Projet de Calcul Numérique : TP Poisson 1D

BAUMANN Pierre - M1 CHPS

0 : Introduction

Ce TP a pour but de résoudre le problème de l'équation de la chaleur stationnaire par méthodes directes et itératives, et de comparer les performances, précision et complexité des différentes méthodes.
Nous faisons cela en C en utilisant les bibliothèques BLAS et LAPACK.

Dans ce rapport nous allons dans un premier temps étudier l'équation de la chaleur, nous allons la transformer afin de pouvoir la traiter numériquement.

Ensuite, nous discuterons du format de stockage bande, et des matrices tri-diagonales.
Nous en profiterons pour nous nous familiariser avec les fonctions de BLAS et LAPACK.

Enfin, nous allons étudier les méthodes directes et itératives pour résoudre le problème de Poisson 1D.
Avec les méthodes directes, nous utiliserons des fonctions de résolution de systèmes linéaires comme la factorisation LU, descente et remontée, et avec les méthodes itératives, nous utiliserons des méthodes de résolution de systèmes linéaires itératives comme Richardson, Jacobi et Gauss-Seidel.
Nous comparerons les performances, précision et complexité des différentes méthodes.

Enfin, nous explorerons d'autres formats de stockage de matrices creuses, comme le format CSR et CSC, et nous implémenterons les méthodes directes et itératives avec ces formats.

Pour finir, nos repasserons brièvement sur les résultats obtenus, les avantages et inconvénients des différentes méthodes.

1 : Travail préliminaire : Étude de l'équation de la chaleur

Nous travaillons avec l'équation de la chaleur en 1D, dans un milieu immobile, linéaire, et homogène, avec des termes sources et isotropes.

L'équation de la chaleur peut se poser sous la forme d'un système d'équations différentielles partielles, avec des conditions aux bords, ici les conditions de Dirichlet.

$$\begin{cases} -k\frac{\delta^2 T}{\delta x^2} = g, x \in]0,1[\\ T(0) = T_0 \\ T(1) = T_1 \end{cases}$$

avec k la conductivité thermique, T la température, g un terme source, T_0 et T_1 les températures aux bords.

Dans ce projet, nous allons supposer qu'il n'y a pas de terme source ($g = 0$).

L'équation est posée dans un milieu continu, et donc pour pouvoir la résoudre numériquement, nous devons la discrétiser.
Nous allons discrétiser l'espace de façon uniforme en $n + 2$ points.

```
x_0     x_1     x_2     ...     x_n     x_{n+1}
 |       |       |               |         |
T_0 -+- T_1 -+- T_2 -+- T_i -+- T_n -+- T_{n+1}
```

En chaque point x_i, nous pouvons donc réécrire l'équation de la chaleur sous forme discrète:

$$-k(\frac{\delta^2 T}{\delta x^2})_i = 0$$

Nous obtenons donc le système linéaire suivant:

$$\begin{cases} T(0) = T_0 \\ -k(\frac{\delta^2 T}{\delta x_i^2}) = 0, \forall i \in [1, n] \\ T(1) = T_1 \end{cases}$$

La chaleur se propage donc de manière linéaire entre les points, et nous connaissons une solution analytique pour ce problème, qui est une fonction linéaire:

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

Cette solution analytique nous permettra de comparer et de valider nos résultats numériques.

Nous allons réécrire ce système $A * u = b$, avec A une matrice à déterminer, u le vecteur solution représentant les $T(x)$, et b le vecteur de droite contenant les conditions.

1.1 : Approximation de la dérivée seconde

La dérivée seconde reste dans le domaine continu, et donc pour pouvoir la calculer numériquement, nous devons l'approximer.
Pour cela, nous allons utiliser un développement de Taylor d'ordre 2, puis une méthode de différences finies centrées.

Le développement de Taylor d'ordre 2 de la fonction T en x_i est:

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

Ensuite, en sommant les deux équations, nous obtenons:

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

Et donc, en isolant les termes en $T(x)$ d'un côté, et les termes en $\frac{d^2T}{dx^2}$ de l'autre, nous obtenons:

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

Ensuite, on va multiplier par -1 pour faire réapparaître notre terme g.

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

Et donc, en isolant g_i, nous obtenons:

$$(\frac{d^2T}{dx^2})_i = g_i = \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2} + o(h^2)$$

Nous avons donc une approximation de la dérivée seconde de T en x_i en fonction de T en x_i, x_{i-1} et x_{i+1}.

