Matlab et l'analyse numérique

Table des matières

I. Introduction

Matlab est un logiciel de calcul et de visualisation, dont les entitées de base sont des matrices. Matlab est une abréviation de Matrix Laboratory. Il est un langage interprété : il propose des facilités de programmation et de visualisation, ainsi qu'un grand nombre de fonctions réalisant diverses méthodes numériques. La meilleure façon d'apprendre à utiliser ce logiciel est de l'utiliser vous même, en faisant des essais, en commettant des erreurs et en essayant de comprendre les messages d'erreur qui vous seront renvoyés.

II. Analyse numérique

A. Méthodes directes de résolutions des systèmes linéaires

Considérons le système linéaire $Ax = b$ avec le cas simple A inversible diagonale :

$$\begin{pmatrix} a_{1,1} & 0 & 0 & ... & 0 \\ 0 & a_{2,2} & 0 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}$$

la solution sera $x_i = \frac{b_i}{a_{i,i}}$ avec $i \in [1, n]$. Le programme Matlab associé est le suivant :

```
function x=diago(A,b)
for i=1 :size(A,1)
x(i)=b(i)/A(i,i) ;
end
```

Si on suppose que A est triangulaire, le problème devient :

$$\begin{pmatrix} a_{1,1} & 0 & 0 & ... & 0 \\ a_{2,1} & a_{2,2} & 0 & ... & 0 \\ ... & ... & ... & ... & ... \\ a_{n,1} & ... & a_{n,n-2} & a_{n,n-1} & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}$$

et la solution sera

$$\begin{cases} x_1 = \frac{b_1}{a_{1,1}} \\ x_i = \frac{1}{a_{i,i}}(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j). \end{cases}$$

Le code matlab associé :

```
function x=triang(A,b)
x(1)=b(1)/A(1,1) ;
for i=2 :size(A,1)
som=b(i) ;
for j=1 :i-1
som=som-A(i,j)*x(j) ;
end
x(i)=som/A(i,i) ;
end
```

1) Méthode de Gauss: Il y a 4 principes fondamentales dans la résolution des systèmes linéaires, la solution x ne change pas lorsque on :
- permute 2 lignes
- permute 2 colonnes
- divise par un même terme non nul les éléments d'une ligne
- ajoute ou retranche à une ligne un certain nombre de fois une autre ligne

Donc on a intérêt à transformer le système linéaire en un système équivalent facile à résoudre : triangulaire. Soit le système à 4 inconnus suivant :

$$\begin{cases} 2x_1 + 4x_2 - 2x_3 = -6 \\ x_1 + 3x_2 + x_4 = 0 \\ 3x_1 - x_2 + x_3 + 2x_4 = 8 \\ -x_1 - x_2 + 2x_3 + x_4 = 6 \end{cases}$$

Le pivot dans ce cas est le coefficient de x_1 dans la première ligne ($pivot = 2$). Donc pour éliminer les coefficients de x_1 dans les autres lignes on effectue l'opération suivante pour chaque ligne :$L_i = L_i - \frac{a_{i,1}}{pivot} L_1$ et on obtient :

$$\begin{cases} 2x_1 + 4x_2 - 2x_3 = -6 \\ 0 + x_2 + x_3 + x_4 = 3 \\ 0 - 7x_2 + 4x_3 + 2x_4 = 17 \\ 0 + x_2 + x_3 + x_4 = 3 \end{cases}$$

La première variable a été éliminée de toutes les équations sauf une. On procède de la même façon pour les autres variables jusqu'à obtenir une matrice triangulaire. Le code Matlab associé à la triangularisation de Gauss est le suivant :

```
function [A,b]=descent(A,b)
for k=1 :size(A,1)-1
pivot=A(k,k) ;
if pivot =0
for i=k+1 :size(A,1)
b(i)=b(i)-A(i,k)/pivot*b(k)
for j=k+1 :size(A,1)
```

```
A(i,j)=A(i,j)-A(i,k)/pivot*A(k,j)
end
end
end
end
```

Après avoir obtenir une matrice triangulaire supérieure, on doit résoudre le système suivant :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & ... & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & ... & a_{2,n} \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}$$

