UNIVERSITÉ DE TOULON

ISITV

Compte rendu d'Analyse Numérique 1

Nom de l'auteur
Ingéniérie XXX

2013-2014

Résolution de systèmes linéaires
Factorisation LU

Responsable(s) du stage :
Monsieur M. ERSOY, Maître de Conférences, ISITV

Certainement pleins de fautes de frappes .....
Ce document permet de donner une idée de rapport lorsque vous serez confronté à une méthode numérique et par la même occasion servira de template pour ceux qui veulent faire du latex!!!!

Résumé

On s'intérèsse à la résolution d'un système linéaire $Ax = f$ où $A \in M_n(\mathbb{R})$, $f \in \mathbb{R}^n$ pour $n \in \mathbb{N}$ par le biais de l'algorithme de Thomas. Il s'agit ici de vérifier l'efficacité de cette approche en l'implémentant dans Matlab et de l'appliquer au problème du fléchissement d'une poutre.

Sommaire

1 L'algorithme de Thomas

En générale, les sytèmes linéaires sont issues de problèmes physiques complexes (équations aux dérivées partielles ou équations différentielles) et la dimension de la matrice est souvent très grande. Par conséquent, la résolution peut avoir un coup de calcul plus ou moins important suivant la nature/structure de la matrice. Pour cette raison, il existe différentes méthodes numériques dans la littérature. Nous allons nous intéresser à la factorisation LU

d'une matrice A afin de résoudre un système $Ax = f$. Nous nous limiterons au cas d'une matrice tridiagonale (qui correspond en générale à une discrétisation d'un opérateur du second ordre). Une telle factorisation est toujours possible quand la matrice A est symétrique définie positive.

La factorisation LU consiste à écrire une matrice non-singulière A comme le produit de deux matrices L et U. Dans le cas d'une matrice tridiagonale, les matrices L et U se présentent sous la forme suivante :

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_1 & 1 & 0 & \dots & 0 \\ 0 & l_2 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & l_{n-1} & 1 \end{pmatrix} \text{ et } U = \begin{pmatrix} d_1 & u_2 & 0 & \dots & 0 \\ 0 & d_2 & u_3 & \dots & 0 \\ 0 & 0 & d_3 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & u_n \\ 0 & 0 & \dots & 0 & d_n \end{pmatrix} .$$

Remarque 1.1 *Nous noterons dans la suite* $l := (l_i)_{1 \leqslant i \leqslant n-1}$, $d = (d_i)_{1 \leqslant i \leqslant n}$ *et* $u = (u_i)_{2 \leqslant i \leqslant n}$ *les coefficients des matrices* L *et* U *en vue de faire un post traitement vectoriel.*

1.1 Factorisation LU d'une matrice tridiagonale

En notant $b = (b_i)_{1 \leqslant i \leqslant n}$ les coefficients diagonaux,$a = (a_i)_{1 \leqslant i \leqslant n-1}$ les coefficients sous-diagonaux et $c = (c_i)_{2 \leqslant i \leqslant n}$ les coefficients sur-diagonaux de la matrice A, la factorisation LU s'obtient par l'algorithme suivant :

input : $a \in \mathbb{R}^{n-1}$, $b \in \mathbb{R}^n$, $c \in \mathbb{R}^{n-1}$
output: $l \in \mathbb{R}^{n-1}$,$d \in \mathbb{R}^n$

Initialisation :
$n \leftarrow \text{length}(b)$
$d_1 \leftarrow b_1$

for $i \leftarrow 2$ **to** n **do**
  $l_{i-1} \leftarrow \frac{a_{i-1}}{d_{i-1}}$
  $u_i \leftarrow c_i$
  $d_i \leftarrow b_i - l_{i-1}u_i$
end

Algorithm 1: Factorisation LU d'une matrice tridiagonale

Remarque 1.2 *Il est facile de voir que dans ce cas* $u = c$ *et par conséquent l'algorithme renvoit seulement* l *et* d.

La résolution de ce système en exploitant la factorisant LU se résume ainsi en deux étapes :

- la première est une étape de descente qui consiste a résoudre le système triangulaire inférieure $Ly = f$,

– la seconde est une étape de remontée qui consiste a résoudre le système triangulaire supérieure $Ux = y$.

