RENDU_TP/TD
Travaux Pratiques de calcul numérique 3

Réalisé par:
Mr Lougani Faouzi

Résumé :
Ce rapport contient des explications,solutions d'exercices se basant sur Les opérations de base de l'algèbre linéaire dense et leur implémentation .Leur formats de stockage (exemple BLAS)
Et surtout, les différentes manières d'écrire les algorithmes et d'organiser les boucles. Ensuite l'implémentation d'une méthode de résolution directe pour les systèmes linéaires est étudiée . Enfin, durant cette partie on a pu avoir des idées sur l'optimisation de l'implémentation de ce type d'algorithme en prenant en compte les propriétés des opérateurs tant mathématiques que structurelles.

Les fichiers/rapport/code c et scilab sont disponibles sur :

https://github.com/lougani-faouzi/calcul_numerique/tree/master/TP_BLAS_LAPACK_CODE_RAPPORT/

EXERCICE 01:

1. Approximer la dérivée seconde de T au moyen d'un schéma centré d'ordre 2.

on a:

$$u(x_i + h) = u(x_i) + h\left(\frac{du}{dx}\right)_i + \frac{h^2}{2}\left(\frac{d^2u}{dx^2}\right) + o(h^2)$$

$$u(x_i - h) = u(x_i) - h\left(\frac{du}{dx}\right) + \frac{h^2}{2}\left(\frac{d^2u}{dx^2}\right) + o(h^2)$$

on somme les equations:

$$\frac{-u(x_i - h) + 2u(xc_i) - u(x_i + h)}{h^2} = g_i + o(h^2)$$

pour tout $i \in [1, n]$:

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = g_i$$

Donc: Le schema est:

$$-u_{i+1} + 2u_i - u_{i+1} = h^2 g_i$$

2. Ecriture du systeme linaire correspondant au problème (1):

Les conditions de bord: $u_0 = T_0 \quad i = 0$

$$-u_0 + 2u_1 - u_2 = h^2 g_1 \quad \text{pour } i = 1$$

$$-u_1 + 2u_2 - u_3 = h^2 g_2 \quad \text{pour } i = 2$$

$$-u_{k-1} + 2u_k - u_{k+1} = h^2 g_k \quad \text{pour } i = k$$

$$-u_{n-1} + 2u_n - u_{n+1} = h^2 g_n \quad \text{pour } i = N$$

$$u_{n+1} = T_1 \quad \text{pour } i = N+1$$

Avec les conditions de Bord :

pour $i = 1$

$$2u_1 - u_2 = h^2 g_1 + T_0$$

pour $i = n$

$$-u_{n-1} + 2u_n = h^2 g_n + T_1$$

En explicitant le système linéaire $Au = g$

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 \cdots & 0 \\ 0 & & & & \\ \vdots & & & & 0 \\ & 0 & & 2 & -1 \\ 0 & & 0 & -1 & 2 \end{pmatrix} \qquad u = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}$$

$$g = \begin{pmatrix} h^2 g_1 + T_0 \\ h^2 g_2 \\ \vdots \\ h^2 g_{n-1} \\ h^2 g_n + T_1 \end{pmatrix}$$

Comme on a pas de source de chaleur :

$\forall i \in [1, n]$ on a :

$$h g_i = 0$$

D'où $g = \begin{pmatrix} T_0 \\ 0 \\ \vdots \\ 0 \\ T_1 \end{pmatrix}$

Exercice 02 :
Le but de cet exercice est d'avoir un environnement de travail compatible afin de réaliser le tp .
J'ai installer **liblapack** et **libblas** mais lorsque le compile le makefile une erreur
./include/blaslapack_headers.h:2:10: fatal error: lapacke.h: Aucun fichier ou dossier de ce type
```
    2 | #include <lapacke.h>
      |          ^~~~~~~~~~~
compilation terminated.
```

Pour régler ce problème on exécute les commandes suivantes :

```
dpkg -L liblapack-dev
sudo apt-get install liblapacke-dev
```

Une compilation avec la commande **make all** exécute le code qui résout l'équation de la chaleur ainsi que les autres .

Exercice 03 :

1/En C, comment doit on déclarer et allouer une matrice pour utiliser BLAS et LAPACK :
On doit déclarer la matrice comme un pointeur en C .
et l'allouer de manière dynamique (c'est a dire usage de malloc)
comme exemple la matrice AB du code ***tp2poisson1Ddirect.c*** :

```
double *AB; //la matrice de contenant des éléments en double précision
AB = (double *) malloc(sizeof(double)*lab*la); // l'allocation de matrice de
//dimension  lab*la
```

2/La signification de la constante **LAPACK_COL_MAJOR** :
Cette constante spécifie que les tableaux bidimensionnels sont de colonne principale .

