Quelques méthodes numériques en mathématique programmées sous SCILAB.

K.Barty

29 avril 2004

Table des matières

1 Résolution de systèmes linéaires **1**
1.1 Élimination de Gauss 2
1.2 Factorisation LU 3
1.3 Factorisation de Cholesky 5
1.4 Méthode de Jacobi 6

2 Résolution numérique d'équation **7**
2.1 Méthode de la Dichotomie 7
2.2 Méthode de la Section Dorée 9

3 Interpolation de Lagrange **11**

4 Programmation Dynamique **12**
4.1 Un problème de gestion de stock 12

1 Résolution de systèmes linéaires

Dans cette partie nous sommes concernés par le problème suivant : Résoudre l'équation

$$Ax = b \; ; \tag{1}$$

sans pour autant inverser la matrice A ce qui comme nous le savons est numériquement très coûteux.

1.1 Élimination de Gauss

```
function [A,b]=algo(A,b,k)

  n=size(A,'c')
  for i=k+1:n
    b(i)=b(i)-(A(i,k)/A(k,k))*b(k)
    for j=k+1:n
      A(i,j)=A(i,j)-(A(i,k)/A(k,k))*A(k,j)
    end;
    for j=1:k
      A(i,j)=0
    end
  end;

endfunction

function [A,b]=gauss(A,b)
// Methode  de Gauss
// On suppose que tous les pivots A(k,k) sont differents de 0.
// En entree il y a les coefficients de la matrice A et du vecteur b qui
// déterminent le système lineaire Ax=b
//  En  sortie le couple (A,b) determine un sytème lineaire equivalent,
//  mais numeriquement plus simple a resoudre, la matrice A etant
//  triangulaire superieure.
  n=size(A,'c')
  for k=1:n
    [A,b]=algo(A,b,k)
  end;

endfunction;
```

```
A=[3 -4 0;12 -9 -1;24 -4 8]
```
← Définition de la

A =

```
!  3.   - 4.    0. !
!  12.  - 9.  - 1. !
!  24.  - 4.    8. !
```

//matrice `A`

```
b=[12;-6;23]
```
← Second membre du système

```
 b  =

!   12.  !
```

```
! - 6.  !
!   23. !
```

`getf gauss.sci` ← On charge la fonction gauss

//dans scilab

```
[A1,b1]=gauss(A,b)
 b1  =

!   12.  !
! - 54.  !
!   143. !
 A1  =

!   3.  - 4.    0.  !
!   0.    7.  - 1.  !
!   0.    0.    12. !
```

← Appel de la fonction gauss

Le système :

$$A_1 x = b_1 \ ;$$

est plus facile à résoudre que (1) car la matrice A_1 est triangulaire.

1.2 Factorisation LU

On factorise A en un produit d'une matrice triangulaire inférieure L et d'une matrice triangulaire supérieure U, autrement dit $A = LU$. Cela permet une résolution numérique plus facile en deux étapes. La première étape consiste à résoudre le problème :

$$Lx' = b \ ;$$

la seconde étape à résoudre le problème :

$$Ux = x'.$$

Le vecteur x est alors solution de l'équation (1).

```
function [L,U]=facLU(A)
// Factorisation LU d'une matrice inversible A
//
```

```
U=zeros(A);
L=eye(A);
//
U(1,:)=A(1,:);
//
dimmat=size(A,'r');
//
for i=2:dimmat
    for j=1:i-1
        L(i,j)=(A(i,j)-L(i,1:j-1)*U(1:j-1,j))/U(j,j);
    end
    for j=i:dimmat
        U(i,j)=A(i,j)-L(i,1:i-1)*U(1:i-1,j);
    end
end
endfunction
```

```
A=[3 -4 0;12 -9 -1;24 -4 8]
```
← Définition de la

```
 A  =

!  3.   - 4.    0. !
!  12.  - 9.  - 1. !
!  24.  - 4.    8. !
```

//matrice `A`

```
getf facLU.sci
```
← On charge la fonction facLU

//dans scilab

```
[L,U]=facLU(A)
```
← Appel de la fonction facLU

```
 U  =

!   3.  - 4.    0.  !
!   0.    7.  - 1.  !
!   0.    0.    12. !
 L  =

!   1.    0.    0. !
!   4.    1.    0. !
!   8.    4.    1. !
```

1.3 Factorisation de Cholesky

On détermine la factorisation de Cholesky d'une matrice symétrique A. Autrement dit on calcule la matrice C telle que $A = CC'$.

```
function [C]=facCho(A)
// Factorisation de Cholesky de la matrice A
//
C=zeros(A);
C(1,1)=sqrt(A(1,1));
dimmat=size(A,'r');
for i=2:dimmat
    for j=1:i-1
        C(i,j)=(A(i,j)-(C(i,1:j-1)*C(j,1:j-1)'))/C(j,j);
    end
    C(i,i)=sqrt(A(i,i)-(C(i,1:i-1)*C(i,1:i-1)'));
end

endfunction
```

```
A=[3 -4 0;12 -9 -1;24 -4 8]
```
← Définition de la