Mathématiquement la solution est :

$$\begin{cases} x_n = \frac{b_n}{a_{n,n}} \\ x_i = \frac{1}{a_{i,i}}(b_i - \sum_{j=i-1}^{n} a_{i,j}x_j \end{cases}$$

et le code matlab associé :

```
function x=triang(A,b)
n=size(A,1);
x(n)=b(n)/A(n,n);
for i=n-1 :-1 :1
som=b(i);
for j=i+1 :n
som=som-A(i,j)*x(j);
end
x(i)=som/A(i,i);
end
```

et la fonction globale :

```
function x=gauss(A,b)
[U,c]=descent(A,b);
x=triang(U,c);
```

2) Méthode LU: On a trouvé que le système $Ax = b$ peut être transformé en $Ux = c$ donc on doit chercher L telle que $A = LU$ et $b = Lc$. A chaque étape de l'algorithme on a pour $i = k+1, ..., n$:

$$\begin{cases} a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}} a_{k,j}^{(k)} \text{ pour } j = k+1, ..., n \\ b_i^{(k+1)} = b_i^{(k)} - \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}}(b_k^{(k)} \end{cases}$$

Matriciellement : $A^{(k+1)} = M^{(k)}A^{(k)}$ et $b^{(k+1)} = M^{(k)}b^{(k)}$ avec

$$M^{(k)} = \begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & ... & m_{k+1,k} & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & m_{n,k} & 0 & 1 \end{pmatrix}$$

et $m_{i,k} = -\frac{a_{i,k}}{a_{k,k}}$. A la dernière itération, $A^{(n)} = U = M^{(n-1)}A^{(n-1)} = M^{(n-1)}M^{(n-2)}...M^{(1)}A = MA$ donc $A = M^{-1}U$ et on posant $L = M^{-1}$ on obtient $A = LU$.
Comment construire L et U ?
Idée : reprendre l'étape de triangularisation de la méthode de Gauss.

Les matrices élémentaires $M^{(k)}$ sont inversibles et leurs inverses sont les matrices $L^{(k)}$ triangulaires inférieures telles que :

$$L^{(k)} = \begin{cases} l_{i,j} = 0 \text{ si } j > i \\ l_{i,i} = 1 \text{ pour } i = 1, ..., n \\ l_{i,k} = \frac{a_{i,k}}{a_{k,k}} = -m_{i,k} \text{ pour } i = k+1, ..., n \end{cases}$$

Donc on obtient le code matlab de la décomposition LU suivant :

```
function [L,U]=decompose_lu(A)
n=size(A,1);
for k=1 :n-1
pivot=A(k,k);% startégie de pivot
si pivot =0
L(k,k)=1;
for i=k+1 :n
L(i,k)=A(i,k)/pivot;
for j=k+1 :n
A(i,j)=A(i,j)-L(i,k)A(k,j);
end
end
end
end
```

3) Décompostion de Cholesky: Le théorème de Cholesky : Si A est une matrice symétrique définie positive, il existe une unique matrice réelle triangulaire inférieure L telle que $A = LL^T$. On commence par calculer la première colonne de L : $l_{1,1} = \sqrt{a_{1,1}}$ et $a_{1,j} = l_{1,1}l_{j,1}$ d'où $l_{j,1} = \frac{a_{1,j}}{l_{1,1}}$ et de même façon on calcule le i^{eme} colonne après avoir calculer les $(i-1)$ premières colonnes :

$$\begin{cases} l_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2} \\ l_{j,i} = \frac{a_{i,j} - \sum_{k=1}^{i-1} l_{i,k}l_{j,k}}{l_{i,i}} \end{cases}$$

et le code matlab associé :

```
function L=cholesky(A)
n=size(A,1);
L(1,1)=sqrt(A(1,1))
L(2 :n,1)=(1/L(1,1))*A(2 :n,1);
for k=2 :n
L(k :n,k)=A(k :n,k);
for j=1 :k-1
L(k :n,k)=L(k :n,k)-L(k,j)*L(k :n,j);
end
L(k,k)=sqrt(L(k,k));
L(k+1 :n,k)=(1/L(k,k))*L(k+1 :n,k);
end
```

B. Calcul des valeurs propres

1) Méthode de puissance: La méthode de la puissance itérée est utilisée pour calculer la plus grande valeur propre et le vecteur propre associé d'une matrice symétrique définie positive A de taille $n \times n$. Ainsi A possède n valeurs propres : $\lambda_1, ..., \lambda_n$ telles que $|\lambda_1| > |\lambda_2| \geq ... \geq |\lambda_n|$. Soient $u_1, ..., u_n$ les vecteurs propres associés.
On pose x_0 un vecteur de IR^n de norme égale a 1 (ou quelconque) que l'on décompose de la façon suivante : $x_0 = \sum_{i=1}^{n} x_i u_i$ et on calcule la suite x_{k+1} comme suit :

$$\begin{array}{l} x_k = A^k x_0 \\ x_k = \sum_{i=1}^{n} \lambda_i^k a_i u_i \end{array}.$$