3/A quoi correspond la dimension principale (leading dimension)généralement notée **ld** :
En général, la dimension principale (leading dimension) est égale au nombre d'éléments dans la dimension principale.
Il est également égal à la distance en éléments entre deux éléments voisins dans une ligne de dimension mineure.

4/Que fait la fonction **dgbsv** ?
DGBSV calcule la solution d'un système linéaire A * X = B, où A est une matrice de de taille N avec des sous-diagonales KL et les superdiagonales KU, et X et B sont des matrices N-by-NRHS.

Quelle méthode implémente-t-elle ?
Elle implémente la méthode :**LAPACKE_dgbsv**

Donc on a pour le code étudié ***tp2poisson1Ddirect.c***:
-Pour un stockage en priorité ligne :
LAPACKE_dgbsv(LAPACK_ROW_MAJOR,la, kl, ku, NRHS, AB, la, ipiv, RHS, NRHS);
-Pour un stockage en priorité colonne :
LAPACKE_dgbsv(LAPACK_COL_MAJOR,la, kl, ku, NRHS, AB, lab, ipiv, RHS, la);

5/modifier ***tp2poisson1Ddirect.***c et ***libpoisson1D.c*** afin d'effectuer les opérations pour un stockage en priorité ligne et non colonne :

Il faut remplir la fonction **set_GB_operator_rowMajor_poisson1D()** du fichier l***ibpoisson1D.c*** **,**pour se faire il suffit de faire la transposé de ce qu'on fait en **set_GB_operator_colMajor_poisson1D()** déjà donnée .

On aura :

```
void set_GB_operator_rowMajor_poisson1D(double* AB, int *lab,int
*la){

  //TODO
  int ii_1,ii_2,ii_3,ii_4,ii_5;
  int jj;

  for(jj=0;jj<(*la);jj++){
  ii_1=jj;
  ii_2=(*la)+jj;
  ii_3=2*(*la)+jj;
  ii_4=3*(*la)+jj;
  AB[ii_1]=0.0;
  AB[ii_2]=-1.0;
  AB[ii_3]=2.0;
  AB[ii_4]=-1.0;

  }
  AB[*la]=0.0;
  AB[4*(*la)-1]=0.0;
}
```

Pour faire l'appel il faut aller au fichier ***tp2_poisson1D_direct.c*** et modifier la variable du main **row en lui affectant 1** c'est a dire :

```
  info=0;

  /* working array for pivot used by LU Factorization */
  ipiv = (int *) calloc(la, sizeof(int));

  int row = 1; // c'est ici qu'on doit modifier row=1 pour
  // que set_GB_operator_rowMajor_poisson1D soit appelée

  if (row == 1){ // LAPACK_ROW_MAJOR
    set_GB_operator_rowMajor_poisson1D(AB, &lab, &la);
    //write_GB_operator_rowMajor_poisson1D(AB, &lab, &la,
AB_row.dat");

    info = LAPACKE_dgbsv(LAPACK_ROW_MAJOR,la, kl, ku, NRHS, AB,
a, ipiv, RHS, NRHS);
```

Exercice 4 :

1. Utiliser la fonction BLAS dgbmv pour les deux stockage (ligne ou colonne).

Pour le fichier ***tp2_poisson1D_direct.c***

Comme le but est d'écrire les données dans "AB_col.dat" et "AB_row.dat" (cad les deux stockage ligne ou colonne). une analyse nous conduit a decommenter les 2 lignes :

//write_GB_operator_rowMajor_poisson1D(AB, &lab, &la, "AB_row.dat") ;
//write_GB_operator_colMajor_poisson1D(AB, &lab, &la, "AB_col.dat");

Pour le fichier ***lib_poisson1D.c***

On voit que la fonction **write_GB_operator_colMajor_poisson1D()** est vide donc il faut remplir de façon à avoir un stockage colonne donc on aura :

```
void write_GB_operator_colMajor_poisson1D(double* AB, int* lab, int* la, char* filename){
  //TODO
  int ii;
  int jj;
  FILE *file;
  // On parocours notre
  file=fopen(filename,"w");
  if(file!=NULL){
    // On parocours jsqu'a notre dimension n dans notre cas c'est la
    for(ii=0;ii<(*la);ii++){
    // on parcours jusqu'a la leading dimension
     for(jj=0;jj<(*lab);jj++){
         fprintf(file,"%lf\t",AB[ii*(*lab)+jj]);
     }
     fprintf(file,"\n");
   }
   fclose(file);
  }
  else{

  perror(filename);

  }

}
```

Il nous reste plus qu'a faire l'appel dans le main qui se trouve dans ***tp2_poisson1D_direct.c***

Exercice 5 :

1. Implémentez la méthode de factorisation LU du TD/TP 2 pour les matrices tridiagonales.

Avant d'implémenter il faut :

- Montrer avant qu'une matrice tridiagonale A s'écrit sous forme LU
- Pour ce faire on utilise la méthode gauss sans pivot ci-joint.