A =

```
!  3.   - 4.    0. !
!  12.  - 9.  - 1. !
!  24.  - 4.    8. !
```

//matrice `A`

```
b=[12;-6;23]
```
← Second membre du système

```
 b  =

!   12. !
! - 6.  !
!   23. !

getf gauss.sci
```
← On charge la fonction gauss

//dans scilab

```
[A1,b1]=gauss(A,b)
```
← Appel de la fonction gauss

```
 b1  =
```

```
!   12.  !
! - 54.  !
!   143. !
 A1  =

!   3.  - 4.    0.  !
!   0.    7.  - 1.  !
!   0.    0.    12. !
```

1.4 Méthode de Jacobi

La méthode de Jacobi est une technique itérative pour résoudre un système linéaire.

```
function [x]=Jacobi(A,b,x0,k)
//
// On veut resoudre le systeme Ax=b
// de maniere recursive
// x0 condition initiale de l'algorithme de Jacobi
// k nombre d'itération
//
  n=size(A,'c')
  x=x0;
  for p=1:k
    for i=1:n
      x(i)=(b(i)-A(i,1:n)*x+A(i,i)*x(i))/A(i,i)
    end
  end

endfunction
```

```
A=[264 110 98;110 105 21;98 21 54]
 A  =

!   264.    110.    98. !
!   110.    105.    21. !
!   98.     21.     54. !
```

```
b=[12;8;-5]
 b  =

!   12. !
!   8.  !
! - 5.  !

x0=[0;0;0]
 x0  =

!   0. !
!   0. !
!   0. !

getf Jacobi.sci

[x1]=Jacobi(A,b,x0,1000)                                    ← Appel de
 x1  =

!   0.4176994 !
! - 0.2074027 !
! - 0.7699830 !

                              // la fonction

                              //Jacobi

norm(A*x1-b)
 ans  =

    3.202D-15
```

2 Résolution numérique d'équation

2.1 Méthode de la Dichotomie

Nous allons présenter la méthode de Dichotomie pour minimiser une fonction unimodale.

```
function [intervalle,iter]=dicho(fonction,intervalle,epsilon,iter)
// Methode de dichotomie utilisant la dédivée
```

```
// En entree
// fonction   : fonction supposee unimodale de la forme
//              y=fonction(x).
// intervalle : vecteur 1x2, intervalle de depart pour l'optimisation.
// epsilon    : reel positif, precision de l'optimisation.
// iter       : entier, nombre d'iteration pour l'algotithme.
// En sortie
// intervalle : vecteur 1x2, resultat de la dichotomie.
// iter       : nombre d'iterations restantes a effectuer.
//
// Test de  l'intervalle de départ
//
init=iter;
binf=min(intervalle);
bsup=max(intervalle);
if norm(binf-bsup)==%inf then error('une des bornes de l''intervalle est infinie'), end;
  bdemi   = (binf+bsup)/2;
  f_binf  = fonction(binf);
  f_bsup  = fonction(bsup);
  f_bdemi = fonction(bdemi);
  u=file('open','results_dicho','unknown');

  // Methode de dichotomie
  while ((abs(bsup-binf)>= epsilon) & (iter >= 1))
    b1quart   = (bdemi+binf)/2;
    b3quart   = (bsup+bdemi)/2;
    f_b1quart = fonction(b1quart);
    f_b3quart = fonction(b3quart);

    tab=[f_binf,f_b1quart,f_bdemi,f_b3quart,f_bsup];
    [a,b]=gsort(tab,'c','i');

    if (b(1) == 1) | (b(1) == 2) then
      bsup    = bdemi;
      bdemi   = b1quart;
      f_bsup  = f_bdemi;
      f_bdemi = f_b1quart;
    elseif b(1) == 3  then

      binf    = b1quart;
      bsup    = b3quart;
      f_binf  = f_b1quart;
      f_bsup  = f_b3quart;
    elseif (b(1) == 4) | (b(1) == 5) then
      binf     = bdemi;
```

```
      bdemi    = b3quart;
      f_binf   = f_bdemi;
      f_bdemi  = f_b3quart;
    end
    fprintf(u,'iter = %2.0f binf = %10.8f bsup = %10.8f precision %10.8f',...
init-iter+1,binf,bsup,abs(bsup-binf));
    iter=iter-1;
  end
  intervalle=[binf,bsup];
  file('close',u)
endfunction
```

```
deff("[y]=func(x)","y=x^2-4*x+6")
```
← Définition de la //fonction `func`

```
getf dicho.sci
```
← On charge la fonction `dicho` //dans scilab

```
[intervalle,iter]=dicho(func,[-20,10],0.001,10)
 iter  =

    0.
 intervalle  =

!   1.9873047    2.0166016 !
```

