MDI224

Interpolation par splines cubiques

Travaux Pratique 2 - Deuxième semestre de 2011

Professeur: Roland BADEAU

Anthony CLERBOUT Casier: 234
Tiago CHEDRAUOI SILVA Casier: 214

Décembre 15, 2011

Table des matières

Table des figures

1 Résolution du système linéaire

1.1 Méthode de relaxation

Nous pouvons exprimer analytiquement la valeur de x_i^{k+1}

$$\begin{aligned} J(x) &= \frac{1}{2}x^T Ax - x^T b \\ &= \frac{1}{2}\sum_i \sum_j x_i^{k+1} x_j^{k+1} a_{ij} - \sum_i x_i^{k+1} b_i \\ \frac{\delta J}{\delta x_i^{k+1}} &= \sum_{j\leq i} x_j^{k+1} a_{ij} + \sum_{j>i} x_j^k a_{ij} - b_i = 0 \\ x_i^{k+1} &= \frac{1}{a_{ij}}(b_i - \sum_{j<i} x_j^{k+1} a_{ij} - \sum_{j>i} x_j^k a_{ij}) \end{aligned}$$

On observe qu'il s'agit de l'expression de x_i^{k+1} dans l'algorithme de Gauss-Seidel et également dans l'algorithme de relaxation avec w=1.

1.2 Méthode du gradient à pas constant

1.2.1 Complexité

A chaque itération on fait : $g = A * x_n - b$ et $xn = x(:, i+1) - beta * g$ Donc : Comme A est une matrice tridiagonal, $A * x_n$ fait 3N multiplications, et $beta * g$ fait N. Donc, on a 4N multiplication a chaque itération pour la méthode du gradient à pas constant.

Pour le méthode de relaxation on fait :$x_i^{k+1} = \frac{1}{a_{ij}}(b_i - \sum_{j<i} x_j^{k+1} a_{ij} - \sum_{j>i} x_j^k a_{ij})$, et cela nous donne aussi 4N multiplications.

1.2.2 Convergence de l'algorithme

On pose $e^{n+1} = x^n - x^*$. On a donc :

$$\begin{aligned} e^{n+1} &= x^{n+1} - x^* \\ &= x^n - \beta(Ax^n - b) - x^* \\ &= (I_n - \beta A)e^n \end{aligned}$$

On sait qu'il existe une base orthonormale de vecteurs propres de A, notons là $(p_i)_{1\leq i\leq N}$ Soit v appartenant à R^n, on pose $v = sum_{i=1}^N v_i.p_i$

On a alors :

$$\begin{aligned}(I_n - \beta A)v &= \sum_{i=1}^{N}(I_n - \beta A)v_i p_i \\ &= \sum_{i=1}^{N}(1 - \beta\lambda_i)v_i p_i\end{aligned}$$

Ainsi :

$$\begin{aligned}||I_n - \beta A||_2^2 &= (\sum_{i=1}^{N}(1 - \beta\lambda_i)v_i p_i, \sum_{j=1}^{N}(1 - \beta\lambda_j)v_j p_j) \\ &= \sum_{i=1}^{N}(1 - \beta\lambda_i^2)v_i^2 \\ &\leq max(1 - \beta\lambda_i)\sum_{j=1}^{N} v_j^2 = (max|1 - \beta\lambda_i|^2 * ||v||_2^2)\end{aligned}$$

Pour tout v appartenant à R^n, on a donc :

$$||e^{n+1}||_2 \leq max|1 - \beta\lambda_i|||e^n||_2$$

On note $\rho(b) = max|1 - \beta\lambda_i|$ et par récurrence, on obtient :

$$||e^n||_2 \leq \rho(B)||e^0||_2$$

On doit donc vérifier que $\rho(B) < 1$!
Pour tout $i \in [1, N]$: :

$$\lambda_N \leq \lambda_i \leq \lambda_1.$$

Donc :

$$1 - \beta\lambda_1 \leq 1 - \beta\lambda_i \leq 1 - \beta\lambda_N$$

Donc :

$$\rho(B) = max|1 - \beta\lambda_i| = max(|1 - \beta\lambda_1|, max|1 - \beta\lambda_N|)$$

L'algorithme converge seulement si $\rho(B) < 1$.
Donc $|1 - \beta\lambda_1| < 1$ est équivalent à :

$$1 - \beta\lambda_1 < 1 \text{ et } \beta\lambda_1 - 1 < 1$$
$$\text{Soit } 0 < \beta < \frac{2}{\lambda_1}.$$