En mettant λ_1^k en facteur on obtient : $x_k = \lambda_1^k(\sum_{i=1}^{n}(\frac{\lambda_i^k}{\lambda_1^k})a_i u_i)$. Or $\frac{\lambda_i^k}{\lambda_1^k}$ pour $i > 2$ tendent vers 0 donc au bout de quelques itérations on obtient la plus grande valeur propore : $|\lambda_1| \approx \frac{||x_{k+1}||}{||x_k||}$ et le vecteur propre associé $u_1 \approx x_k$. Le code matlab associé est le suivant :

```
function [lambda,u]=puissance(A,x0,eps)
x=x0;
lambda=0;lambda_anc=1;
while abs(lambda-lambda_anc)>eps
lambda_anc=lambda;
u=x/abs(x);
x=A*u;
lambda=u'*x;
end
```

2) Méthode de puissance inverse: La plus petite valeur propre de A (en valeur absolue) est aussi la plus grande de A^1 (en valeur absolue) :

$$Au = \lambda u \text{ ssi } \frac{1}{\lambda}u = A^{-1}u \quad (1)$$

On peut donc appliquer la méthode de la puissance à A^{-1} mais la matrice A^{-1} doit être calculée. En général, on ne calcule pas l'inverse de la matrice A, mais on réalise sa décomposition LU. Donc si on suppose que $PA = LU$, le code Matlab sera :

```
function [lambda,u]=inverse(P,L,U,x0,eps)
x=x0;
lambda=0;lambda_anc=1;
while abs(lambda-lambda_anc)>eps
lambda_anc=lambda;
u=x/abs(x);
c=P*u;
x=triang(U,c);%utiliser la fonction triang
lambda=1/(x'*u);
end
```

C. Discrétisation de l'EDP

Soit $\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = 0$ avec $(x, y) \in [a, b] \times [c, d]$. La discrétisation de l'intervalle $[a, b]$ est effectué en posant $h_x = h = \frac{b-a}{n_x}$ et on obtient $x(i) = x_i = a + ih$ et $h_y = h = \frac{d-c}{n_y}$ et on obtient $y(j) = y_j = c + jh$ pour $i = 0, 1, ..., n_x$ et $j = 0, 1, ..., n_y$.
La méthode de différences finies consiste à approximer les dérivées partielles d'une équation au moyen des développements de Taylor et ceci se déduit directement de la définition de la dérivée. Posons $u(x_i, y_j) = u_{i,j}$, on a :

$$\begin{cases} \frac{\delta^2 u}{\delta x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} \\ \frac{\delta^2 u}{\delta y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} \end{cases}$$
$$\Delta u = \frac{u_{i+1,j} - 4u_{i,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{h^2}$$

Soit à résoudre l'équation de Laplace sur l'intervalle $[a, b] \times [c, d]$ avec les conditions aux limites :

$$\begin{cases} \Delta u = 0 \\ u(x, c) = u(x, d) = u(a, y) = 0 \\ u(b, y) = 100 \end{cases}$$

Si on discrétise l'intervalle $[a, b] \times [c, d]$ avec un pas h on obtient un système linéaire $Ax = b$ avec A de taille $n_x n_y \times n_x n_y$ et b de taille $n_x n_y$:

$$A = \begin{pmatrix} -4 & 1 & & & 1 & & & & & \\ 1 & -4 & 1 & & & 1 & & & & \\ 0 & 1 & -4 & 1 & & & 1 & & & \\ & \ddots & & \ddots & & & & \ddots & & \\ & & & 1 & -4 & 0 & & & 1 & \\ 1 & & & & 0 & -4 & 1 & & & 1 \\ 0 & \ddots & & \ddots & & 1 & \ddots & \ddots & \ddots & \\ \vdots & & & & & & & & & \\ 0 & & & & 1 & & & & 1 & -4 \end{pmatrix} \text{ et } B = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ -100 \\ \vdots \\ \vdots \\ -100 \\ \vdots \\ \vdots \\ -100 \end{pmatrix}$$