Ces deux étapes (diviser pour mieux régner) permettent considérablement de réduire le cout de calcul (donc le temps de calcul). A titre d'exemple, le cout dans le cas d'une matrice carrée de dimension n est de $O(n^2)$ pour la factorisation LU et de $O(n^3)$ pour l'élimination de Gauss.

1.2 L'étape de descente

La résolution du système triangulaire inférieure $Ly = f$ s'obtient par l'algorithme suivant :

input : $l \in \mathbb{R}^{n-1}$, $f \in \mathbb{R}^n$
output: $y \in \mathbb{R}^n$

Initialisation :
$n \leftarrow \text{length}(f)$
$y_1 \leftarrow f_1$

for $i \leftarrow 2$ **to** n **do**
  $y_i \leftarrow f_i - l_{i-1} y_{i-1}$
end

Algorithm 2: algorithme de descente

1.3 L'étape de remontée

La résolution du système triangulaire supérieure $Ux = y$ s'obtient par l'algorithme suivant :

input : $d \in \mathbb{R}^n$, $c \in \mathbb{R}^{n-1}$, $y \in \mathbb{R}^n$
output: $x \in \mathbb{R}^n$

Initialisation :
$n \leftarrow \text{length}(y)$
$x_n \leftarrow \frac{y_n}{d_n}$

for $i \leftarrow n-1$ **to** a **do**
  $x_i \leftarrow \frac{y_i - c_{i+1} x_{i+1}}{d_i}$
end

Algorithm 3: algorithme de remonté

2 Applications numériques

Dans la section précédente, nous avons rappelé le principe de la factorisation LU ainsi que les algorithmes. Nous allons dans un premier temps valider nos codes pour la résolution d'un système linéaire dans un cas simple puis nous l'appliquerons à la résolution d'un problème

plus complexe, à savoir le fléchissement d'une poutre (dont nous rappellerons la discrétisation par différences finies en section 2.2).

2.1 Validation de la méthode

Il s'agit dans cette section qu'aucune erreur n'a été commise lors de la transcription des algorithmes 1, 2 et 3 aux fonctions `facto`, `desc` et `mont` que vous trouverez en Annexe B.

2.1.1 Validation de la fonction `facto`

Nous allons d'abord vérifier que la fonction `facto` de l'Annexe B.1 renvoie la factorisation LU d'une matrice A. On considère la matrice

$$A = \begin{pmatrix} 1 & 4 & 0 \\ 2 & 10 & 5 \\ 0 & 6 & 18 \end{pmatrix}$$

qui admet la décomposition LU suivante :

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 3 & 0 \end{pmatrix} \text{ et } U = \begin{pmatrix} 1 & 4 & 0 \\ 0 & 2 & 5 \\ 0 & 0 & 3 \end{pmatrix} .$$

On vérifie alors rapidement avec le code matlab suivant

```
clear all
clc
% On choisit L et U
L = [1 0 0; 2 1 0; 0 3 1];
U = [1 4 0; 0 2 5; 0 0 3];
%
A = L*U;

a = [2 6];
b = [1 10 18];
c = [4 5];

[l,d] = facto(a,b,c)
```

que la fonction `facto` (c.f. Annexe B.1) renvoie

```
 l =

   2   3

d =

   1   2   3
```

qui correspond à la décomposition LU de A.

2.1.2 Validation de la fonction `desc`

On s'intérèsse a présent à la validation de la fonction `desc` (voir Annexe B.2) qui correspond à l'algorithme 2 en vue de résoudre le système $Ax = f$ avec $f = \begin{pmatrix} 3 \\ 3 \\ 3 \end{pmatrix}$ et A définit comme précédemment. Dans ce cas, en ajoutant les lignes de codes

```
 f = [3;3;3];
y = desc(l,f)
```

on trouve

```
 y =

    3    -3    12
```

qui est bien solution du système $Ly = f$.