Exercice 5 : factorisation LU.

Soit A une matrice tridiagonale tel que :

$$A = \begin{pmatrix} a_1 & c_1 & 0 & \cdots & 0 & 0 \\ b_1 & a_2 & c_2 & \cdots & 0 & 0 \\ & \ddots & & & & \\ 0 & 0 & 0 & \cdots & a_{n-1} & c_{n-1} \\ 0 & 0 & 0 & \cdots & b_{n-1} & a_n \end{pmatrix} \quad \forall i \in [\![1,n]\!] \ a_i \neq 0$$

On applique la méthode de gauss sans pivot

- On a pas de critères autre que $a_i \neq 0$ on choisis pour notre exemple : $\pi_k = a^k_{kk}$

- $k=1$, $i=1$ alors $\pi_1 = a_1$ et $P_1 = I_4$

$$E_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{-b_1}{\pi_1} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ donc } A_2 = E_1 A_1 = \begin{pmatrix} a_1 & c_1 & 0 & 0 \\ 0 & \frac{-b_1 c_1}{\pi_1} + a_2 c_2 & 0 & \\ 0 & b_2 & a_3 & c_3 \\ 0 & 0 & b_3 & a_4 \end{pmatrix}$$

- $k=2$, $i=2$ alors $\pi_2 = a_2 - \frac{b_1 c_1}{\pi_1}$, $P_2 = I_4$

$$E_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{-b_2}{\pi_2} & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow A_3 = E_2 E_1 A_1 = \begin{pmatrix} \pi_1 & c_1 & \cdots & 0 \\ & \pi_2 c_2 & & \\ & & a_3 - \frac{b_2 c_2}{\pi_2} & c_3 \\ 0 & \cdots & b_3 & a_4 \end{pmatrix}$$

• $k=3$, $i=3$ $\quad \pi_3 = a_3 - \frac{b_2 c_2}{\pi_2}$

⋮

Après n itération aura le système suivant.

$$A_n = \begin{pmatrix} \pi_1 & c_1 & & \\ & \pi_2 & c_2 & 0 \\ & & \ddots & \ddots \\ & 0 & & c_{n-1} \\ & & & \pi_n \end{pmatrix} = \prod_{k=n-1}^{1} E_k$$

où : $E_k = (\gamma_{ij})$

donc on distingue les cas suivants $\begin{cases} \gamma_{ii} = 1 \\ \gamma_{i-1,i} = \frac{-b_i}{\pi_i} & i<j \\ \gamma_{ij} = 0 & i>j \end{cases}$

et $\begin{cases} \pi_1 = a_1 \\ \pi_k = a_k - \frac{b_{k-1} c_{k-1}}{\pi_{k-1}} \end{cases}$, $\forall k > 1$

(*) Decomposition LU

il faut montrer alors que

$$A = LU = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ e_1 & 1 & \cdots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & e_{n-1} & 1 \end{pmatrix} \begin{pmatrix} d_1 & c_1 & \cdots & 0 & 0 \\ 0 & d_2 & \cdots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & d_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & d_n \end{pmatrix}$$

pour notre cas notons $U = \begin{pmatrix} \pi_1 & c_1 & & 0 \\ & \ddots & \ddots & \\ & & \ddots & c_{n-1} \\ 0 & & & \pi_n \end{pmatrix}$

soit $L^{-1} = \prod_{k=n-1}^{1} E_k$ alors $L = \prod_{k=1}^{n-1} (E_k)^{-1}$

Calculons E_1^{-1} : $E_1 E_1^{-1} = I_n$

$$\begin{pmatrix} 1 & & & \\ -\frac{b_1}{\pi_1} & 1 & & 0 \\ 0 & & \ddots & \\ 0 & 0 & & 1 \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1n} \\ \vdots & & & \vdots \\ B_{n1} & \cdots & \cdots & B_{nn} \end{pmatrix} = \begin{pmatrix} 1 & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix}$$

On obtient colonne par colonne.

$$\begin{pmatrix} B_{11} = 1 \\ B_{21} = \frac{b_1}{\pi_1} \\ B_{31} = 0 \\ \vdots \\ B_{n1} = 0 \end{pmatrix} ; \begin{pmatrix} B_{12} = 0 \\ B_{22} = 1 \\ B_{32} = 0 \\ \vdots \\ B_{n2} = 0 \end{pmatrix} ; \text{ puis pour chaque colonne } j : B_{jj} = 1 \quad B_{ij} = 0 \quad i \neq j$$