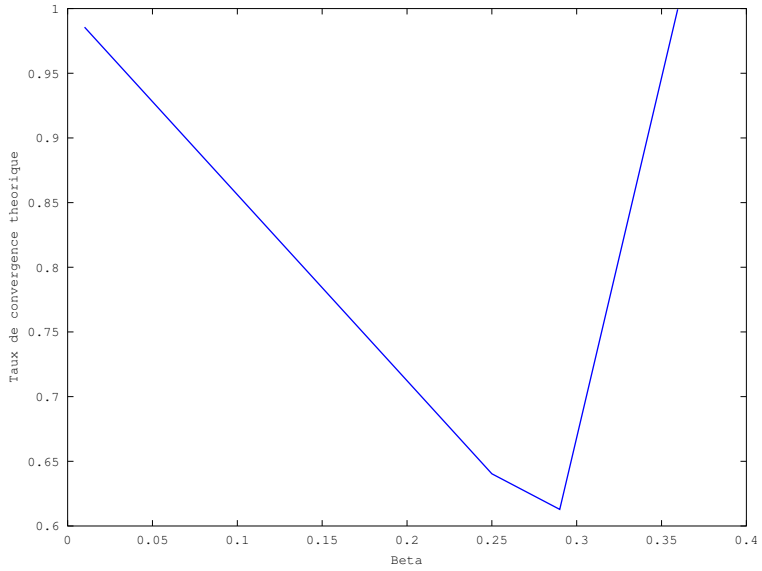


FIG. 1: Taux de convergence de la méthode gradient en fonction de beta : théorique

On a trouvé $\lambda_N = 1.4384$ et $\lambda_1 = 5.5616$, donc $\beta_{optimal} = \frac{2}{7}$. Cette valeur est plus petite que la valeur du taux de convergence de la méthode de relaxation : $w_{optimal} = 1.1$.

1.2.3 Taux de convergence optimal

Premièrement,$\rho(B)$ = taux de convergence et la vitesse de convergence du gradient à pas fixe est $-ln(\rho(B))$.
Comme $\rho(B) = max(1-\beta\lambda_n, \beta\lambda_1 - 1)$. Donc $\rho(B)$ est minimale lorsque $1-\beta\lambda_n = \beta\lambda_1 - 1$
Soit : $\beta = \beta_0 = \frac{2}{\lambda_1+\lambda_n}$

$$\rho(B) \quad = \quad \frac{\lambda_1 - \lambda_n}{\lambda_1 + \lambda_n}$$

D'où quand N est grand, la vitesse de convergence est :

$$-ln(1 - \frac{2\lambda_n}{\lambda_1 + \lambda_n}) = -ln(1 - \frac{2}{1-k}) \quad \text{Avec } k = \frac{\lambda_1}{\lambda_n} \quad .$$

1.2.4 Implémentation

Pour voir la méthode du gradient à pas constant, on a fait le code suivant :

Méthode du gradient à pas constant

```
% % % % % % % % % % % % % % % % % % % % %
% 19/01/11
% Chedraoui Silva,Tiago
% Casier: 214
% CLERBOUT, Anthony
% Casier: 234
% TP2: interpolation par splines
% cubiques partie II
% Description: Gradient a pas constant
% % % % % % % % % % % % % % % % % % % % %

function x = mygradient(A,b,x0,beta,eps
    )

% Entree
% A : matrice
% b : vecteur
% x0: vecteur d'initialisation
% beta: le pas
% esp: critere de convergence


xn=x0;
x=[x0 xn];
i=1;

while (norm (x(:,i+1) - x(:,i))>eps ||
    i==1 ),

  %  Pour la  fonction
  %    J=1/2*xT*A*x-xTb;
  %  Le gradient est donne par
  %    g=Ax-b;

  g = A*xn-b;

  xn = x(:,i+1)-beta*g;
  x = [x xn];
  i++;

end;
```

Losqu'on l'utilise, on trouve comme solution :

$$x = [0.99996\ 1.00002\ 0.99998\ 1.00002\ 0.99996]^T$$

Cette solution n'est pas optimale, mais est beaucoup proché de la solution opimale ($x^* = [1.0\ 1.0\ 1.0\ 1.0\ 1.0]^T$) ;