2.1.3 Validation de la fonction `mont`

Dans les deux dernières sous-sections, nous avons factorisé la matrice

$$A = \begin{pmatrix} 1 & 4 & 0 \\ 2 & 10 & 5 \\ 0 & 6 & 18 \end{pmatrix}$$

par LU où

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 3 & 0 \end{pmatrix} \text{ et } U = \begin{pmatrix} 1 & 4 & 0 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{pmatrix} .$$

Nous avons ensuite calculer la solution du système $Ly = f$. A présent, on valide l'étape de remontée en utilisant la ligne de code suivant :

```
x = mont(c,d,y)
inv(A)*f % On verifie
```

qui nous retourne

```
 x =

   49.0000  -11.5000    4.0000

ans =
```

```
 49.0000
-11.5000
  4.0000
```

On peut donc désormais utiliser l'algorithme de Thomas pour la résolution de système linéaire en abordant un problème un peu plus délicat.

2.2 Application au fléchissement d'une poutre

On considère une poutre de longueur $L := 1$, étirée selon son axe, soumise à une force transversale $f(x)\,dx$ par unité de longueur dx et fixée à ses extrémités. Le déplacement $u(x)$ du point d'abscisse x est solution du problème aux limites :

$$\begin{cases} -u''(x) + c(x)u(x) = f(x), \quad x \in (0,1), \\ u(0) = 0, \\ u(1) = 0. \end{cases} \tag{1}$$

Afin de résoudre l'équation (1), nous allons procéder à une discrétisation par différences finies, i.e. nous allons construire une approximation u_i de $u(x_i)$ où $x_i = ih$, $0 \leqslant i \leqslant n+1$ est un noeud du maillage de l'interval $[0,1]$ où n represente le nombre de noeuds et $h = \frac{1}{n+1}$ le pas de la subdivision (ou pas d'espace).

En supposant u "suffisamment" régulière et h "petit", par développement de Taylor au voisinage du points x_i, on peut approcher la dérivée seconde $u''(x_i)$ par

$$u''(x_i) \approx \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{h^2}$$

à l'ordre 2.

Dès lors l'équation (1) s'écrit sous la forme d'un système linéaire (voir TD pour plus de détails)

$$A_h U_h = F_h \tag{2}$$

où

$$A_h = \frac{1}{h^2} \begin{pmatrix} 2 + c(x_1)h^2 & -1 & 0 & \dots & 0 \\ -1 & 2 + c(x_2)h^2 & -1 & \dots & 0 \\ 0 & -1 & 2 + c(x_3)h^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & 0 & -1 & 2 + c(x_n)h^2 \end{pmatrix},$$

$$U_h = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \text{ et } F_h = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}.$$

Il s'agit alors d'appliquer l'algorithme de Thomas au système (2). On obtient alors l'algorithme de résolution suivant :

input : $n \in \mathbb{N}^*$
output: $U_h \in \mathbb{R}^n$

Initialisation :
$h \leftarrow \frac{1}{(n+1)}$
Meshpoints :
for $i \leftarrow 0$ **to** $n+1$ **do**
  $x_i \leftarrow ih$
end
a^h,b^h and c^h of A_h and the source term f^h :
for $i \leftarrow 1$ **to** $n-1$ **do**
  $a_i^h \leftarrow \frac{-1}{h^2}$
  $c_i^h \leftarrow a_i^h$
end
for $i \leftarrow 1$ **to** n **do**
  $b_i^h \leftarrow \frac{2}{h^2} + c(x_i)$
  $f_i^h \leftarrow f(x_i)$
end
$(l^h, d^h) \leftarrow facto(a^h, b^h, c^h)$
$y^h \leftarrow desc(l^h, f^h)$
$U_h \leftarrow desc(c^h, d^h, y^h)$
Plot U_h with boundary conditions