Ainsi $E_k^{-1} = 2 I_n - E_k =$

$$\begin{pmatrix} 1 & & & & k \\ & \ddots & & & \vdots \\ k & \cdots & 1 & & \\ & & \frac{b_k}{\pi_k} & & \\ & & \vdots & & \\ & & 0 & & 1 \end{pmatrix}$$

On effectue le produit

$$L = E_1^{-1} \times E_2^{-1} \times \cdots \times E_{n-2}^{-1} \times E_{n-1}^{-1}$$

$$L = E_1^{-1} \times E_2^{-1} \times \cdots \times E_{n-3}^{-1}$$

$$\begin{pmatrix} 1 & & & & 0 \\ 0 & \ddots & & & \\ & \ddots & 1 & & \\ & & \frac{b_{n-2}}{\pi_{n-2}} & 1 & 0 \\ & & & \frac{b_{n-1}}{\pi_{n-1}} & 1 \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & & & 0 \\ \frac{b_1}{\pi_1} & \ddots & & \\ & \ddots & 1 & \\ 0 & & \frac{b_{n-1}}{\pi_{n-1}} & 1 \end{pmatrix}$$

Donc A s'écrit sous forme LU.

En notant (avec n la dimension de la matrice)
b= (bi) pour tout 1⩽i⩽n les coefficients diagonaux de la matrice A.
a= (ai) pour tout 1⩽i⩽n les coefficients sous-diagonaux de la matrice A.
c= (ci) pour tout 1⩽i⩽n les coefficients sur-diagonaux de la matrice A.

l=(li) pour tout 1⩽i⩽n-1
d= (di) pour tout 1⩽i⩽n
u= (ui) pour tout 2⩽i⩽n

Tel que l,d,u sont des coefficients des matrices L et U.
On aura la fonction facto suivante:

```
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
endfunction
```

2. Proposez une méthode de validation :
On considère la matrice A :

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

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 scilab suivant (inclut dans le fichier LU.sci) :

```
--> exec('/home/lougani/Bureau/CN_Lougani_Faouzi,
--> 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)
 l  =

    2.
    3.
 d  =

    1.
    2.
    3.

--> |
```

On voit que l et d correspondent à la décomposition LU de A.

3-Analyse de l'algorithme :

Le but de la factorisation LU c'est de l'exploiter pour résoudre le système linaire u=c ,(de la notation précédente) .
On distingue deux méthodes :

- 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.

Complexité en temps (temps de calcul) :
le temps pour la factorisation LU est de $O(n^2)$par contre on est à $O(n^3)$ pour l'élimination de gauss .

Complexité en mémoire :
la manipulation des matrices creuses avec d'élimination de Gauss nécessite le stockage en mémoire de tous les éléments compris dans la bande la plus large. Afin de réduire l'occupation mémoire on réordonne la matrice afin de rapprocher les éléments le plus possible de la diagonale principale.

Exercice 6 :

1. Appliquez la méthode de Jacobi, puis de Gauss-Seidel a une matrice tridiagonale.

```
function [x,errf,k]=jacobi(a,b,nit,eps,x0)
   n=size(x0,1);
   x=zeros(n,1);
   k=1;
   errf=zeros(nit,1);
   err=4;
   while k<nit
       x(1,1)=(1/a(1,1))*(b(1)-a(1,2)*x0(2));
   x(n,1)=(1/a(n,n))*(b(n)-a(n,n-1)*x0(n-1));
       for i=2:n-1
           x(i,1)=(1/a(i,i))*(b(i)-a(i,i-1)*x0(i-1)-a(i,i+1)*x0(i+1))
       end
       err=norm(x-x0);
       if err<eps
           break
       end
       errf(k,1)=err;
       k=k+1;
       x0=x

   end
endfunction

//Pour tester doit entrer
//a=[2,1,0;1,2,1;0,1,2]
//b=[3;4;3]
//x0=[0;0;0]
//[x,errf,k]=jacobi(a,b,40,1e-14,x0)
//tel que nit=nb iterations qu'on veut et la precision qu'on veut=eps=1e-14
```

```
function [x,errf,k]=Gseidel(a,b,nit,eps,x0)

   n=size(x0,1);
   x=zeros(n,1);
   k=1;
   errf=zeros(nit,1);
   err=0;
   while k<nit
       x(1,1)=(1/a(1,1))*(b(1)-a(1,2)*x0(2));
       //on initialise les valeur de la matrice
       for i=2:n-1
           x(i,1)=(1/a(i,i))*(b(i)-a(i,i-1)*x(i-1)-a(i,i+1)*x0(i+1))
       end
        x(n,1)=(1/a(n,n))*(b(n)-a(n,n-1)*x(n-1));
       err=norm(x-x0);
       if err<eps
           break
       end
       errf(k,1)=err;
       k=k+1;
       x0=x
   end

endfunction

//Pour tester doit entrer
//a=[2,1,0;1,2,1;0,1,2]
//b=[3;4;3]
//x0=[0;0;0]
//[x,errf,k]=jacobi(a,b,40,1e-14,x0)
//tel que nit=nb iterations qu'on veut et la precision qu'on veut=eps=1e-14
```