1.2.5 Variation du taux de convergence

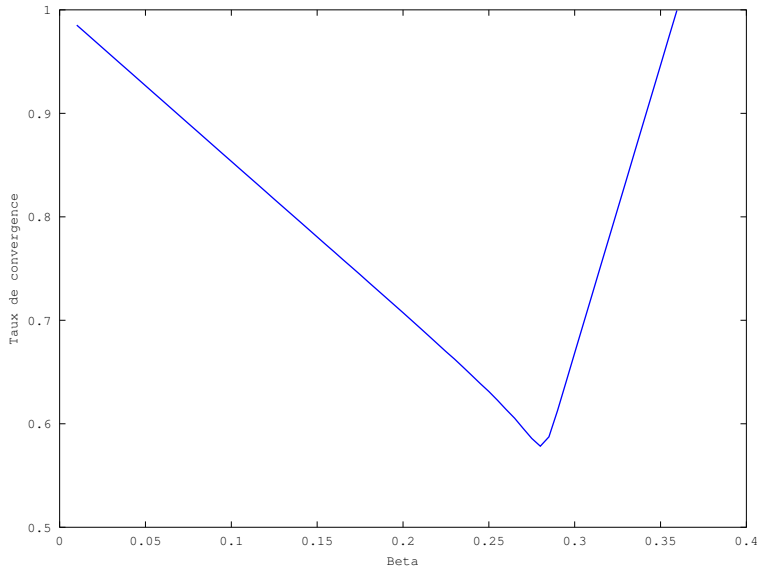


FIG. 2: Taux de convergence de la méthode gradient en fonction de beta : réel

1.3 Méthode du gradient à pas optimal

1.3.1 β qui minimise $J(x_k) - \beta g_k$

Pour trouver β qui minimise $J(x_k) - \beta g_k$, avec $g_{k+1} = -\nabla J(x_k + \beta_k g_k)$.
Soit :

$$\begin{aligned} f'(\beta_k) = 0 &= (\nabla J(x^k + \beta_k g_k), g_k) \\ &= -(g_{k+1}, g_k) = -g_{k+1}^T g_k \end{aligned}$$

On a $g_{k+1} = b - Ax_k - \beta_k A g_k = g_k - \beta_k A g_k$
Donc :

$$\begin{aligned} (g_{k+1}, g_k) = 0 &\Rightarrow (g_k - \beta_k A g_k, g_k) = 0 \\ &\Leftrightarrow (g_k, g_k) - \beta_k (A g_k, g_k) = 0 \\ &\Leftrightarrow \beta_k = \frac{(g_k, g_k)}{(A g_k, g_k)} = \frac{(g_k^T, g_k)}{g_k^T A^T g_k} \end{aligned}$$

1.3.2 Complexité

Pour cette méthode on fait : $g = A * xn - b$, 3N multiplications, et après $beta = g' * g/(g' * A * g)$, 6N multiplications, et enfin $xn = x(:, i+1) - beta * g$, N multiplications. Donc, on a 10N.

Ce coût est plus grand que le coût des valeurs antérieures.

1.3.3 Implémentation

Pour voir la méthode du gradient à pas optimal, on a fait le code suivant :

Méthode du gradient à pas optimal

```
% % % % % % % % % % % % % % % % % % % % %
% 19/12/11
% Chedraoui Silva,Tiago
% Casier: 214
% CLERBOUT, Anthony
% Casier: 234
% TP2: interpolation par splines
% cubiques partie II
% Description: Gradient a pas constant
% % % % % % % % % % % % % % % % % % % % %

function x = gradient_optimal(A,b,x0,
    eps)

% Entree
% A : matrice
% b : vecteur
% x0: vecteur d'initialisation
% esp: critere de convergence

xn=x0;
x=[x0 xn];
i=1;

while (norm (x(:,i+1) - x(:,i))>eps ||
    i==1),

  %  Pour la  fonction
  %    J=1/2*xT*A*x-xTb;
  %  Le gradient est donne par
  %    g=Ax-b;

  g = A*xn-b;

  beta = g'*g/(g'*A*g);

  xn = x(:,i+1)-beta*g;
  x = [x xn];
  i++;

end;
```

1.3.4 Taux de convergence

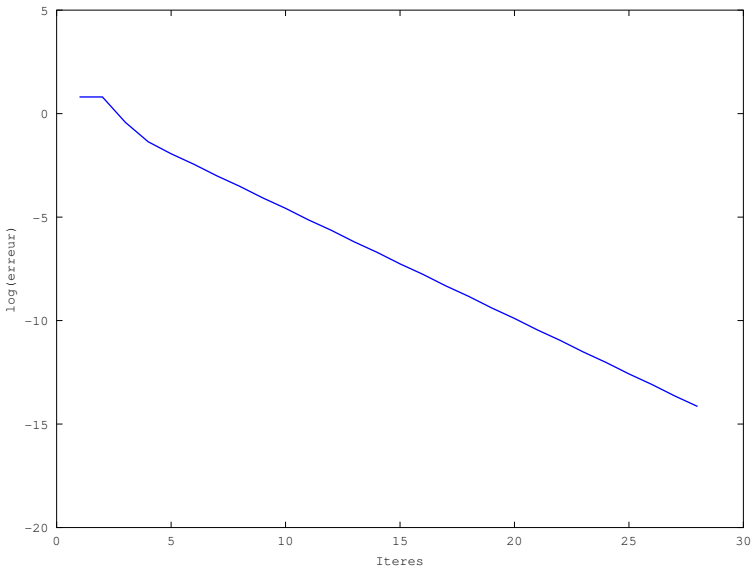


FIG. 3: Convergence de la méthode gradient optimal

En utilisant la fonction polyfit de Matlab, on a trouvé p(x)= -0.547007 x + 1.029611 . Donc, le taux de convergence de l'algorithme trouvée est : -0.54.