Algorithm 4: algorithme de résolution du problème du fléchissement d'une poutre

2.2.1 Cas test et calcul de l'ordre de convergence

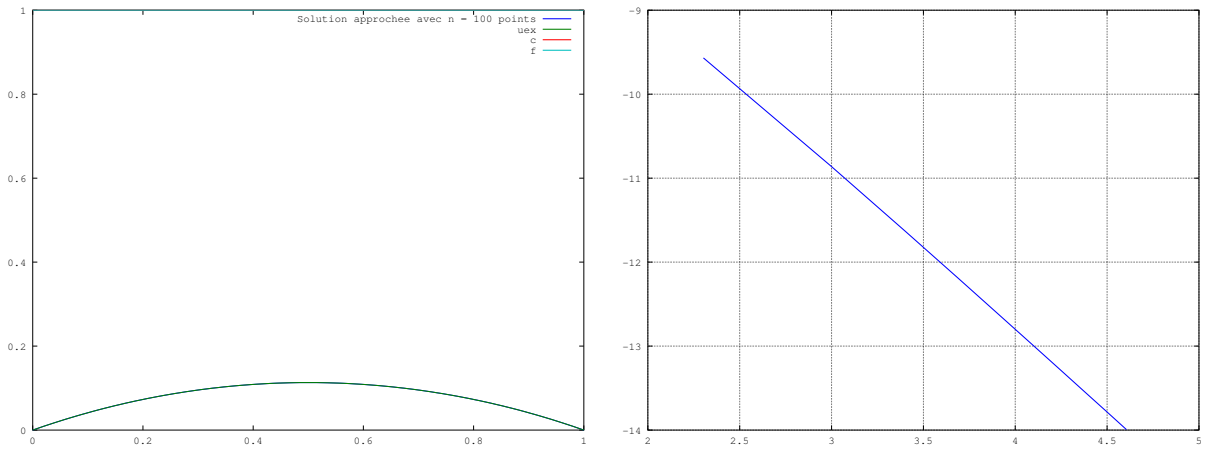
Avant de lancer une simulation numérique pour lequel on ne sait pas déterminer la solution analytique, nous allons considérer un cas simple où la fonction u est connue. En particulier, si on choisit $f(x) = c(x) = 1, \forall x \in [0,1]$ alors la solution de l'équation (1) est donnée par :

$$u(x) = -e^x \frac{-1+e^{-1}}{-e+e^{-1}} + e^{-x} \frac{e-1}{-e+e^{-1}} + 1.$$

Nous noterons cette solution u_{ex}. La solution approchée pour $n = 10$ et la solution exacte sont representés à la figure 1(a) : un calcul de la norme infinie de l'erreur avec la commande `norm(Vecteur,inf)` donne une erreur de $6.9930e-05$. Il est clair que notre méthode marche très bien.

En vue de calculer l'ordre de convergence (plus l'ordre est élevé et meilleure est la méthode), on calcule pour $n = 10$ à 100 par pas de 10 et on calcule $\|u-u_{ex}\|_\infty$ qu'on stocke dans un tableau ordre (voir Annexe B.4). On obtient alors un ordre de 1.9285 proche de l'ordre 2 théorique annoncé (voir debut de section). La figure 1(b) représente en abscisse le vecteur $N = [10\ 20\ 30\ \ldots\ 100]$ et en ordonnée la norme infinie de l'erreur en échelle logarithmique.

Ce cas test montre que notre algorithme 4 donne une approximation de la solution à l'ordre 2. On a donc *a priori* une bonne representation de la solution de l'équation (1). On va donc dans la suite abordé des cas où la solution analytique est inconnue (ou difficile à calculer).



(a) Solution approchée.

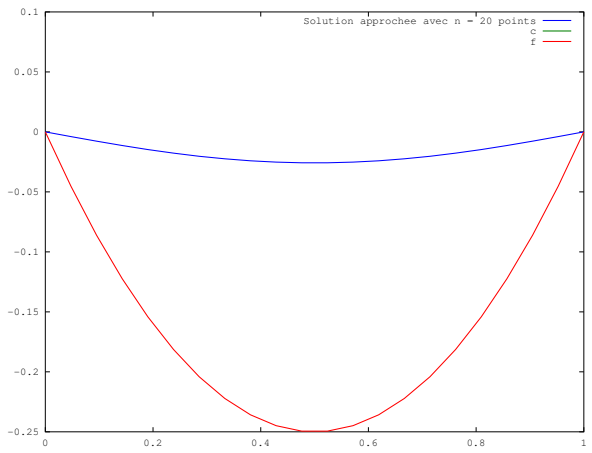
(b) Courbe d'erreur.