1.2 : Système linéaire

Le système linéaire que nous devons résoudre est donc:

$$\begin{cases} u_0 = T_0 \\ -u_0 + 2u_1 - u_2 = -h^2g_1 \\ -u_1 + 2u_2 - u_3 = -h^2g_2 \\ ... \\ -u_{n-1} + 2u_n - u_{n+1} = -h^2g_n \\ u_{n+1} = T_1 \end{cases}$$

En écriture plus compacte:

$$\begin{cases} u_0 = T_0 \\ -u_{i-1} + 2u_i - u_{i+1} = -h^2g_i, \forall i \in [[1, n]] \\ u_{n+1} = T_1 \end{cases}$$

De plus, comme nous avons posé $g = 0$, nous avons:

$$\begin{cases} u_0 = T_0 \\ -u_{i-1} + 2u_i - u_{i+1} = 0, \forall i \in [[1, n]] \\ u_{n+1} = T_1 \end{cases}$$

Sous la forme d'un système matriciel $A * u = b$, nous avons:

$$\begin{bmatrix} 2 & -1 & 0 & ... & 0 & 0 & 0 \\ -1 & 2 & -1 & ... & 0 & 0 & 0 \\ 0 & -1 & 2 & ... & 0 & 0 & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 2 & -1 & 0 \\ 0 & 0 & 0 & ... & -1 & 2 & -1 \\ 0 & 0 & 0 & ... & 0 & -1 & 2 \end{bmatrix} * \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ ... \\ u_{n-1} \\ u_n \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} T_0 \\ 0 \\ 0 \\ ... \\ 0 \\ 0 \\ T_1 \end{bmatrix}$$

La matrice A est appelée matrice de Poisson 1D, et est une matrice tri-diagonale.
La diagonale principale est composée de 2, les diagonales au dessus et en dessous de la diagonale principale sont composées de -1, et les autres éléments sont nuls.

Analyse d'erreur de la matrice obtenue

Déjà, nous avons une perte de précision due à la discrétisation de l'espace, qui est de l'ordre de $O(h^2)$.
De plus, nous avons une erreur d'approximation de la dérivée seconde, qui est de l'ordre de $O(h^2)$.

Nous pouvons aussi calculer le préconditionnement et le rayon spectral de la matrice A, pour avoir une idée de la qualité numérique de la matrice.

$$\lambda_{max}(A) = 4 * sin^2(\frac{\pi}{2} * \frac{n}{n+1}) \approx 4$$
$$\lambda_{min}(A) = 4 * sin^2(\frac{\pi}{2 * (n+1)}) \approx 0$$
$$\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \approx \frac{4}{h^2}$$
$$\rho(A) = \frac{\lambda_{max}(A) - \lambda_{min}(A)}{\lambda_{max}(A) + \lambda_{min}(A)} \approx 1 - \frac{\pi^2 * h^2}{2}$$

On peut voir que choisir un h petit permet améliore le rayon spectral, ainsi que l'approximation de notre problème continue, mais augmente le conditionnement de la matrice, ce qui peut entraîner des erreurs numériques.

Il est donc important de choisir un h qui permet d'avoir une bonne approximation de notre problème continue, tout en gardant un conditionnement de la matrice acceptable, et de trouver un compromis entre les deux.

2 : Méthodes directes et stockage bande

2.3 : Référence et utilisation de BLAS/LAPACK

Prise en main de BLAS et LAPACK

2.3.1 : Déclaration et allocation de matrices en C pour BLAS et LAPACK

Pour utiliser les fonctions de BLAS et LAPACK, nous devons stocker nos matrices en 1D, et donc nous devons allouer un espace mémoire contigu de taille $n * m$ pour une matrice $n * m$.

Exemple pour une matrice 3x2:

```c
double a[3][2] = {{1, 2}, {3, 4}, {5, 6}};  // Statique (et contigu)
double* b = malloc(3*2 * sizeof(double));   // Dynamique
```

2.3.2 : Différence entre LAPACK_ROW_MAJOR et LAPACK_COL_MAJOR

LAPACK_COL_MAJOR est une constante à passer en argument à certaines fonctions Lapack pour indiquer que nos matrices sont stockées avec une priorité colonne.
C'est à dire que tous les éléments d'une colonne sont stockés de manière contiguë en mémoire, et que les colonnes sont stockées les unes à la suite des autres.

Comme les matrices sont stockées en 1D dans le code, la matrice 2D
`m[2][2] = {1,2,3,4}` sera interprétée comme

$$m = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

en LAPACK_COL_MAJOR et

$$m = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

en LAPACK_ROW_MAJOR.