2.2 Méthode de la Section Dorée

C'est une variante de la méthode de Dichotomie, elle permet entre autre de n'avoir à évaluer qu'une seule fois la fonction à minimiser à chaque itération.

```
function [intervalle,iter]=section_doree(fonction,intervalle,epsilon,iter)
//
init=iter;
binf=min(intervalle);
bsup=max(intervalle);
tau=(1+sqrt(5))/2;
tau=1/tau;
if norm(binf-bsup)==%inf then error('une des bornes de l''intervalle est infinie'), end;
  b1   = bsup-tau*(bsup-binf);
```

```
  b2   = binf+tau*(bsup-binf);
  f_binf  = fonction(binf);
  f_bsup  = fonction(bsup);
  f_b1    = fonction(b1);
  f_b2    = fonction(b2);
  u=file('open','results_doree','unknown');
  // Methode de la section doree
  while ((abs(binf-bsup)>= epsilon) & (iter >= 1))
    tab=[f_b1,f_b2];
    [a,b]=gsort(tab,'c','i');
    if b(1)==1 then
      bsup=b2;
      b2=b1;
      f_b2=f_b1;
      b1=binf+bsup-b2;
      f_b1=fonction(b1);
    else
      binf=b1;
      b1=b2;
      f_b1=f_b2;
      b2=binf+bsup-b1;
      f_b2=fonction(b2);
    end
    fprintf(u,'iter = %2.0f binf = %10.8f bsup = %10.8f precision %10.8f',...
init-iter+1,binf,bsup,abs(bsup-binf));
    iter=iter-1;
  end
  file('close',u)
  intervalle=[binf,bsup];
endfunction
```

`deff("[y]=func(x)","y=x^2-4*x+6")` ← Définition de la

//fonction `func`

`getf section_doree.sci` ← On charge la fonction `dicho`

//dans scilab

```
[intervalle,iter]=section_doree(func,[-20,10],0.001,10)
 iter  =
```

```
    0.
 intervalle  =

!   1.8847051    2.1286236 !
```

3 Interpolation de Lagrange

Étant donné une fonction $f : \mathbb{R} \to \mathbb{R}$ et $x_0, \ldots, x_n$ $n+1$ points dans $\mathbb{R}$. Lagrange à résolu le problème de déterminer le polynome P vérifiant :

$$P(x_i) = f(x_i), \quad i = 0, \ldots, n.$$

On montre que ce polynome P s'écrit sous la forme suivante :

$$P(x) = f[x_0] + \sum_{j=1}^{n} f[x_0, \ldots, x_j] \Pi_{i=0}^{j-1}(x - x_i) \ ;$$

avec :

$$f[x_i] = f(x_i), \quad f[x_i, \ldots, x_{i+k+1}] = \frac{f[x_{i+1}, \ldots, x_{i+k+1}] - f[x_i, \ldots, x_{i+k}]}{x_{i+k+1} - x_i}. \tag{2}$$

Nous allons ranger les éléments $f[x_i, \ldots, x_{i+k+1}]$ dans une matrice A de la manière suivante :

- l'indice de ligne sera l'indice du dernier terme dans le crochet $[x_i, \ldots, x_{i+k+1}]$ auquel on rajoute 1 c'est-à-dire $i + k + 2$;
- l'indice de colonne correspond au nombre d'éléments entre le crochet $[x_i, \ldots, x_{i+k+1}]$ c'est-à-dire $k + 2$.

Autrement dit $A(i + k + 2, k + 2) = f[x_i, \ldots, x_{i+k+1}]$. En notant $points = [x_0, \ldots, x_n]$ ($points(i) = x_{i-1}$), l'équation (2) devient :

$$A(i + k + 2, k + 2) = \frac{A(i + k + 2, k + 1) - A(i + k + 1, k + 1)}{points(i + k + 2) - points(i + 1)}.$$

On procède alors à un changement de variable en posant $I = i + k + 2$ et $J = k + 2$ il vient donc :

$$A(I, J) = \frac{A(I, J-1) - A(I-1, J-1)}{points(I) - points(I - J + 1)}$$

Pour des raisons d'efficacité nous avons programmé deux fonctions. La première calcule la matrice A, la seconde calcule le polynome P en utilisant A.

```
function [y]=interpolation(x,d,points)
// Polynome d'interpolation de Lagrange d'une fonction f
   n=size(points,'*')
```

```
  y=d(1)
  T=x-points(1);
  for i=2:n
    y=y+d(i)*T;
    T=T*(x-points(i));
  end

endfunction



function [d]=preposs(f,points)
  n=size(points,'*')
  A=zeros(n,n);


 for k=1:n
  A(k,1)=f(points(k));
 end
  d(1)=A(1,1);
  for j=2:n
      for i=j:n
       A(i,j)=(A(i,j-1)-A(i-1,j-1))/(points(i)-points(i-j+1));
      end
      d(j)=A(j,j);
  end
endfunction
```