FIGURE 1 – Solution approchée de l'équation (1) avec $f(x) = c(x) = 1, \forall x \in [0, 1]$.

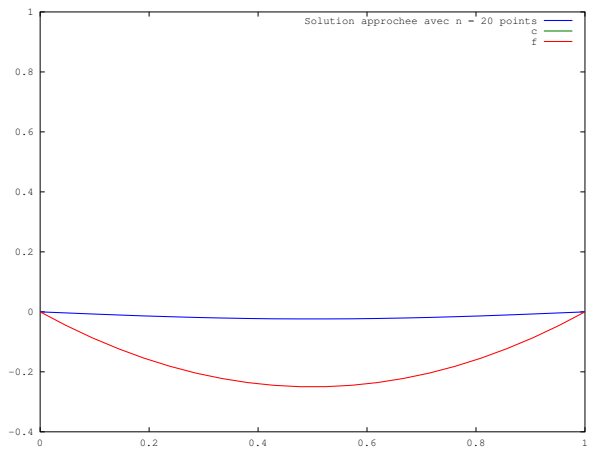
2.2.2 Applications

Dans toute la suite, on fixe n à 20.

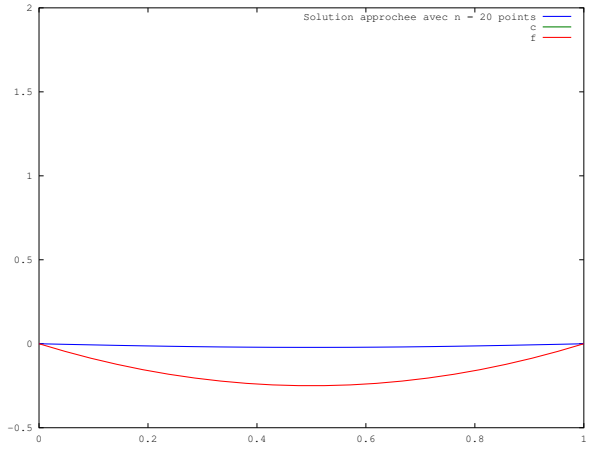
Sensibilité du modèle Dans ce premier test, nous allons considérer la fonction $f(x) = x(x-1)$ et $c(x) = \alpha \in \mathbb{R}$, $\forall x \in [0, 1]$. Les résultats de ce test sont donnés à la figure 2.



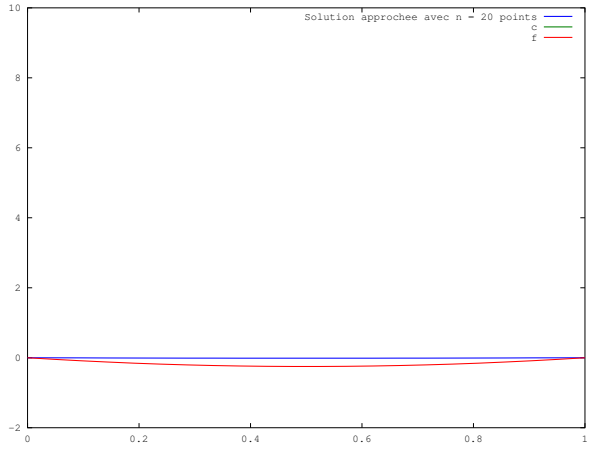
(a) $\alpha = 0.1$.



(b) $\alpha = 1$.



(c) $\alpha = 2$.



(d) $\alpha = 10$.

FIGURE 2 – Poutre homogène sous contrainte f.

