

TP de Calcul Numérique

Nicolas BOUTON

2020

1 Exercice 1

1.0.1 Question 1

Approximation de T'' au moyen d'un schéma centré d'ordre 2.

Developpement limité :

$$\begin{aligned}T(x_i + h) &= T(x_i) + h \left(\frac{\partial T}{\partial x} \right)_i + h^2 \left(\frac{\partial^2 T}{\partial x^2} \right)_i + O(h^2) \\T(x_i - h) &= T(x_i) - h \left(\frac{\partial T}{\partial x} \right)_i + h^2 \left(\frac{\partial^2 T}{\partial x^2} \right)_i + O(h^2)\end{aligned}$$

On somme et on inverse le signe :

$$\begin{aligned}-T(x_i + h) + 2T(x_i) - T(x_i - h) &= -h^2 \left(\frac{\partial^2 T}{\partial x^2} \right)_i + O(h^2) \\ \frac{-T(x_i + h) + 2T(x_i) - T(x_i - h)}{h^2} &= - \left(\frac{\partial^2 T}{\partial x^2} \right)_i\end{aligned}$$

Donc

$$T'' = \frac{T(x+h) - 2T(x) + T(x-h)}{h^2}$$

1.0.2 Question 2

On a :

$$-k \left(\frac{\partial^2 T}{\partial x^2} \right)_i = g_i, k > 0$$

On se permet de négligé k car c'est une constante dans nos prochain calcul:

$$-T(x_i + h) + 2T(x_i) - T(x_i - h) = h^2 g_i$$

On écrit le système d'équation :

$$\begin{array}{ll} u_0 = T_0 & i = 0 \\ -u_0 + 2u_1 - u_2 = h^2 g_1 & i = 1 \\ \dots & \dots \\ -u_{k-1} + 2u_k - u_{k+1} = h^2 g_k & i = k \\ \dots & \dots \\ -u_{n-1} + 2u_n - u_{n+1} = h^2 g_n & i = n \\ u_n = T_n & i = n + 1 \end{array}$$

Avec les conditions aux bords on obtient :

$$\begin{array}{l} 2u_1 - u_2 = h^2 g_1 + T_0 \\ -u_{n-1} + 2u_n = h^2 g_n + T_n \end{array}$$

Donc on explicite le système linéaire $Au = g$:

$$A = \left[\begin{array}{ccccccc|c} 2 & -1 & 0 & - & - & - & 0 & \\ -1 & 2 & -1 & . & & & & | \\ 0 & -1 & . & . & . & & & | \\ | & . & . & . & . & . & & | \\ | & & . & . & . & -1 & 0 & | \\ | & & & . & -1 & 2 & -1 & | \\ 0 & - & - & - & 0 & -1 & 2 & \end{array} \right]$$

$$u = \left[\begin{array}{c} T_1 \\ | \\ T_n \end{array} \right]$$

$$g = \left[\begin{array}{c} h^2 T_1 + T_0 \\ h^2 T_2 \\ | \\ h^2 T_{n-1} \\ h^2 T_n + T_1 \end{array} \right]$$

Comme il n'y a pas de source de chaleur, on a $\forall i \in [1, n] : h^2 g_i = 0$

$$\text{D'où } g = \begin{bmatrix} T_0 \\ 0 \\ | \\ 0 \\ T_1 \end{bmatrix}$$

Nous avons donc :

$$\begin{bmatrix} 2 & -1 & 0 & - & - & - & 0 \\ -1 & 2 & -1 & . & & & | \\ 0 & -1 & . & . & . & & | \\ | & . & . & . & . & . & | \\ | & & . & . & . & -1 & 0 \\ | & & & . & -1 & 2 & -1 \\ 0 & - & - & - & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ | \\ | \\ | \\ T_{n-1} \\ T_n \end{bmatrix} = \begin{bmatrix} T_0 \\ 0 \\ | \\ | \\ | \\ 0 \\ T_1 \end{bmatrix}$$

Et la solution analytique qui se dégage est :

$$T(x) = T_0 + x(T_1 - T_0)$$

2 Exercice 2

2.1 Arch

2.1.1 Bibliothèque

Pour l'installation des bibliothèques **cblas** et **lapacke** :

```
$ sudo pacman -S cblas lapacke
```

2.1.2 Makefile

Il faut modifier la ligne qui link les librairies en linkant la bibliothèque

cblas:

```
#
```

```
# - librairies
```

```
LIBS=-llapacke -lcblas -lm
```

3 Exercice 3

3.1 Question 1

Les matrices pour utiliser **BLAS** et **LAPACK** en **C** sont allouées et déclarées de la même manière que les tableaux en **C**. Mais elles doivent

être stockées dans l'un des formats suivant :

- stockage conventionnel en 2 dimension (ex: `int tab[10][10]`)
- stockage compact pour les matrices symétrique, hermitienne et triangulaire (stockage dans un tableau à 1 dimension des éléments de la matrice supérieur ou inférieur)
- stockage bandes pour les matrices à bandes (cad que les diagonales autour de la diagonale principale contiennent la plupart des NNZ) (GB)
- utilisation de 2 ou 3 tableaux à 1 dimension pour stocker les matrices bidiagonale et tridiagonale respectivement

source : <http://performance.netlib.org/lapack/lug/node121>

3.2 Question 2

- Les constantes **LAPACK_ROW_MAJOR** et **LAPACK_COL_MAJOR** signifie la priorité ligne ou colonne respectivement de la représentation de la matrice.
- Effectivement, cet argument sert si on utilise un stockage par priorité ligne ou colonne car il faut préciser si on a utilisé une priorité ligne ou colonne pour stocker la matrice pour pouvoir faire les bons calculs.