2.3.3 : Dimension principale

La dimension principale, notée ld, correspond à la taille de la matrice dans sa dimension désignée par LAPACK_COL_MAJOR ou LAPACK_ROW_MAJOR.

Cela sera donc le nombre de lignes de la matrice en LAPACK_COL_MAJOR et au nombre de colonnes en LAPACK_ROW_MAJOR.
Nous pouvons aussi la voir comme le nombres d'éléments entre deux éléments consécutifs de la même ligne (en LAPACK_COL_MAJOR) ou de la même colonne (en LAPACK_ROW_MAJOR).

Manipulation de matrices bandes

Le stockage bande est un format de stockage pour des matrices particulières, qui n'ont des éléments non nuls que dans une bande autour de la diagonale principale.
Par exemple, une matrice tri-diagonale ou penta-diagonale.

$$T = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 1 & 2 & 0 & 0 & 0 \\ 0 & 3 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 1 & 2 & 0 \\ 0 & 0 & 0 & 3 & 1 & 2 \\ 0 & 0 & 0 & 0 & 3 & 1 \end{bmatrix} P = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 1 & 2 & 3 & 0 & 0 \\ 5 & 4 & 1 & 2 & 3 & 0 \\ 0 & 5 & 4 & 1 & 2 & 3 \\ 0 & 0 & 5 & 4 & 1 & 2 \\ 0 & 0 & 0 & 5 & 4 & 1 \end{bmatrix}$$

Comme nous pouvons le voir sur ces matrices, le nombre d'éléments non nuls devient très vite important en fonction de la taille de la matrice.
Pour une matrice de taille n avec b bandes, la proportion d'éléments non nuls est de
$\frac{b*n}{n^2} = \frac{b}{n}$
, et donc pour une matrice de taille $n = 1000$ avec 5 bandes, nous avons une proportion d'éléments non nuls de 0.005, et donc le stockage bande est très efficace pour les matrices creuses.

Les bandes n'ont pas besoin d'être symétriques, et on peut généraliser.
Pour une matrice A avec kl bandes en dessous de la diagonale principale et ku bandes au dessus, on peut stocker la matrice en stockage bande avec $kl + ku + 1$ bandes, et donc une matrice A de taille $n * m$ avec kl bandes en dessous et ku bandes au dessus, sera stockée en une matrice B de taille $n * (kl + ku + 1)$.

Exemple:

$$kl = 2, ku = 1$$
$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & 0 \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{bmatrix}$$

sera stockée en

$$B = \begin{bmatrix} 0 & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} & a_{67} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} & a_{77} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & a_{76} & 0 \\ a_{31} & a_{42} & a_{53} & a_{64} & a_{75} & 0 & 0 \end{bmatrix}$$

Ici, nous pouvons voir que nous avons gagné un gain de place considérable.
De plus, cet exemple reste assez petit, mais vous pouvez imaginez avec des matrices bien plus grandes.
En complexité spatiale, nous avons $O(n * (kl + ku))$ au lieu de $O(n^2)$ pour une matrice pleine, qui est tout le temps inférieur car $kl + ku < n$, et très souvent $kl + ku << n$.

Nous allons maintenant utiliser les fonctions de BLAS et LAPACK pour manipuler des matrices bandes.

2.3.4 : DGBMV

DGBMV = Double General Band Matrix Vector multiplication.

Avec A une matrice bande, x et y des vecteurs, alpha et beta des scalaires, elle calcule
$\alpha * A * x + \beta * y$,
ou
$\alpha * A^T * x + \beta * y$
selon si l'on demande de transposer A ou non.

D'abord elle commence par calculer `y = beta*y` séquentiellement, en séparant des cas optimisés.

- Si beta = 0, alors y = 0
- Si beta = 1, alors on ne fait rien
- Sinon, on multiplie chaque élément de y par beta

Nous pouvons aussi spécifier de multiplier seulement les n éléments de y, avec l'argument incy, et dans ce cas on a aussi 2 cas optimisés pour incy = 1 et incy != 1.
Cette optimisation semble inutile, car en C cela reviendrai à différencier les cas
`for (int i = 0; i < n; i++) {}`
`for (int i = 0; i < n; i += incy) {}` .
Mais cela est sûrement dû au fait que le code a été écrit il y a longtemps, et que les compilateurs n'étaient pas aussi optimisés qu'aujourd'hui.

Ensuite, elle calcule le produit matrice-vecteur colonne par colonne, en calculant l'index de début de la colonne dans la matrice A, et en multipliant chaque élément de la colonne par le vecteur x, et en ajoutant le résultat à y avec l'opération triadique
`y(i) = y(i) + x(i)*A(i,j)` .