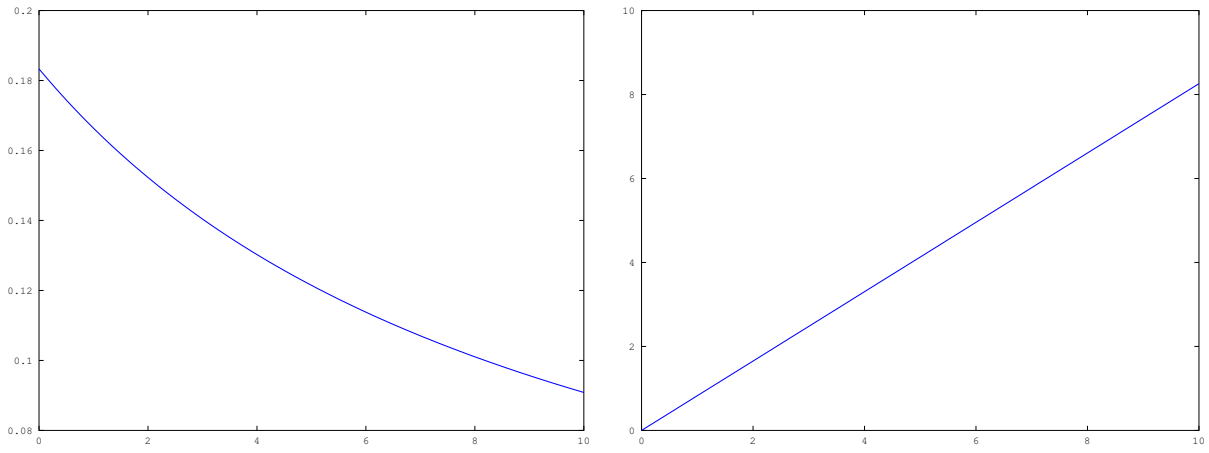
Ce premier test rend compte de la physique de l'équation (1). En effet, c est le coefficient de résititivité de la poutre et il est ici supposé constant. Par conséquent, plus ce coefficient est petit et plus la poutre se déforme. C'est exactement ce que nous observons à la figure 2. On calcule également la norme $\|u\|_1$ pour montrer que cette dernière est quasi nulle en présence d'un grand coefficient α :

α	0.1	1	2	10	100
$\|u\|_1$	0.34649	0.31778	0.29101	0.17389	0.0034534

On peut également faire une étude de la réponse de la poutre aux variations de α en présence d'une force fixe (ou l'inverse). Ce résultat est représenté à la figure 3(a) :

On étudie a présent la réponse du modèle par rapport à la force f. Nous supposerons ici c est une fonction constante égale à 1 et $f = \alpha$. Comme dans le paragraphe précédent, en

faisant varier la force, on peut tracer la réponse du modèle. Ce résultat est représenté à la figure 3(b).



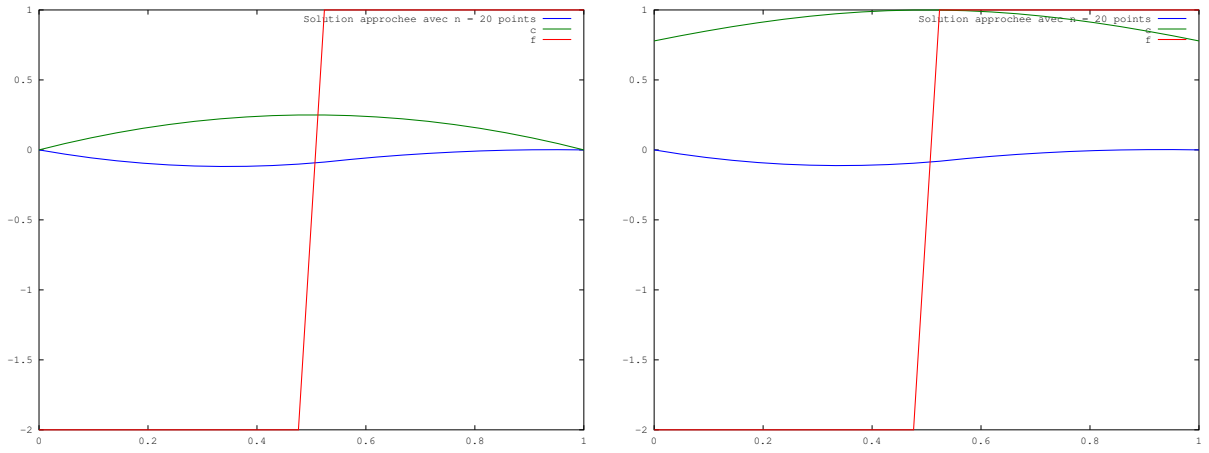
(a) test 1 (p/r à c). (b) test 2 (p/r à f).

