Université de Picardie Jules Verne
Master 2 AAM
Calcul Scientifique
2019-2020

Méthodes de sous-espaces de Krylov 3

1 Quelques autres méthodes

1.1 Gradient Conjugué Généralisé

Lorsque A est inversible mais non SDP, on peut appliquer la méthode du gradient conjugué au système d'*équations normales*

$$A^T Ax = A^T b.$$

On dispose du résultat suivant :

Lemme 1
Soient $p^0, p^1, \cdots, p^{m-1}$ une base de Krylov de l'espace $K_m(A; r^0)$, $A^T A$ orthogonale, c'est à dire que

$$(A^T Ap^i, p^j) = (Ap^i, Ap^j) = 0 \, i \neq j.$$

Alors le vecteur x^m minimisant le résidu dans l'espace affine $x^0 + K_m(A; r^0)$ est défini par

$$x^m = x^0 + \sum_{i=0}^{m-1} \frac{(r^i, Ap^i)}{(Ap^i, AP^i)} p^i,$$

et l'on peut calculer x^m de proche en proche par la relation de recurrence

$$x^m = x^{m-1} + \frac{(r^{m-1}, Ap^{m-1})}{(Ap^{m-1}, AP^{m-1})} p^{m-1}.$$

L'idée est de calculer le vecteurs de descente courant en utilisant des combinaisons linéaires des vecteurs de descente précédents.

Algorithm 1 CGR

1: **Calculer** $r^0 = b - Ax^0$. **Poser** $p^0 = r^0$
2: **for** $k = 0, 1, \cdots$ **do** jusqu'à convergence
3: **Poser** $\alpha_k = \dfrac{(r^k, Ap^k)}{(Ap^k, Ap^k)}$
4: **Poser** $x^{k+1} = x^k + \alpha_k p^k$
5: **Poser** $r^{k+1} = r^k - \alpha_k Ap^k$
6: **Calculer** $\beta_{i,k} = -\dfrac{(Ar^{k+1}, Ap^k)}{(Ap^k, Ap^k)}$, pour $i = 0, \cdots, k$
7: **Poser** $p^{k+1} = r^{k+1} + \sum_{i=0}^{k} \beta_{i,k} p^k$
8: **end for**

1.2 Gradient Bi-conjugué BCG

L'idée est de construire un processus de projection sur $K_m(A; v^1) = \{v^1, Av^1, \cdots, A^{m-1}v^1\}$ orthogonalement à

$L_m(A; r^0) = \{w^1, A^T w^1, \cdots, (A^T)^{m-1}w^1\}$. Ici $\|v^1\| = \|w^1\| = 1$ et $(v^1, w^1) \neq 0$, on prendra $v^1 = w^1$. On associe au système $Ax = b$ le *système dual* $A^T x^* = b^*$.

Algorithm 2 BCG

1: **Calculer** $r^0 = b - Ax^0$. **Choisir** $r^{*,0}$ tel que $(r^0, r^{*,0}) \neq 0$.
2: **Poser** $p^0 = r^0$, $p^{*,0} = r^{*,0}$.
3: **for** $k = 0, 1, \cdots$ **do** jusqu'à convergence
4: **Poser** $\alpha_k = \dfrac{(r^k, r^{*,k})}{(Ap^k, p^{*,k})}$
5: **Poser** $x^{k+1} = x^k + \alpha_k p^k$
6: **Poser** $r^{k+1} = r^k - \alpha_k A p^k$
7: **Poser** $r^{*,k+1} = r^{*,k} - \alpha_k A^T p^{*,k}$
8: **Calculer** $\beta_k = -\dfrac{(r^{k+1}, r^{*,k+1})}{(r^k, r^{*,k})}$
9: **Poser** $p^{k+1} = r^{k+1} + \beta_k p^k$
10: **Poser** $p^{*,k+1} = r^{*,k+1} + \beta_k p^{*,k}$
11: **end for**

Lemme 2
Les suites de vecteurs (p^k) et $(p^{*,k})$ vérifient

$$(r^i, r^{*,j}) = 0 \text{ et } (Ap^i, p^{*,j}) = 0, i \neq j.$$

Remarque 1
On retrouve CG lorsque $A^T = A$.