2.Analysez la complexité de vos algorithmes. Quelles modifications pouvez-vous apportez pour diminuer la complexité :
Pour les deux algorithme :

- Si on est dans le cas de matrices denses ,la complexité de chaque itération est de $O(n^2)$ car c'est un produit de matrice vecteur et aussi dans ce cas on fait un GAXPY
- Pour notre cas on traites les matrices creuses (cad stockage diagonales),la complexité est égale à O(nb_lignes*nb_colonnes), donc ce qui vaut O(n) avec n est grand

Pour diminuer on peut exploiter la structure de la matrice ,par exemple faire un appel à une seule boucle (Pour les **i**) .

On ne peut pas dire que la complexité de Jacobi est égale à une valeur et celle de Gauss-Seidel est égale à une autre valeur ,car dans les deux cas comme déjà vue en cours et en le validant avec leurs codes, on dépend de plusieurs facteurs comme :

- Type de matrice (creuse ,denses)
- mode de Stockage
- Taille du problème (nombre d'itérations)
- format de données dans certains cas (double précision)

3. Effectuez quelques itérations de ces méthodes pour le problème Poisson 1D, pour n= 3.

On va dérouler à la main les formules vues en cours

Cas Jacobi :

On a la formule de Jacobi :

$$\xi_i^{k+1} = \frac{1}{a_{ii}}(B(i) - \sum_{j=1}^{n} a_{ij} - \xi_j^k) \quad pour\ i = 1, .., n\ \ et\ \ i \neq j$$

Si on a une matrice tridiagonale A tel que :

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} B = \begin{pmatrix} 3 \\ 5 \\ 3 \end{pmatrix}$$

Alors : $X^0 = (2, 1, 1)$

L'itération 1 :

$$\xi_1^1 = \frac{1}{a_{11}}(B(1) - a_{12}\xi_2^0 - a_{13}\xi_3^0)$$
$$\xi_1^1 = \frac{1}{1}(3 - 2 * 1 - 0 * 1)$$
$$\xi_1^1 = 1$$

L'itération 2 :

$$\xi_2^1 = \frac{1}{a_{22}}(B(2) - a_{21}\xi_1^0 - a_{23}\xi_3^0)$$
$$\xi_1^1 = \frac{1}{1}(5 - 2 * 2 - 2 * 1)$$
$$\xi_1^1 = -1$$

L'itération 3 :

$$\xi_3^1 = \frac{1}{a_{22}}(B(3) - a_{31}\xi_3^0 - a_{32}\xi_2^0)$$
$$\xi_1^1 = \frac{1}{1}(3 - 0 * 1 - 2 * 1)$$
$$\xi_1^1 = 1$$

Cas Gauss-Seidel :

On a la formule de Gauss-Seidel:

$$\xi_i^{k+1} = \frac{1}{a_{ii}}(B(i) - \sum_{j=1}^{i-1} a_{ij}\xi_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}\xi_j^k)$$

On applique pour la matrice tridiagonale précédente on aura :

$$X^0 = (2, 1, 1)$$

L'itération 1 :

$$\xi_1^1 = \frac{1}{a_{11}}(B(1) - a_{12}\xi_2^0 - a_{13}\xi_3^0)$$
$$\xi_1^1 = \frac{1}{1}(3 - 2 * 1 - 0 * 1)$$
$$\xi_1^1 = 1$$

L'itération 2:

$$\xi_2^1 = \frac{1}{a_{22}}(B(2) - a_{21}\xi_1^1 - a_{23}\xi_3^0)$$
$$\xi_1^1 = \frac{1}{1}(5 - 2 * 1 - 2 * 1)$$
$$\xi_1^1 = 1$$