FIGURE 3 – Réponse du modèle en fonction de α (abscisse : α, ordonnée $\|u\|_1$)

Cas discontinu On considère ici une fonction f discontinue

$$x \in [0,1],\ f(x) = \begin{cases} 1 & \text{si} \quad x > 0.5 \\ -2 & \text{sinon} \end{cases}$$

On étudie le déplacement de la poutre pour deux fonctions c variables. Nous obtenons les résultats suivants :



(a) $c(x) = x(1-x)$. (b) $c(x) = e^{(-(x-0.5)^2)}$.

FIGURE 4 – En présence d'une contrainte discontinue.

A Algorithme d'élimination de Gauss

B Code numérique

B.1 Fonction : factorisation LU

La fonction factorisation prend comme arguments les vecteurs a, b et c (voir remarque 1.1) et renvoie les vecteurs l et d.

Remarque B.1 *Toutefois, il faut remarquer que dans cette fonction $c(i-1)$ correspond à $c_i = u_i$ de l'algorithme 1 en raison de la numérotation qui commence par 2 dans l'algorithme et par 1 dans matlab (de par sa structure vectoriel).*

```
function [l,d] = facto(a,b,c)
n = length(b);
d(1) = b(1);
for i=2:n
  l(i-1) = a(i-1)/d(i-1);
  d(i)   = b(i)-l(i-1)*c(i-1);
end
end
```

B.2 Code numérique

La fonction `desc` prend comme arguments le vecteur l et le second membre f er renvoie la solution du système $Ly = f$. La fonction associée à l'algorithme B.2 est :

```
function [y] = desc(l,f)
n = length(f);
y(1) = f(1);
for i=2:n
  y(i)   = f(i)-l(i-1)*y(i-1);
end
end
```

B.3 Code numérique

La fonction `mont` prend comme argument y et c et d et renvoie la solution du système $Ux = y$, soit donc la solution du système $Ax = f$.

```
function [x] = mont(c,d,y)
n = length(y);
x(n) = y(n)/d(n);
for i=n-1:-1:1
```

```
  x(i)   = (y(i)-c(i)*x(i+1))/d(i);
end
end
```

B.4 Code numérique pour la résolution du fléchissement d'une poutre

Le code ci-dessous permet, en fonction du nombre n, de déterminer une solution approchée de l'équation (1).

```
function U =flechissement(n)
h = 1/(n+1); %pas de subdivision
x = [0:h:1]; %maillage

% Definition de Ah et de son second membre
ah = -ones(1,n-1)./h^2;
bh = 2*ones(1,n)/h^2+c(x(2:n+1));
fh = f(x(2:n+1));

%Algorithme de Thomas
[lh,dh] = facto(ah,bh,ah);
yh      = desc(lh,fh);
Uh      = mont(ah,dh,yh);


%Affichage de la solution
U = [0  Uh  0];
plot(x,U,x,c(x),x,c(x));
legend(['Solution approchee avec n = ',num2str(n),' points'],'c','f')
end
```

Le code suivant permet lui de déterminer l'ordre de convergence en fonction de l'erreur en norme infinie :

```
clear all
clc
close all

Err = [];

% Calcul de l'erreur en norme infinie
n=5:5:100;
for i=1:length(n)
 h = 1/(n(i)+1);
 x = [0:h:1];
```

```
 U   = flechissement(n(i));
 Uex = uex(x);
 Err=[Err,norm(U-Uex,inf)];
end

%Courbe d'erreur en échelle logarithmique
plot(log(n),log(Err))
grid on

%Calcul de l'ordre de convergence
ordre=polyfit(log(n),log(Err),1);
ordre=abs(ordre(1))
```


Acknowledgements

Ne pas hésiter à mettre des références bibliographiques, par exemple d'après [1] et [2] ou encore [1, 2]...

Références

[1] Prénom Nom. titre. In *Nom du livre*, volume numéro de volume of *Nom de serie*, page page. année.

[2] Prénom1 Nom1, Prénom2 Nom2, and Prénom3 Nom3. Titre. *Journal*, numéro de volume(numéro) :917–934, année.