3.3 Question 3

La **leading dimension** permet de savoir qu'elle élément correspond à la prochaine colonne ou la prochaine ligne suivant le stockage colonne ou ligne respectivement.

- Si on choisit un stockage priorité ligne, alors la **leading dimension** correspond au nombre d'élément d'une ligne pour pouvoir accéder à la ligne suivante.
- Si on choisit un stockage priorité colonne, alors la **leading dimension** correspond au nombre d'élément d'une colonne pour pouvoir accéder à la colonne suivante.

3.4 Question 4

3.4.1 Résumé

La fonction LAPACKE_dgbsv permet de calculer le résultat d'un système linéaire du type $A * X = B$, avec \mathbf{X} l'inconnu, \mathbf{A} une matrice et \mathbf{B} le second membre, où \mathbf{X} et \mathbf{B} peuvent être des vecteurs ou des matrices.

3.4.2 Argument

Elle prend en argument la dimension de la matrice, le nombre du sous-diagonale ainsi que de sur-diagonale, la leading dimension de \mathbf{A} et de \mathbf{B} , le nombre de colonne de \mathbf{B} ainsi que tableau d'entier pour stoker les indices de permutation.

3.4.3 Implémentation

Cette fonction implémente une décomposition \mathbf{LU} à pivot partiel et la méthode de dessente et de remonté.

3.4.4 Note importante

Pour la factorisation \mathbf{LU} , la fonction a besoin d'un vecteur de travail ou il stockera les pivots. Suivant le stockage choisis on rajoutera une ligne ou une colonne avant de stocker notre matrice car le vecteur doit apparaître en premier.

3.4.5 Sources

<http://www.math.utah.edu/software/lapack/lapack-d/dgbsv.html>

3.5 Question 5

A titre comparatif nous prendrons une matrice de taille 10 x 10.

3.5.1 Stockage priorité colonne

0.000000	0.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	-1.000000
0.000000	-1.000000	2.000000	0.000000

3.5.2 Stockage priorité ligne

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000
2.000	2.000	2.000	2.000	2.000	2.000	2.000	2.000	2.000	2.000
-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	0.000

Pour que le tableau soit affichable nous avons dû enlever les 3 derniers 0 de la sortie, mais cela ne change pas le résultat.

3.5.3 Remarques

Nous pouvons apercevoir que nous avons le bon résultat, étant donné que la première ligne qui est réservée à **BIAS** pour son vecteur de travail est constituée de zéro ainsi que le premier élément de la diagonale supérieure et le dernier élément de la diagonale inférieure.

De plus pour vérifier le résultat il suffit de calculer la transposée de l'une des matrices et la comparer à la deuxième car elles doivent être égales. Ici la transposée d'une des 2 matrices est égale à la deuxième.

4 Exercice 5

4.1 Question 1

4.1.1 Equation

La fonction `cblas_dgbmv` permet de calculer l'équation suivante :

$$y = \alpha * A * x + \beta * y$$

Dans notre cas on a l'équation suivante :

$$A * u = b$$

Qu'on peut écrire avec les notations de l'équation juste au dessus :

$$A * y = x$$

Avec la question précédente on a calculer y. On peut donc tester la fonction sur l'équation suivante :

$$x = 1 * A * y + 0 * x$$

4.1.2 Argument

La fonction **cblas_dgbmv** prends à peut près les mêmes paramètres que **LAPACKE_dgbsv** plus les coefficient α et *beta*, le nombre de ligne et de colonne de la matrice en format classic ainsi qu'un paramètre qui indique si la matrice est une transposé ou non.

La fonction prend comme précondition que y et x soit des vecteurs.

4.1.3 Note importante

Contrairement à la fonction **LAPACKE_dgbsv**, la fonction **cblas_dgbmv** n'attend pas à ce qu'on lui laisse un vecteur au début de la matrice.

4.2 Question 2

Comme méthode de validation, nous proposons de calculé l'erreur relative suivante.

Etant donné que nous calculons l'équation suivante :

$$b = A * u$$

Nous allons calculer l'erreur relative suivante :

$$relres = \frac{\|EX_SOL - B\|}{\|B\|}$$

où **B** est en fait le résultat calculé et **EX_SOL** la solution exacte. Le résultat exacte de **B** est stocké dans le fichier **B.dat**.

4.2.1 Priorité Ligne

Le résultat est stocké dans le fichier **Y_row.dat**.

————— Poisson 1D —————

INFO DGBSV = 0

The relative residual error is relres = 1.764638e-16

DGBSV :

The relative residual error is relres = 5.102197e-16

————— End —————

Pour la priorité ligne il faut stocké les lignes en colonne et mettre comme leading dimension la leading dimension des colonnes donc conformément au code FORTRAN de BLAS qui stipule que la transposé de la matrice est calculé et utilisé à la place de la matrice d'entrée si jamais on entre l'argument **CblasNoTrans**.