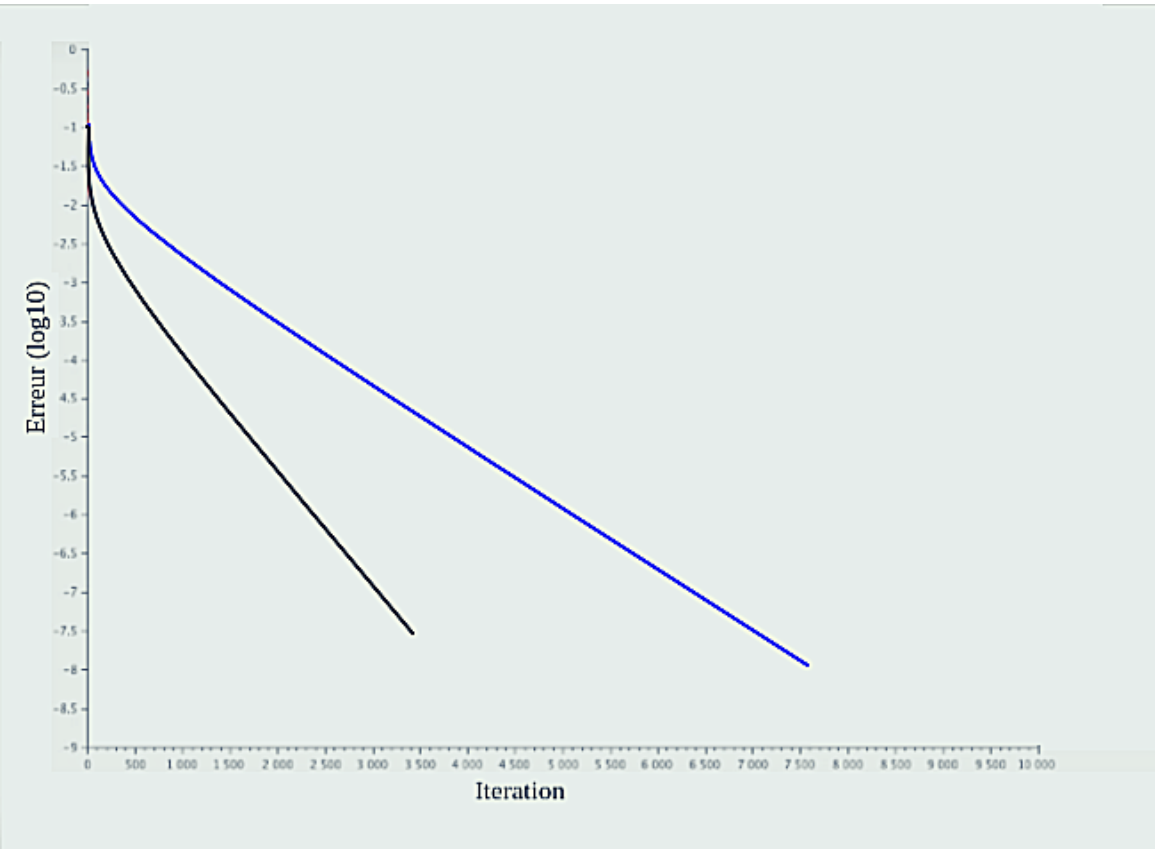
L'itération 3:

$$\xi_3^1 = \frac{1}{a_{33}}(B(3) - a_{31}\xi_1^1 - a_{32}\xi_2^1)$$
$$\xi_1^1 = \frac{1}{1}(3 - 0 * 2 - 2 * 1)$$
$$\xi_1^1 = 1$$

On peut déduire alors :

$$X^1 = (1, 1, 1)$$

5. Analysez la convergence :
On fait un plot2d sous scilab de l'erreur et l itération on aura le graphe suivant
La courbe en bleu c'est celle de Jacobi
La courbe en noir c'est celle de Gauss-Seidel



-On fixe notre tolérance a 10^8 et on voit qu'on converge à 10^8 .
-Jacobi converge 2 fois lentement en nombre d'itération que Gauss-Seidel pour des tailles N plus petites donc ça valide notre analyse théorique (faite au cours)
-On a moins d'erreur en Gauss-Seidel des que N (le nombre d'itérations augmente) donc c'est la fonction a choisir (optimale)

Complexité en terme de temps ou comme vue en cours complexité arithmétique :

- Mon code stocke que les diagonales ,donc la complexité est linéaire
- Chaque itération de Gauss-Seidel va coûter plus cher (6n) que celle de Jacobi (2n)
- Si on fait une mesure de temps avec (tic,toc) on aura :
  1. le temps de Jacobi=0.1455
  2. le temps de Gauss=0.2822
  3. Donc même si il ya deux fois moins d'itérations sur Gauss-Seidel ça coûte deux fois plus chère.
  4. En résumé faire moins d'itérations c'est avoir plus de qualité numérique