La fonction est optimisée pour les matrices diagonales, et la boucle la plus interne ne fait que ku + kl + 1 itérations, où ku et kl sont les nombres de diagonales au dessus et en dessous de la diagonale principale.

Cette fonction sépare les cas pour les incréments de x et y, et pour la transposition de A qu'elle fait à la volée en inversant les indices d'accès à A pour ne pas avoir à stocker A^T en mémoire.

En terme de complexité temporelle, la fonction doit:

- Calculer $y = \beta * y$ en $O(n)$
- Calculer $\alpha * A * x$, qui serait en $O(n^2)$ pour les matrices pleines, mais ici est en $O(n * (kl + 1 + ku))$ pour les matrices bandes.
- Finalement, additionner les résultats à y en $O(n)$.

Donc la complexité temporelle totale est en $O(n * (kl + 1 + ku))$.

En complexité spatiale, la fonction n'a pas besoin de mémoire supplémentaire à part les arguments, et donc est en $O(1)$, dit en place.

Quant à la qualité numérique, cette fonction est très précise, car elle ne fait pas énormément d'opérations, et donc les erreurs d'arrondis sont minimisées.

2.3.5 : DGBTRF

DGBTRF = Double General Band matrix TRiangular Factorisation.

Cette fonction effectue la factorisation LU d'une matrice bande A, en stockant les facteurs L et U dans la matrice A.
Elle utilise la méthode de Gauss avec pivot partiel, et stocke les pivots dans le vecteur ipiv.

Elle utilise le format de stockage de matrice bande, mais nécessite un peu plus de place pour stocker L et U, et donc la matrice A doit être carrée, et les dimensions de A doivent être égales.

Elle effectue la factorisation LU en 2 étapes:

- Factorisation LU de la matrice bande A, en stockant les facteurs L et U dans la matrice A.
- Stockage des pivots dans le vecteur ipiv.

En terme de complexité temporelle, la factorisation LU d'une matrice bande est en $O(n * (kl + ku)^2)$, car chaque colonne de la matrice bande ne possède que $kl + ku + 1$ éléments non nuls, et donc chaque étape de la factorisation LU ne nécessite que $(kl + ku + 1)^2$ opérations au maximum au lieu de n^2 opérations pour une matrice pleine.

En complexité spatiale, nous avons besoin d'avoir des superdiagonales libres pour stocker des résultats intermédiaires, qui est l'intérêt du kv, donc nous avons besoin de n éléments en plus par rapport à juste la matrice tri-diagonale.

En terme de qualité numérique, la factorisation LU est très précise, car elle utilise la méthode de Gauss avec pivot partiel, qui est une méthode stable numériquement.
Elle permet d'éviter les erreurs d'arrondis lorsque nous rencontrons des éléments très petits lors de la résolution du système linéaire.
L et U sont les facteurs de la matrice A plus une matrice de perturbation E.
avec $|E| < \epsilon * f(kl + ku + 1) * |L| * |U|$, où ϵ est la précision machine, et $f(kl + ku + 1)$ est une fonction linéaire en $kl + ku + 1$.

2.3.6 : DGBTRS

DGBTRS = Double General Band matrix TRiangular Solve.

Elle résout un système linéaire $A * x = b$ avec une matrice bande
La matrice A doit avoir été factorisée avec dgbtrf, et est donc sa décomposition LU.
Elle résout le système linéaire en 2 étapes:

- Résolution du système linéaire $L * y = b$ avec $y = U * x$.
- Résolution du système linéaire $U * x = y$ avec U la matrice triangulaire supérieure de la décomposition LU de A.

Elle utilise la méthode de substitution avant et arrière pour résoudre le système linéaire, avec les méthodes de remontée et de descente, et nécessite le vecteur ipiv de la factorisation LU de A.

En terme de complexité temporelle, la résolution du système linéaire avec une matrice bande est en $O(n * (kl + ku))$, car chaque colonne de la matrice bande ne possède que $kl + ku + 1$ éléments non nuls, et donc chaque étape de la résolution du système ne nécessite que $kl + ku + 1$ opérations au maximum au lieu de n opérations pour une matrice pleine.

En complexité spatiale, la fonction n'a pas besoin de mémoire supplémentaire à part les arguments, et donc est en $O(1)$, dit en place.

La précision numérique est de l'ordre de $O(\kappa(A) * \epsilon)$, où $\kappa(A)$ est le conditionnement de la matrice A, et ϵ est la précision machine, car la méthode de substitution avant et arrière est stable numériquement.

2.3.7 : DGBSV

DGBSV = Double General Band matrix Solve.