4.2.2 Priorité Colonne

Le résultat est stocké dans le fichier **Y_col.dat**.

————— Poisson 1D —————

INFO DGBSV = 0

The relative residual error is relres = 1.764638e-16

DGBSV :

The relative residual error is relres = 5.102197e-16

————— End —————

4.2.3 Explication

L'erreur relative de la fonction `cblas_dgbmv` est un peu plus élevée que l'erreur relative de la fonction `LAPACKE_dgbmv` qui peut être expliqué par le fait que le bit de signe ne change pas et nous nous retrouvons avec des zéro négatif au lieu de positif mais l'erreur reste acceptable car il est de l'ordre de la précision machine.

4.2.4 Conclusion

Nous pouvons donc valider l'appel à blas pour la priorité colonne mais pas pour la priorité ligne pour le moment.

5 Exercice 5

5.1 Question 1

5.1.1 Implémentation scilab

Voir le premier rapport.

5.1.2 Implémentation C

5.1.3 Code

Le code est séparé en deux fichiers, un contenant le code des fonctions qui se trouve dans `src/lib.lu.c`, et un fichier qui contient le code de test qui est dans `src/tp2_facto.lu.c`.

Pour compiler : `$ make tp2facto.lu`

Pour exécuter : `$ make run_tp2facto.lu`

5.1.4 Explication

Nous avons effectué une résolution d'un système linéaire tel que $A * x = b$ ou bien même $LU * x = b$ grâce à une factorisation LU.

Puis avec une méthode de descente : $L * y = b$.

Puis avec une méthode de remonté : $U * x = y$.

Pour cela nous avons dû utiliser 1 matrice A et deux vecteurs X et B où A est une matrice tridiagonale stockée en format général bandes prioritaire

colonne de taille $3 * la$ (où **la** est la leading dimension de A), X un vecteur de taille **la** ainsi que B un vecteur de taille **la**.

Et nous effectuons les opérations suivantes :

$$\begin{cases} A = LU(A) \\ L * X = B \\ U * B = X \end{cases}$$

- on effectue la factorisation LU de A que l'on stocke dans A
- on effectue la méthode de descente sur L et B et l'on stocke le résultat dans X
- on effectue la méthode de remonté sur U et X et l'on stocke le résultat dans B
- le résultat est bien dans le second membre c'est-à-dire B

5.2 Question 2

Pour la méthode de validation, nous allons calculer l'erreur relative avant et s'assurer que l'erreur est proche de la précision machine.

5.2.1 Calcul

$$relres = \frac{\|EX_SOL - B\|}{\|B\|}$$

où EX_SOL est la solution exacte et B la solution calculé.

5.2.2 Résultat

———— Facto LU ————

The relative residual error is relres = 1.764638e-16

———— End ————

5.2.3 Conclusion

Le résultat de l'erreur est bien proche de la précision machine donc on peut valider notre algorithme et notre implémentation.

5.2.4 Comparaison avec `cblas_dgbmv`

L'erreur relative pour les deux calculs sont les mêmes, en effet la fonction de BLAS fait les mêmes que ce que l'ont a fait pour le cas particulier d'une matrice tridiagonale mais sur des matrices avec plus ou moins de diagonales. Il faudra étudier le temps que prends les deux méthodes pour pouvoir mieux les comparer.

6 Exercice 6

6.1 Question 1

On ajoute une nombre d'itération maximale pour éviter une boucle infini à cause de l'une des raisons suivantes :

- la solution ne converge pas
- la solution converge mais met très longtemps à convergé et nous voulons limité le temps

6.1.1 Jacobi

Algorithm 1: Applique le méthode de Jacobi

Data: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $max_count, \epsilon \in \mathbb{R}$
Result: $x \in \mathbb{R}^n$, $count, relres \in \mathbb{R}$, $resvec \in \mathbb{R}^n$
 $D = \text{zeros}(n, n)$ ▷ Declare matrice D
for $i = 1 : n$ **do**
 $D(i, i) = \frac{1}{A(i, i)}$ ▷ Init. matrice D
end
 $count = 0$ ▷ Init. compteur
 $x = \text{zeros}(n)$ ▷ Init. vecteur résultat à 0
 $resvec = \text{zeros}(max_count)$ ▷ Init. vecteur relres à 0
 $normb = \text{norm}(b)$ ▷ Init. norme de b
 $r = b - A * x$ ▷ Init. résidu
 $relres = \text{norm}(r) / normb$ ▷ Init. erreur relative
while $relres > \epsilon$ and $count < max_count$ **do**
 $x = x + D * r$ ▷ Calcul de Richardson
 $r = b - A * x$ ▷ Calcul du résidu
 $relres = \text{norm}(r) / normb$ ▷ Calcul de l'erreur relative
 $resvec(count) = relres$ ▷ Sauv. vecteur
 $count = count + 1;$ ▷ Inc. compteur
end

6.1.2 Gauss Seidel

Algorithm 2: Applique le méthode de Gauss Seidel

Data: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $max_count, \epsilon \in \mathbb{R}$
Result: $x \in \mathbb{R}^n$, $count, relres \in \mathbb{R}$, $resvec \in \mathbb{R}^n$