Le programme Matlab pour construire A et b est le suivant :

```
function [A,b]=edp(a,b,c,d,h)
nx=(b-a)/h;
ny=(d-c)/h;
n=nx*ny;
% (remplissage des éléments de la matrice
A)
A=zeros(n);
for i=1 :n-1
A(i,i)=-4;
A(i+1,i)=1;
A(i,i+1)=1;
if (mod(i,(nx-1))==0)
A(i+1,i)=0;
A(i,i+1)=0;
end
end
for i=1 :n-nx+1
A(nx-1+i,i)=1;
A(i,nx-1+i)=1;
end
A(n,n)=-4;
% (remplissage des éléments de la matrice
B)
for i=1 :n
B(i)=0;
if (mod(i,nx-1)==0)
B(i)=-100;
end
end
```

III. Recherches Opérationnelles

A. Méthode de simplexe

```
function[A, x, z] = simplex(c, A, b);
c = -c;
[m, n] = size(A);
A = [A eye(m)];
b = b( :);
```

```
c = c( :)';
A = [A b];
d = [c zeros(1,m+1)];
A = [A;d];
mi=min(A(m+1,1 :m+n));
col=find(A(m+1, :)==mi);
subs = n+1 :m+n;
while mi < 0 & abs(mi) > eps
t = A(1 :m,col);
if all(t <= 0)x = zeros(n,1);
z = inf;
return;
end
t1=A(1 :m,m+n+1)
t2=A(1 :m,col);
l=find(t2 > 0);
[mi, row] = min(t1(l)./t2(l));
row = l(row);
if  isempty(row)
A(row, :)= A(row, :)/A(row,col);
subs(row) = col;
for i = 1 :m+1
if i  = row
A(i, :)= A(i, :)-A(i,col)*A(row, :);
end
end
end
mi=min(A(m+1,1 :m+n));
col=find(A(m+1, :)==mi);
end
z = A(m+1,m+n+1);
x = zeros(1,m+n);
x = x(1 :n)';
```

B. Théorie des graphes

1) Algorithme de Kruskal:

```
%w :poids de l'arbre et T :matrice
d'adjacence de l'arbre function [w,T] =
krus(G)
ligne = size(G);
% cration de la matrice d'adjacence
X = [];
for i = 1 : ligne
X(G(i,1),G(i,2)) = 1;
X(G(i,2),G(i,1)) = 1;
end
n = size(X,1);
for i = 1 : ligne - 1
d = ligne + 1 - i;
for j = 1 : d - 1
if G(j,3) > G(j + 1,3)
G([j j + 1], :) = G([j + 1 j], :);
end
end
end
aretes = zeros(1,n);
T = zeros(n);
% tester l'existence d'un cycle si on
insert l'arete 'new'
for i = 1 : ligne
new = G(i,[1 2]);
g=max(aretes)+1;
test=0;
n=length(aretes);
if aretes(new(1))==0 && aretes(new(2))==0
aretes(new(1))=g;
aretes(new(2))=g;
elseif aretes(new(1))==0
aretes(new(1))=aretes(new(2));
elseif aretes(new(2))==0
aretes(new(2))=aretes(new(1));
elseif aretes(new(1))==aretes(new(2))
test=1;
else
m=max(aretes(new(1)),aretes(new(2)));
for i=1 :n
if aretes(i)==m
aretes(i)=min(aretes(new(1)),aretes(new(2)));
end
end
end
if test == 1
G(i, :) = [0 0 0];
end
end
w = sum(G( :,3)'); for i = 1 : ligne
if G(i,[1 2])  = [0 0]
T(G(i,1),G(i,2)) = 1;
T(G(i,2),G(i,1)) = 1;
end
end
```

2) Algorithme de Ford:

```
%poids :matrice des poids
%source :sommet de départ
%dest :sommet d'arrivée
function [chemin,cost]=ford(poids, source,
dest)
clc;
n = size(poids, 1);
distance(1 :n) = inf;
somprec(1 :n) = inf;% sommet précédent
distance(source) = 0;
for i = 1 :n-1
for j = 1 :n
for k = 1 :n
if ((distance(j) + poids(j,k) <
distance(k)) && (poids(j,k)  = 0))
distance(k) = distance(j) + poids(j,k);
somprec(k) = j;
end
end
end
end
chemin = [dest];
traverse = dest;
cost=0;
while (somprec(traverse)  = source)
chemin = [somprec(traverse) chemin];
traverse = somprec(traverse);
cost=cost+poids(chemin(1),chemin(2));
end
chemin = [somprec(traverse) chemin];
cost=cost+poids(chemin(1),chemin(2));
```