1.3 CGS : Conjugate Gradient Squared

Un inconvénient majeur de BCG est l'utilisation de A^T, dans certains cas on ne sait faire que la multiplication Ax, car A est connue seulement implicitement. Des méthodes de bigradient sans utilsation de la transposée (*Transpose Free* ou TF) ont été proposés, telles que TFQMR et BiCGSTAB (voir plus bas). La méthode CGS est due à Sonneveld en 1984 (deux ans avant GMRES) et est donc motivée par éviter l'utilisation de A^T dans BCG. En reliant le résidu initial à r^j et à p^j on peut écrire

$$r^j = \phi_j(A) r^0 \text{et } p^j = \pi_j(A) r^0,$$

où $\phi_J, \pi_j \in \mathbf{P}_j$ et $\phi_j(0) = 1$. Ainsi

$$r^{*,j} = \phi_j(A^T) r^{*,0} \text{et } p^{*,j} = \pi_j(A^T) r^{*,0}.$$

Après un certain nombre certain de calculs de simplification, on obtient

Algorithm 3 CGS

Calculer $r^0 = b - Ax^0$. **Choisir** $r^{*,0}$ tel que $(r^0, r^{*,0}) \neq 0$.
Poser $p^0 = u^0 = r^0$.
for $k = 0, 1, \cdots$ **do** jusqu'à convergence
    Poser $\alpha_k = \dfrac{(r^k, r^{*,0})}{(Ap^k, r^{*,0})}$
    Poser $q^k = u^k - \alpha_k A p^k$
    Poser $x^{k+1} = x^k + \alpha_k (u^k + q^k)$
    Poser $r^{k+1} = r^k - \alpha_k A(u^k + q^k)$
    Calculer $\beta_k = \dfrac{(r^{k+1}, r^{*,0})}{(r^k, r^{*,0})}$
    Poser $u^{k+1} = r^{k+1} + \beta_k q^k$
    Poser $p^{k+1} = u^{k+1} + \beta_k (q^k + \beta_k p^k)$
end for

1.4 BiCGSTAB

Cet algorithme a été introduit par H. van der Vorst en 1992 ; ll s'agit d'une variation de CGS qui permet de limiter la propagation des erreurs d'arrondis et présente une convergence plus rapide et plus régulière. Il repose sur une relation de récurrence du type

$$r^j = \psi_j(A)\phi_j(A)r^0,$$

avec

$$\psi_{j+1}(t) = (1 - \omega_j t)\psi_j(t).$$

Algorithm 4 BiCGSTAB

Calculer $r^0 = b - Ax^0$. **Choisir** $r^{*,0}$ tel que $(r^0, r^{*,0}) \neq 0$.
Poser $p^0 = r^0$
for $k = 0, 1, \cdots$ **do** jusqu'à convergence
    Poser $\alpha_k = \dfrac{(r^k, r^{*,0})}{(Ap^k, r^{*,0})}$
    Poser $s^k = r^k - \alpha_k A p^k$
    Poser $\omega_k = \dfrac{(As^k, s^k)}{(As^k, As^k)}$
    Poser $x^{k+1} = x^k + \alpha_k p^k + \omega_k s^k$
    Poser $r^{k+1} = s^k - \omega_k A s^k$
    Calculer $\beta_k = \dfrac{(r^{k+1}, r^{*,0})}{(r^k, r^{*,0})} \dfrac{\alpha_k}{\omega_k}$
    Poser $p^{k+1} = r^{k+1} + \beta_k (p^k - \omega_k A p^k)$
end for

2 Exercice

Exercice 1 (Trouver l'erreur)

```
m=20;
b=rand(N*N,1);
toll=1.e-11;
x0=zeros(N*N,1);
Maxit=m;
[x,k,resid]=gmres(A,b,m,toll,x0);
lr1=1:max(size(resid));
[x,R]=CG(A,b,x0,toll,Maxit);
lr2=1:max(size(R));
plot(lr1,log10(resid),'-*b',lr2,log10(R),'-r')
legend('GMRES','CG')
```

```
m=20;
b=rand(N*N,1);
toll=1.e-11;
x0=zeros(N*N,1);
Maxit=m;
[x,k,resid]=gmres(A,b,m,toll,x0);
lr1=1:max(size(resid));
[x,R]=CG(A,b,x,toll,Maxit);
lr2=1:max(size(R));
plot(lr1,log10(resid),'-*b',lr2,log10(R),'-r')
legend('GMRES','CG')
```