D = zeros(n, n) ▷ Declare matrice D
for $i = 1:n$ **do**
 | D(i, i) = A(i, i) ▷ Init. matrice D
end
E = zeros(n, n) ▷ Declare matrice E
for $i = 2:n$ **do**
 | E(i, i - 1) = - A(i, i - 1) ▷ Init. matrice E
end
DE = inverse(D - E) ▷ Inverse (D - E)
count = 0 ▷ Init. compteur
x = zeros(n) ▷ Init. vecteur résultat à 0
resvec = zeros(max_count) ▷ Init. vecteur relres à 0
normb = norm(b) ▷ Init. norme de b
r = b - A * x ▷ Init. résidu
relres = norm(r) / normb ▷ Init. erreur relative
while $\|r\| > \epsilon$ *and* $count < max_count$ **do**
 | x = x + DE * r ▷ Calcul de Richardson
 | r = b - A * x ▷ Calcul du résidu
 | relres = norm(r) / normb ▷ Calcul erreur relative
 | resvec(i) = relres ▷ Stocke erreur relative
 | count = count + 1 ▷ Inc. compteur
end

6.2 Question 2

Pour nos deux algorithmes, nous voyons que la complexité en temps ne peut pas être explicite car tout dépend de l'erreur. D'après le cours on était réduit à la formule ci dessous :

Pour Jacobi :

$$k = -\frac{p}{2 * \log(\cos(\pi * h))}$$

Pour Gauss Seidel :

$$k = -\frac{p}{\log(\cos(\pi * h))}$$

avec

- k le nombre d'itération
- p le nombre de chiffre significatif de l'erreur
- h un entier quelconque

De plus pour toute les complexités nous allons uniquement prendre en considération les informations de base des algorithmes qui permettent de calculé le résultat. C'est-à-dire que nous prendraon pas en comptes les vecteurs ainsi que les calculs qui sauvegarde les erreurs relative par exemple.

6.2.1 Jacobi

Complexité :

- en temps : indéterminé (dépend des paramètre d'entré)
- en temps d'une itération : $O(2n^2 + 2n)$ (2 produit **matrice x matrice** et 2 adition **vecteur x vecteur**)
- en espace : $O(2n^2 + 3n)$ (2 matrice + 3 vecteur)

6.2.2 Gauss Seidel

Complexité :

- en temps : indéterminé (dépend des paramètre d'entré)
- en temps d'une itération : $O(2n^2 + 2n)$ (2 produit **matrice x matrice** et 2 adition **vecteur x vecteur**)
- en espace : $O(4n^2 + 3n)$ (4 matrice + 3 vecteur)

Pour améliorer nos algorithmes d'un point de vu mémoire ainsi qu'en tant on pourrai transformé les matrices en format général bandes, on pourrai facilement calculé l'inverse des matrices pour les 2 méthodes car ici on est dans un cas particuliers des matrices tridiagonnales. Et dans la boucle **do...while** on fera appelle à la fonction de blas **dgvsv** pour nos 2 opérations car se sont en faites des systèmes linéaires. Il faudra juste multiplié par -1 le produit **matrice x vecteur** de la deuxième opération pour les 2 méthodes.

6.2.3 Amélioration pour Jacobi

Pour l'équation de la chaleur, sur la diagonale de la matrice A on a les même valeur et donc l'inverse sera aussi la même valeur pour toute la diagonale donc on peut réduire le produit **matrice x vecteur** ($D * r$) en produit **scalaire x vecteur**.

Algorithm 3: Applique le méthode de Jacobi

Data: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $max_count, \epsilon \in \mathbb{R}$
Result: $x \in \mathbb{R}^n$, $count, relres \in \mathbb{R}$, $resvec \in \mathbb{R}^n$
 $D = 1 / 2$ ▷ Simplification
 $count = 0$ ▷ Init. compteur
 $x = \text{zeros}(n)$ ▷ Init. vecteur résultat à 0
 $resvec = \text{zeros}(max_count)$ ▷ Init. vecteur relres à 0
 $normb = \text{norm}(b)$ ▷ Init. norme de b
 $r = b - A * x$ ▷ Init. résidu
 $relres = \text{norm}(r) / normb$ ▷ Init. erreur relative
while $relres > \epsilon$ **and** $count < max_count$ **do**
 $x = x + D * r$ ▷ Calcul de Richardson
 $r = b - A * x$ ▷ Calcul du résidu
 $relres = \text{norm}(r) / normb$ ▷ Calcul erreur relative
 $resvec(i) = relres$ ▷ Stocke erreur relative
 $count = count + 1$ ▷ Inc. compteur
end

Ce qui nous conduit au complexité suivantes :

- en temps : indéterminé (dépend des paramètre d'entrée)
- en temps d'une itération : $O(n^2 + 3n)$ (1 produit **matrice x matrice**, 1 produit **scalaire x vecteur** et 2 addition **vecteur x vecteur**)
- en espace : $O(n^2 + 3n)$ (1 matrice + 3 vecteur)

6.3 Question 3

Pour $n = 3$ on obtient le nombre d'itération suivante :

ϵ	jacobi	gauss seidel
10^{-1}	9	5
10^{-2}	16	8
10^{-3}	22	12
10^{-4}	29	15
10^{-5}	36	18
10^{-6}	42	22
10^{-7}	49	25
10^{-8}	56	28

Ce qui confirme l'approximation analytique qui dit que **Jacobi** à besoin de 2 fois plus d'itération que **Gauss Seidel** pour convergé.