L'exemple suivant montre le résultat en bleue de l'interpolation de la fonction $x \mapsto sin(x)$ en rouge aux points [0 0.5 1 1.5 2]

4 Programmation Dynamique

4.1 Un problème de gestion de stock

Exercice 4.1. Un homme d'affaire dispose d'un hangar de 2 places pour ranger des containers. Dans chaque place il ne peut ranger qu'un seul container. Cet homme peut scrupuleux c'est arrangé pour connaître le prix sur le marché de ces containers pour les 5 jours à venir. Il sait alors que $p_1 = 5,\ p_2 = 10,\ p_3 = 18,\ p_4 = 11,\ p_5 = 13$. Chaque jour il ne peut faire que les mouvements suivants sur son stock :

- il ne fait rien $u = 0$;
- il achète un container $u = 1$;
- il vend un container $u = -1$.

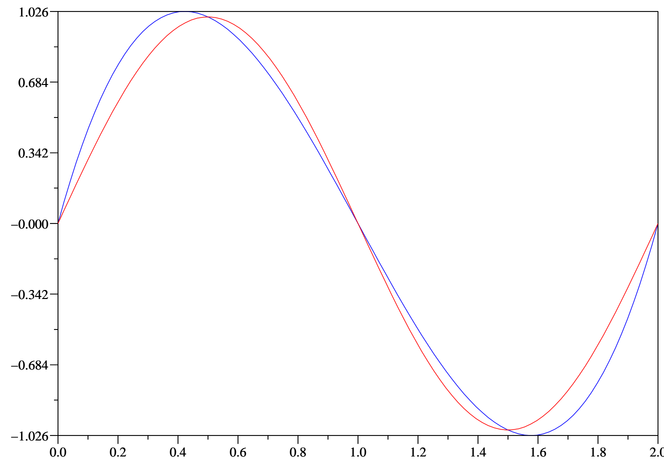

FIG. 1 – Interpolation de Lagrange de la fonction *sinus*

On note x_t ($t = 1, \ldots, 5$) l'état du stock à l'instant t, c'est-à-dire le nombre ($0, 1$ ou 2) de containers dans le hangar, et u_t la décision de l'homme d'affaire. Interpréter les formules suivantes :

$$x_{t+1} = x_t + u_t$$

et :

$$V_t(x) = \max_{u=0,1,-1} \{-p_t u + V_{t+1}(x+u)\}, \quad V_5 = 13x.$$

1. Quel profit peut-il espérer réaliser en partant d'un stock vide ?, en suivant qu'elle stratégie (c'est-à-dire qu'elle suite u_1, u_2, u_3, u_4, u_5 de décisions devra t'il prendre) ?
2. Discuter de la mise en œuvre informatique d'un programme permettant de calculer une stratégie optimale en fonction des prix du marché.

Le programme permettant de mettre en œuvre la dynamique rétrograde est le suivant :

```
function [V,control]=progdyn(prix)
  // Programmation dynamique pour un problème de gestion de stock
  time=size(prix,'*')
  etat=3 //(0,1,2)
  V=zeros(etat,time);
  control=V;
  for i=1:etat
    V(i,time)=prix(time)*(i-1); //cout a l'etat final
  end
  for t=time-1:-1:1
    for i=1:etat
```

```
      arg=[];
      decision=[];
      for u=-1:1
        if ((i-1+u)==0 | (i-1+u)==1 | (i-1+u)==2) then
          decision=[decision u]
          arg=[arg -prix(t)*u+V(i+u,t+1)]
        end
      end
      [val1,val2]=gsort(arg,'c','d')
      V(i,t)=val1(1);
      control(i,t)=decision(val2(1));
    end
  end
endfunction
```

```
prix=[5 10 18 11 13]                                         ← Prix des containers
 prix  =

!   5.    10.    18.    11.    13. !

getf progdyn.sci

[V,control]=progdyn(prix)                                    ← Appel de la fonction
 control  =

!   1.    1.    0.    1.    0. !
!   1.    1.  - 1.    1.    0. !
!   0.    0.  - 1.    0.    0. !
 V  =

!   18.    10.    2.     2.     0.  !
!   28.    23.    20.    15.    13. !
!   33.    33.    33.    26.    26. !
```