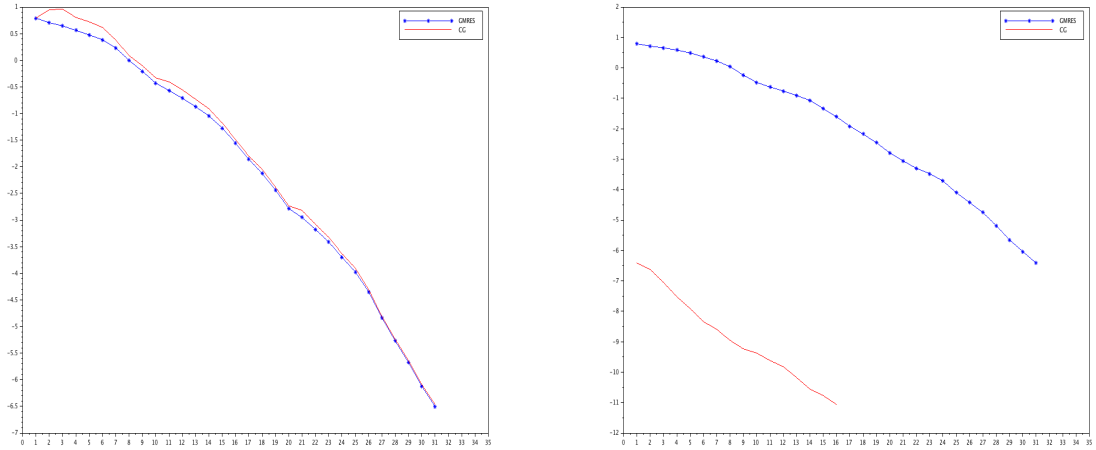

Figure 1: Problème de Dirichlet 2D. Comparaison de CG et de GMRES

Exercice 2 (GMRES versus BiCGSTAB)
On revient au problème de convection diffusion

$$-\Delta u + a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y} = f \quad \text{Dans } \Omega =]0,1[^2, \tag{1}$$

$$u = 0 \qquad \text{sur } \partial\Omega \tag{2}$$

1. Pour le préconditionnement ILU(0), comparer GMRES et BiCGSTAB, pour différentes valeurs de a, b et N. Que remarquez-vous ?

2. Mêmes questions lorsque la dimension du problème est égale à 3.

3 Annexe

3.1 BiCGSTAB

```
function [x, error, iter, flag, residu] = bicgstab(A, x, b, M, max_it, tol)

// -- Iterative template routine --
// Univ.  of Tennessee and Oak Ridge National Laboratory
// October 1, 1993
// Details of this algorithm are described in "Templates for the
// Solution of Linear Systems:  Building Blocks for Iterative
// Methods", Barrett, Berry, Chan, Demmel, Donato, Dongarra,
// Eijkhout, Pozo, Romine, and van der Vorst, SIAM Publications, // 1993.  (ftp netlib2.cs.utk.edu;
cd linalg; get templates.ps).
//
// [x, error, kiter, flag, residu] = bicgstab(A, x, b, M, max_it, tol)
//
// bicgstab.m solves the linear system Ax=b using the
// BiConjugate Gradient Stabilized Method with preconditioning.
//
// input A REAL matrix
// x REAL initial guess vector
// b REAL right hand side vector
// M REAL preconditioner matrix
// max_it INTEGER maximum number of iterations // tol REAL error tolerance //
// output x REAL solution vector
// error REAL error norm
// iter INTEGER number of iterations performed
// flag INTEGER: 0 = solution found to tolerance
// 1 = no convergence given max_it
// -1 = breakdown:  rho = 0
// -2 = breakdown:  omega = 0
// kiter=1;
ndim=max(size(x));
iter = 0; // initialization
flag = 0;

bnrm2 = norm( b );
if ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

r = b - A*x;
error = norm( r ) / bnrm2;
residu=[error]; if ( error < tol ) return, end

   omega = 1.0;
r_tld = r;

for iter = 1:max_it, // begin iteration
```

```
rho = ( r_tld'*r ); // direction vector
if ( rho == 0.0 ) break, end

if ( iter > 1 ),
beta = ( rho/rho_1 )*( alpha/omega );
p = r + beta*( p - omega*v );
else
p = r;
end

p_hat = M \ p;
v = A*p_hat;
alpha = rho / ( r_tld'*v );
s = r - alpha*v;
if ( norm(s) < tol ), // early convergence check
x = x + alpha*p_hat;
resid = norm( s ) / bnrm2;
break;
end
s_hat = M \ s; // stabilizer
t = A*s_hat;
omega = ( t'*s) / ( t'*t );
x = x + alpha*p_hat + omega*s_hat; // update approximation
r = s - omega*t;
error = norm( r ) / bnrm2; // check convergence
residu=[residu error];
if ( error <= tol ), break, end
if ( omega == 0.0 ), break, end
rho_1 = rho;
end
if ( error <= tol | s <= tol ), // converged
if ( s <= tol ),
error = norm(s) / bnrm2;
end
flag = 0;
elseif ( omega == 0.0 ), // breakdown
flag = -2;
elseif ( rho == 0.0 ),
flag = -1;
else // no convergence
flag = 1;
end
kiter=iter;
endfunction
```