6.4 Question 4

librairie : voir dans le fichier **src/iterative.sci**

Explication :

Simple implémentation qui respecte les algorithmes ci-dessus, rien à dire de spécial.

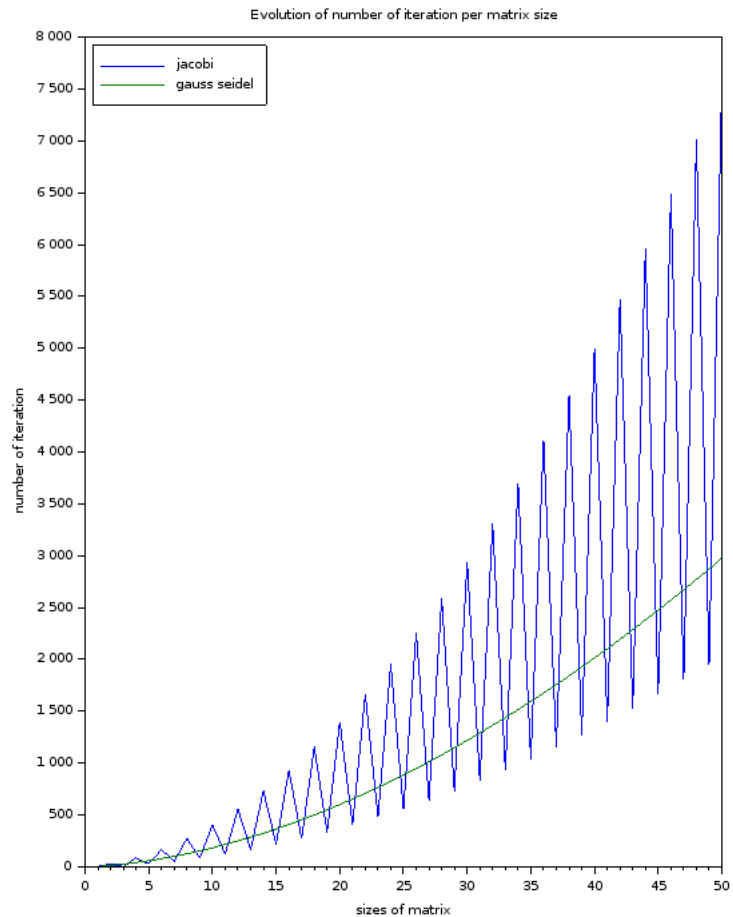
script test : voir dans le fichier **src/tp2_iterative.sci**

Explication :

Il y a deux boucles, une qui vari la taille de la matrice et une qui varie l'erreur souhaité pour voir l'évolution du nombre d'itération.

6.4.1 Varié la taille de la matrice

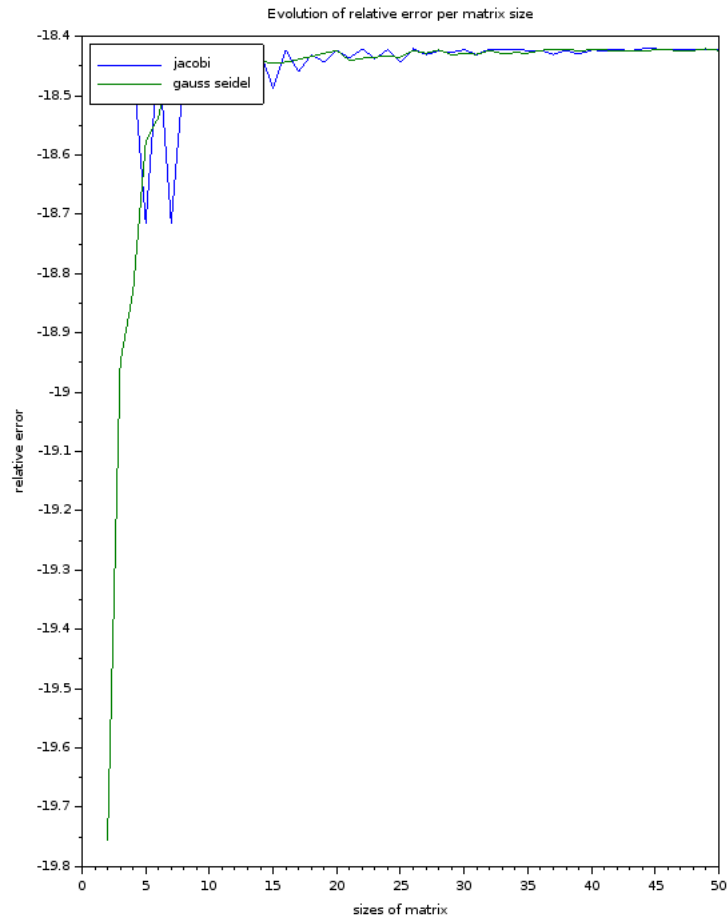
Ici un graphe qui montre l'évolution du nombre d'itération nécessaire en fonction de la taille de la matrice A afin d'atteindre la convergence pour une erreur de 10^{-8} .