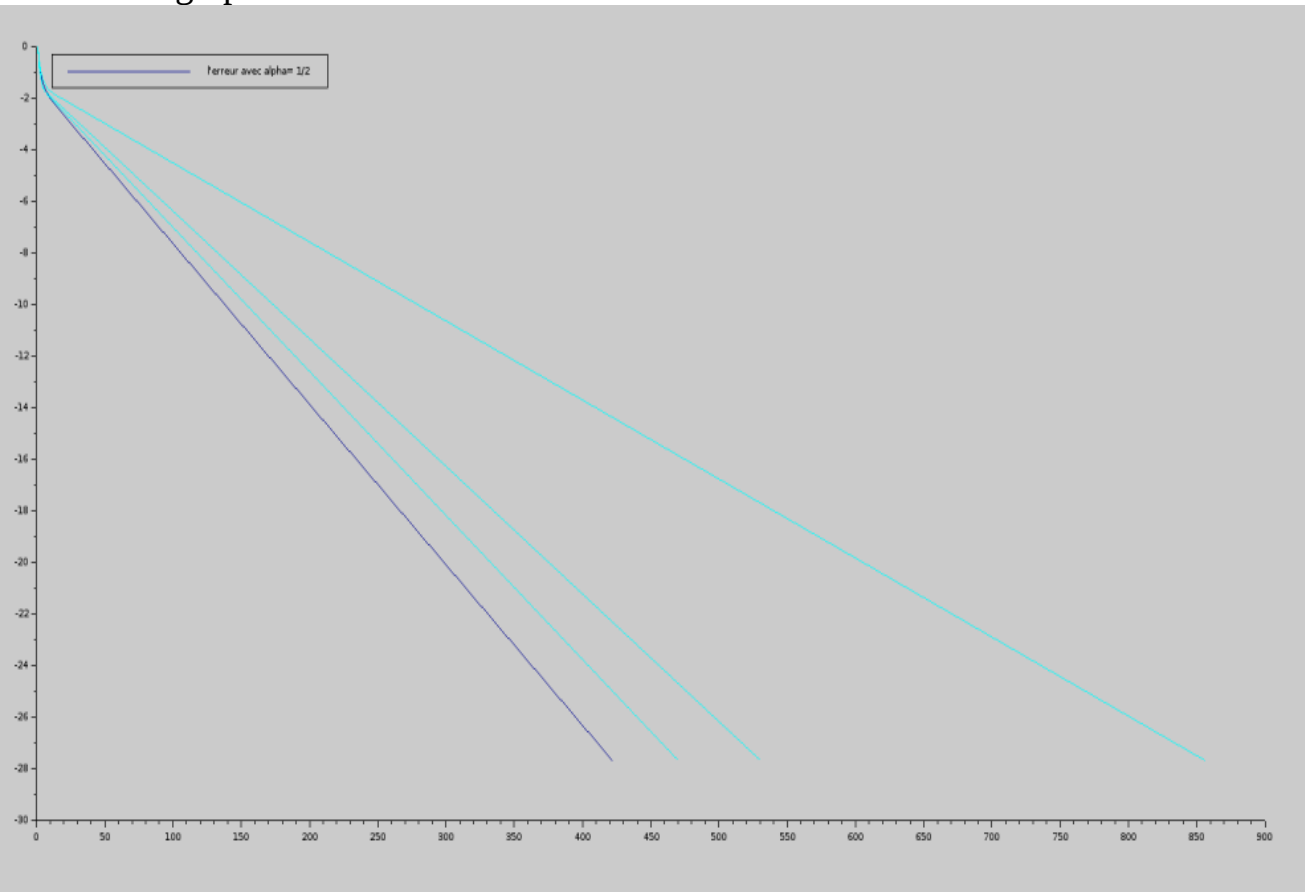
Complexité mémoire:si notre algorithme stocke des éléments en mémoire

Exercice 7 :

1/Code Scilab:
Appliquez des résultats théoriques de Richardson sur le problème de Poisson 1D (équation de la chaleur).
On a le code scilab suivant :

```
function[x,relres,resvec,it]=richardson_poisson_1d(A,b,tol,maxit,x0,alpha)
    it=0;
    res=b-A*x0;
    relres=norm(b-A*x0)/norm(b);
    while (relres>tol)&(it<maxit)
    it=it+1;
    x=x0+alpha;
    res=b-A*x0;
    relres=norm(b-A*x0)/norm(b);
    resvec(it)=relres;
    x0=x;
    end
endfunction
```

Pour la validation on donne : maxit=100 ; tol=1e-8; x0=0 ;
2/Si on donne a ce programme :
alpha=0,25
alpha=0,3
alpha=0,4
alpha=0,5
On aura le graphe suivant :



les 3 lignes bleu clair représentent
alpha=0,25
alpha=0,3
alpha=0,4

En violet c'est **la valeur optimale alpha=0,5** ou l erreur converge rapidement .

