-1}U^{-T}}_{C} G^0 = -CG^0 \\
\rho = -\dfrac{(g^i, h^i)}{(h^i, Ah^i)} & \Leftrightarrow & \rho = -\dfrac{(U^{-T}G^i, UH^i)}{(UH^i, U^{-T}AU^{-1}UH^i)} = -\dfrac{(G^i, H^i)}{(H^i, AH^i)} \\
z^{i+1} = z^i + \rho h^i & \Leftrightarrow & x^{i+1} = x^i + \rho H^i \\
\gamma = \dfrac{(g^{i+1}, g^{i+1})}{(g^i, g^i)} & \Leftrightarrow & \gamma = \dfrac{(U^{-T}G^{i+1}, U^{-T}G^{i+1})}{(U^{-T}G^i, U^{-T}G^i)} = \dfrac{(G^{i+1}, CG^{i+1})}{(G^i, CG^i)} \\
h^{i+1} = -g^{i+1} + \gamma h^i & \Leftrightarrow & H^{i+1} = -CG^{i+1} + \gamma H^i
\end{array}
\tag{1.5}
$$

Observons que la matrice $C = U^{-1}U^{-T}$ est symétrique, définie positive, et que c'est la seule matrice intervenant dans les expressions précédentes (la matrice U est utilisée juste pour construire l'algorithme). Si on note $(a, b)_C = (Ca, b)$ le produit scalaire induit par C sur $\mathbb{R}^n$, l'algorithme (GCP) s'écrit sous la forme :

Algorithme 1.2

Le gradient conjugué préconditionné par une matrice C, symétrique définie positive :
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*

Il est clair de ce qui précède que le choix « idéal » pour la matrice C, serait $C = A^{-1}$, l'inverse de A ; dans ce cas, évidemment, l'utilisation de la méthode du gradient conjugué deviendrait inutile. En pratique, plus la matrice de préconditionnement est « proche" » de A^{-1}, meilleure sera la convergence de l'algorithme. En même temps, la matrice C doit être la plus simple et la plus creuse possible, ce qui impose comme choix les plus faciles

- $C = I$, la matrice identité (algorithme sans préconditionnement) ;
- $C = D^{-1}$, l'inverse de la partie diagonale de A.

La programmation de l'algorithme (GCP) en utilisant les classes `RNM` est donnée dans le fichier d'en-tête suivant (l'application utilisant ce programme doit simplement l'insérer avec la directive de préprocesseur `#include "GC.hpp"`) :

Listing 1.1 *(GC.hpp)*

```
//    exemple de programmation du gradient conjugué préconditionné
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);
   KN<R> & Cg(Ah);                          //    on utilise Ah pour stocker Cg
   g = A*x;
   g -= b;                                                    //    g = Ax-b
   Cg = C*g;                                //    gradient preconditionné
   h =-Cg;

   R g2 = (Cg,g);
   if (g2 < eps*eps)                        //    solution déjà convergée
   {cout << "iter=0" << " ||g||^2 = " << g2 << endl;
    return 1;}

   R reps2 = eps*eps*g2;                                  //    epsilon relatif

   for (int iter=0;iter<=nbitermax;iter++)
     {
       Ah = A*h;
       R ro =  - (g,h)/ (h,Ah);
       x += ro *h;
       g += ro *Ah;                  //    plus besoin de Ah, on va stocker Cg
       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, convergence
          }
       R gamma = g2/g2p;
       h *= gamma;
       h -= Cg;
     }

   cout << " Non-convergence de la méthode du gradient conjugué" <<endl;
   return 0;
}
```