Nous pouvons appercevoir que le nombre d'itération évolue de manière **quadratique** ce qui est normal car nous stockons les matrices en format classique et donc elle augmente le nombre d'élément de façon quadratique et c'est donc plus difficiles d'arrivé à convergence.

Nous voyons aussi que pour jacobi le nombre d'itération varie selon la parité de la taille de la matrice, en effet lorsque la taille est pair elle coute moins d'itération que ses voisins impair.

Erreur relative avant :

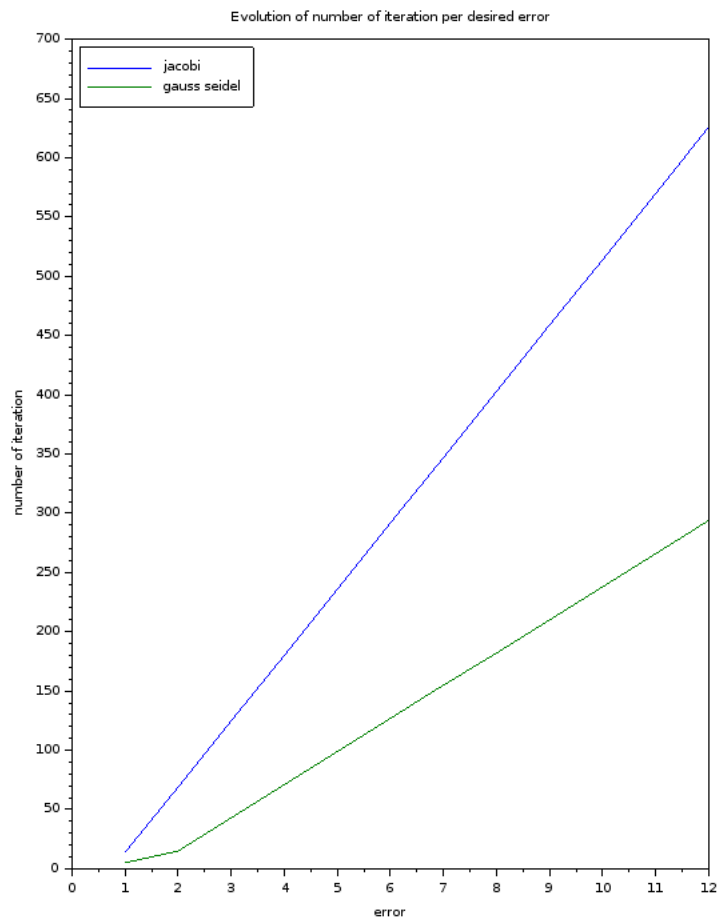


Nous pouvons voir que l'erreur avant est très bonne étant donné qu'on a fixé l'erreur machine à 10^{-16} et nous avons des erreurs supérieures à celle-ci pour des matrices de taille **50 x 50** maximum.

6.4.2 Varié l'erreur souhaité

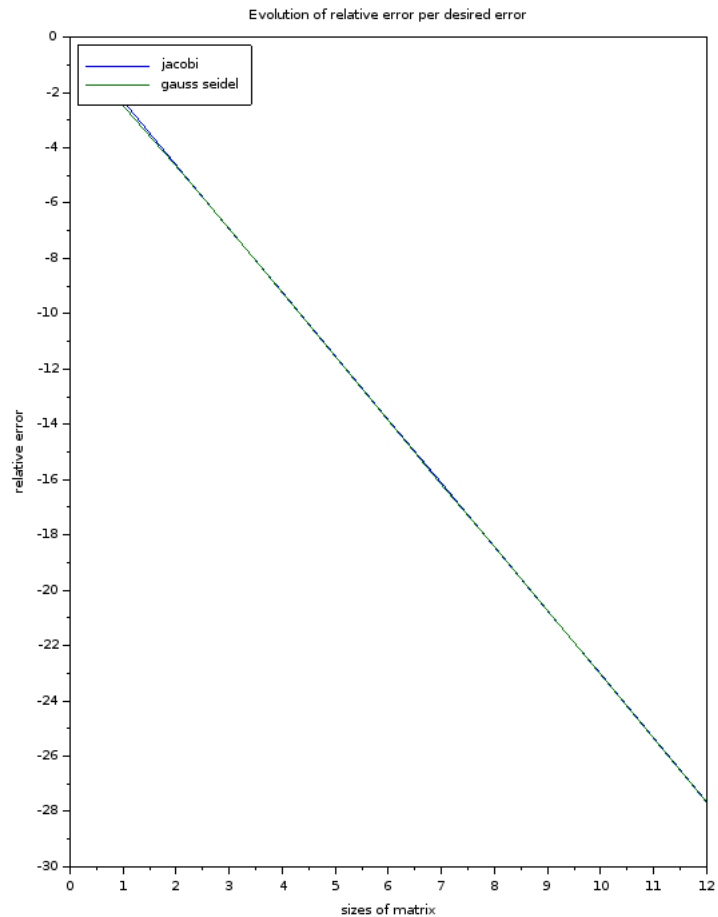
Ici un graphique qui montre l'évolution du nombre d'itération nécessaire en fonction de l'erreur souhaitée afin d'atteindre la convergence pour une taille

de matrice $n = 10$, soit une matrice 10×10 .