L'intéret d'avoir le β optimal est que l'algorithme converge plus rapidement.

1.4 Méthode du gradient conjugué

1.4.1 Complexité

Le nombre d´itérations de cet algorithme est proportionnel à la taille n de la matrice du système. Cela veut dire que le gradient conjugué converge au plus n itérations. Pour une matrice pleine, la méthode demande $2n^3$ opérations mais comme la matrice est creuse, la méthode fera moins que $2n^3$ opérations.

Pour la complexité on fait : $beta = g' * w/(w' * A * w)$; 6N multiplications, $xn = x(:,i+1) - beta * w$; N multiplications, $g = A * xn - b$; 3N multiplications, $a = -g' * A * w/(w' * A * w)$; 7N multiplications, $w = g + a * w$; N multiplications,

Donc : 15 multiplications. Cela est la plus grand des complexités de touts les autres algorithmes, on perd en complexité, mais on gagne grâce au nombre d'itérations qui est le plus petit.

1.4.2 Implémentation

Pour voir la méthode du gradient conjugé, on a fait le code suivant :

Méthode du gradient conjugué

```
% % % % % % % % % % % % % % % % % % % % %
% 19/12/11
% Chedraoui Silva,Tiago
% Casier: 214
% CLERBOUT, Anthony
% Casier: 234
% TP2: interpolation par splines
% cubiques partie II
% Description: Gradient a pas constant
% % % % % % % % % % % % % % % % % % % % %

function x = gradient_conjugue(A,b,x0,
    eps)

xn = x0;
x = [x0 xn];
i = 1;

g = A*xn-b;
w = g;

while (norm (x(:,i+1) - x(:,i))>eps ||
    i==1),

  beta = g'*w/(w'*A*w);

  xn = x(:,i+1)-beta*w;
  x = [x xn];

  g = A*xn-b;
  a = -g'*A*w/(w'*A*w);
  w=g+a*w;

  i++;

end
```

1.4.3 Temps d'exécution

1.5 Application

Méthode	Temps
Gradient conjugué	0.0017250 s
Relaxation	0.0044890
Gradient optimal	0.0044729 s

TAB. 1: Comparatif des temps d´éxécution de différentes méthodes de résolution linéaire

Calcul de spline cubique naturelle

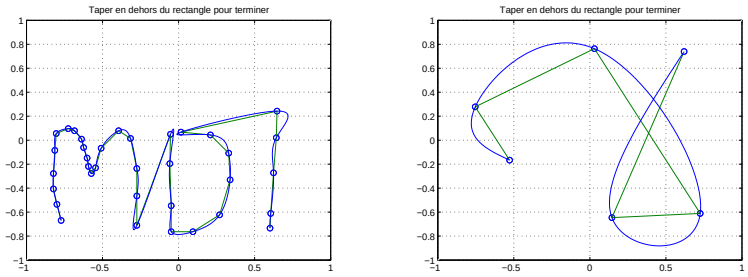
```
%===== tstinvtridiag0.m
% Calcul de spline cubique naturelle
clear
global t t2 t3
Npts=100; t=[0:Npts-1].'/Npts; t2=t.*t;
     t3=t2.*t;
figure(1)
plot([-1;1],[-1;1],'w'); grid
Pk=acqPk; N=length(Pk);
hold on; plot(real(Pk),imag(Pk),'o',
    real(Pk),imag(Pk),'-'); hold off

% Suite des derivees secondes
% Iniatialization matrice A (Ax=b)
A = 4*eye(N-2) + diag(ones(1,N-3),1) +
    diag(ones(1,N-3),-1);
A = A/6;
x0 = zeros(N-2,1);
b=toeplitz(Pk(3:N),[Pk(3) Pk(2) Pk(1)])
    *[1;-2;1];
sol = gradient_conjugue(A,b,x0,10e-6);

% on doit prendre la derniere colonne
    de la matrice sol
dsec = sol(:,size(sol,2));

% la premiere et la dernier derive
    seconde sont 0
dsec = [0;dsec;0];

% Calcul des segments (utiliser global
    t, t2 et t3)
% ...
Skt=[];
for k=1:N-1
Pkt=spline3(Pk(k),Pk(k+1),dsec(k),dsec(
    k+1));
Skt=[Skt;Pkt];
end
hold on; plot(Skt); hold off
```



(a) Example 1 (b) Example 2

FIG. 4: Calcul de spline cubique naturelle