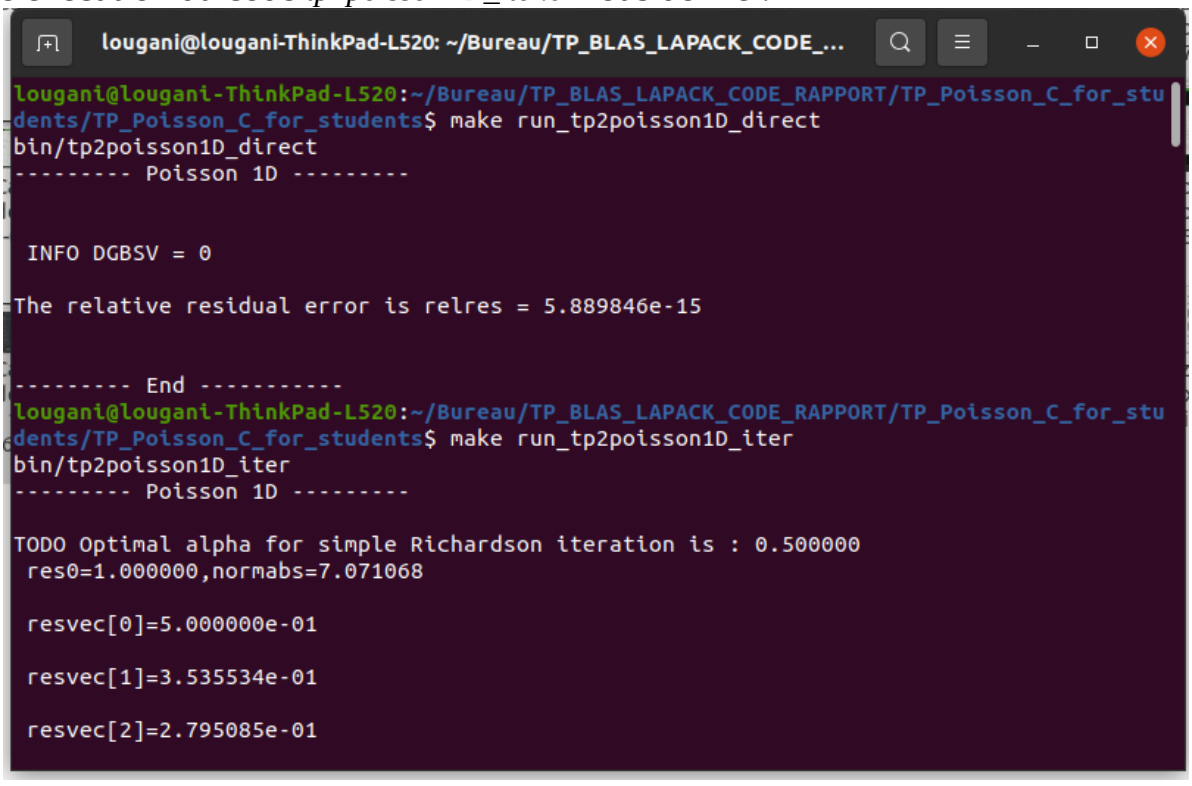
2/Complexité en terme de temps :
Si on mesure avec (tic;toc;) le temps de Richardson on aura :
Le temps =0.1043
Donc Richardson est bien meilleur que Jacobi

3/Résultats :
-Richardson n'est pas équivalente a Jacobi Pour l'alpha optimal.
-La complexité c'est Richardson qui coûte moins cher.
-Le meilleur Richardson scalaire possible c'est Jacobi.

Implémentation C :

Important : Le code C disponible sur mon git n'est pas de mon implémentation,j'ai essayé de comprendre et d'assemblé les parties que le professeur nous donne à chaque séance ,mon but est d'analyser les résultats ,les comparer avec ceux obtenus en scilab .

Une exécution du code *tp2poisson1D_iter.c* nous donne :



- On voit mème en C on a **la valeur optimale alpha=0,5**
- Par contre **resvec[i] pour i=0,….,99** on a pas les mêmes résultats qu'en scilab en C on a plus de précision en Calcul numérique .
- **La valeur de relres en C <La valeur de relres scilab**
- En terme de Complexité ici on est à O(n) car général bande (GB) contrairement à l'implémentation en scilab on est en $O(n)+O(n^2)$. Car matrices creuses

L'implémentation en C est mieux optimale que celle en scilab pour notre Richardson

A retenir :
L implémentation d'algorithmes pour le calcul numérique a besoins d une importante prise en considération sur l'accès aux données, la complexité des algorithmes et le réarrangement des boucles pour optimiser les performances.

Les fichiers/rapport sont disponibles sur :

https://github.com/lougani-faouzi/calcul_numerique/tree/master/TP_BLAS_LAPACK_CODE_RAPPORT/