Nous pouvons apercevoir que lorsque l'on diminue l'erreur souhaitée le nombre d'itération évolue de manière **linéaire** car le fait de diminuer l'erreur souhaitée peut être écrit de façon linéaire tel que $e = e * 10^{-1}$.

Erreur relative avant :



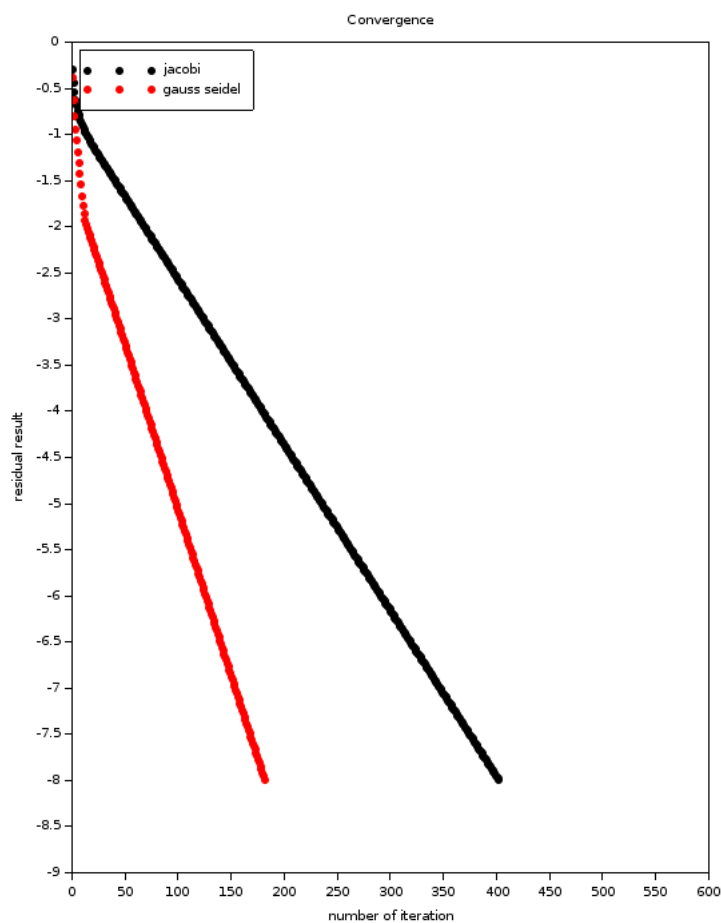
Ici ce n'a pas vraiment grand intérêt de calculer l'erreur relative car justement on varie l'erreur souhaité et donc on y arrive à un moment. Plus on baisse l'erreur souhaité plus l'erreur relative est faible.

6.4.3 Conclusion

Ces 2 graphiques confirment que la méthode de Gauss Seidel converge plus rapidement que celle de Jacobi en termes de nombre d'itération de la méthode. Mais cela ne veut pas dire qu'elle est plus rapide.

6.5 Question 5

Nous prenons un $n = 10$ et une erreur souhaité de $e = 10^{-8}$ pour le problème de poisson 1D et nous allons étudier sa convergence :



On voit que pour les 2 méthodes il y a convergence à 10^{-8} pour notre problème. Et que le nombre d'itération de Jacobi est 2 fois plus élevé que celui de Gauss Seidel.

7 Exercice 7

7.1 Question 1

Soit

$$x^{k+1} = x^k + \alpha(b - Ax^k)$$

avec $\alpha \in \mathbb{R}^{*+}$, $A \in \mathbb{R}^{n \cdot n}$, $b, x^k \in \mathbb{R}^n$, $k \in \mathbb{N}$

$$\begin{aligned} x^{k+1} &= x^k + \alpha b - \alpha A x^k \\ &= (I - \alpha A)x^k + \alpha b \\ &= G_\alpha x^k + \alpha b \end{aligned}$$

On pose $G_\alpha = I - \alpha A$

G_α est inversible et l'équation ci-dessus converge si $\rho(G_\alpha) < 1$

7.2 Question 2

Encadrement pour les valeurs propres (vp) de G_α qu'on note μ_i .

Soit les vp réelles de A, noté λ_i telles que :

$$\lambda_{min} \leq \lambda_i \leq \lambda_{max}$$

Alors les vp de $G_\alpha = (I - \alpha A)$ sont telles que

$$1 - \alpha \lambda_{max} \leq \mu_i \leq 1 - \alpha \lambda_{min}$$

Remarques : Comment en ai-t-on arrivé à cette conclusion ?

$$\text{Soit } P^{-1}AP = D_A = \begin{bmatrix} \lambda_1 & \dots & 0 \\ \dots & \lambda_i & \dots \\ 0 & \dots & \lambda_n \end{bmatrix}$$

$$\begin{aligned} P^{-1}G_\alpha P &= I - \alpha D_A \\ &= D_{G_\alpha} \\ &= \begin{bmatrix} 1 - \alpha \lambda_1 & \dots & 0 \\ \dots & 1 - \alpha \lambda_i & \dots \\ 0 & \dots & 1 - \alpha \lambda_n \end{bmatrix} \end{aligned}$$

Ce qui implique que $\mu_i = 1 - \alpha\lambda_i$

Et comme

$$\begin{aligned}\lambda_{min} &\leq \lambda_i \leq \lambda_{max} \\ -\alpha\lambda_{max} &\leq -\alpha\lambda_i \leq -\alpha\lambda_{min} \\ 1 - \alpha\lambda_{max} &\leq 1 - \alpha\lambda_i \leq 1 - \alpha\lambda_{min}\end{aligned}$$

7.3 Question 3

Quelle condition sur α pour que Richardson converge ?

Les différentes conditions proviennent du signe des valeurs propres de A .

- Soit $\lambda_{min} < 0$ et $\lambda_{max} > 0$
alors au moins une valeur propre de μ_i est > 1
car $1 - \alpha\lambda_{max} < 1$ et $1 - \alpha\lambda_{min} > 1$

Donc $\rho(G_\alpha) > 1 \Rightarrow$ Diverge

Remarque : cela marche uniquement pour les matrices définies positives

- $\lambda_{min} > 0$ toutes les vp de A sont positives

Remarque : cas des matrices SPD

Alors la convergence garantie par $\rho(G_\alpha) < 1$, est satisfaite pour :

$$1 - \alpha\lambda_{min} < 1 \Rightarrow \alpha > 0$$

$$1 - \alpha\lambda_{max} > -1 \Rightarrow \alpha < \frac{2}{\lambda_{max}}$$

Donc la méthode converge pour

$$0 < \alpha < \frac{2}{\lambda_{max}}$$

7.4 Question 4

Essayons maintenant de déterminer la valeur optimale de α que l'on notera α_{opt} .

Tout d'abord quelle est la caractéristique du α_{opt} . C'est la valeur de α qui minimise $\rho(G_\alpha)$.

$$\text{Donc } \rho(G_\alpha) = \max\{|1 - \alpha\lambda_{min}|, |1 - \alpha\lambda_{max}|\}$$

On trace les fonctions :

$$\begin{aligned} f_{min,max} : \mathbb{R} &\rightarrow \mathbb{R} \\ \alpha &\rightarrow |1 - \alpha\lambda_{min,max}| \end{aligned}$$

Intersection entre $-1 + \alpha\lambda_{max}$ et $1 - \alpha\lambda_{min}$

Donc,

$$\begin{aligned} -1 + \alpha_{opt}\lambda_{max} &= 1 - \alpha_{opt}\lambda_{min} \\ \alpha_{opt} &= \frac{2}{\lambda_{min} + \lambda_{max}} \end{aligned}$$

7.5 Question 5

Algorithm 4: Méthode de Richardson

Data: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $max_count, \epsilon, \alpha \in \mathbb{R}$

Result: $x \in \mathbb{R}^n$, $count, relres \in \mathbb{R}$, $resvec \in \mathbb{R}^n$

```
count = 0                                ▷ Initialisation du compteur
x = zeros(n)                             ▷ Init. vecteur résultat à 0
resvec = zeros(max_count)                 ▷ Init. vecteur relres à 0
normb = norm(b)                           ▷ Init. norme de b
r = b - A * x                             ▷ Initialisation du résidu
relres = norm(r) / normb                   ▷ Init. erreur relative
while relres >  $\epsilon$  and count < max_count do
    x = x +  $\alpha$  * r                       ▷ Calcul de richardson
    r = b - A * x                           ▷ Calcul du résidu
    relres = norm(r) / normb                 ▷ Calcul de l'erreur relative
    resvec(count) = relres                   ▷ Sauv. vecteur
    count = count + 1                       ▷ Incrémentation du compteur
end
```

De plus pour toute les complexités nous allons uniquement prendre en considération les informations de base des algorithmes qui permettent de calculé le résultat. C'est-à-dire que nous prendraon pas en comptes les vecteurs ainsi que les calculs qui sauvegarde les erreurs relative par exemple.

Complexité :

- en temps : indéterminé
- en temps pour une itération : $O(n^2 + 3n)$ (1 produit **matrice x vecteur**, 1 produit **scalaire x vecteur** et 2 addition/soustraction **vecteur x vecteur**)
- en espace : $O(n^2 + 3n)$ (1 matrice et 3 vecteur)

Remarques :

Nous pouvons voir que la méthode de richardson se rapproche de la méthode de Jacobi car pour les 2 méthodes M^{-1} est un scalaire, et étant donné que pour la méthode de **Richardson** nous avons calculé l'alpha optimal, la méthode de **Jacobi** est en fait un cas particulié de **Richardson** et cela

ne sert à rien de l'implémenter. Autant implémenter **Richardson** directement.

Question : Comment déterminer les valeurs propres de la matrice A ?

Pour cela il y a plusieurs options :

- calculer les vp à partir du déterminant mais cela est très coûteux
- calculer les vp à l'aide de fonction d'approximation (souvent utilisé) (approximation des vp ou des bornes)
- calculer les vp analytiquement lorsque l'on peut

8 Annexe

Dépôt github : https://github.com/Sholde/CN/tree/master/partie_2/poisson