Cette fonction résout un système linéaire $Ax = b$ avec une matrice bande.
Elle effectue la factorisation LU de la matrice bande et résout le système linéaire en une seule fonction, en appelant dgbtrf et dgbtrs à la suite.

La complexité et la précision de cette fonction sont donc le cumul des complexités et précisions de dgbtrf et dgbtrs.

2.3.8 : Calcul de la norme du résidu avec des appels BLAS

Déjà, rappelons les formules des différentes erreurs, avec x le vecteur solution exact, $\hat{x}$ le vecteur solution approximée, b le vecteur de droite, A la matrice du système linéaire, et r le résidu, on a:

Formule du résidu:

$$r = b - A\hat{x}$$

Erreur relative arrière:

$$relres = \frac{||b - A\hat{x}||}{||A||||\hat{x}||}$$

Erreur relative avant:

$$relres = \frac{||r||}{||b||} = \frac{||b - A\hat{x}||}{||b||}$$

Dans notre cas, on connait le x solution exact, donc on peut utiliser la formule suivante:

$$relres = \frac{||x - \hat{x}||}{||x||}$$

Dans ce TP on utilise l'erreur avant comme résidu, et on connait notre x exact, donc on va utiliser

$$r = \frac{||x - \hat{x}||}{||x||}$$

Pseudo-code:

```
// xex est le vecteur exact
// xappr est le vecteur approximé

// Numérateur
a = ddot(&n, xex, 1, xex, 1) // Somme des carrés des éléments de xex
a = sqrt(a) // Norme de xex

// Dénominateur
b = daxpy(&n, -1, xex, 1, xappr, 1) // xex contient xex - xappr
b = ddot(&n, b, 1, b, 1) // Somme des carrés des éléments de xex - xappr
b = sqrt(b) // Norme de xex - xappr

// Résultat
r = b/a
```

Les signatures des fonctions cblas sont identiques, donc le code C sera très similaire.

2.4: Stockage GB et appel à DGBMV

2.4.1 : Stockage GB en priorité colonne pour la matrice de Poisson 1D

Rappelons la forme de la matrice de Poisson 1D:

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \\ -1 & 2 & -1 & 0 & ... \\ 0 & -1 & 2 & -1 & \\ 0 & 0 & -1 & 2 & \\ & & \vdots & & \end{bmatrix}$$

La matrice de Poisson 1D possède 1 bande au dessus et 1 bande en dessous de la diagonale principale, et donc on peut la stocker en format General Band Storage avec 3 bandes, et donc on a besoin de $3 * n$ éléments pour stocker la matrice.

En ajoutant kv superdiagonales libres, nous avons besoin de $n * (kv + 3)$ éléments pour stocker la matrice.
Voici comment elle est stockée en general band :

$$\begin{bmatrix} 0 & 0 & 0 & 0 & ... & 0 \\ \vdots & kv & lignes & de & zéros & \vdots \\ 0 & 0 & 0 & 0 & ... & 0 \\ 0 & -1 & -1 & ... & -1 & -1 \\ 2 & 2 & 2 & ... & 2 & 2 \\ -1 & -1 & -1 & ... & -1 & 0 \end{bmatrix}$$

Avec une priorité colonne, c'est à dire que les éléments de la colonne sont contigus en mémoire, nous avons cette représentation en mémoire:

```
A = {/*kv zéros*/, 0, 2, -1,
        /*kv zéros*/, -1, 2, -1,
        /*kv zéros*/, -1, 2, -1,
        ...,
        /*kv zéros*/, -1, 2, 0};
```

2.4.3 : Validation de cette représentation

Pour valider notre stockage, nous allons faire appel à des fonctions BLAS travaillant avec des matrices bandes, calculer des produits matrice-vecteur simples, et vérifier que les résultats sont corrects.

Par exemple, nous pouvons utiliser la multiplication avec des vecteurs unitaires pour obtenir les colonnes de la matrice, et vérifier que les résultats sont corrects.

```
// On suppose A et x déjà initialisés, et y déjà alloué
double y[n];
int incx = 1;
int incy = 1;
double alpha = 1;
double beta = 0;
cblas_dgbmv(CblasColMajor, CblasNoTrans, n, n, kv, kv, alpha, A, n, x, incx, beta, y, incy);
// y contient maintenant le résultat de A*x
```

Et normalement on doit obtenir le vecteur colonne correspondant à la colonne de la matrice stockée.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

Nous pouvons aussi multiplier A par le vecteur $(1, 1, ...1.., 1)$ et vérifier que le résultat est $(1, 0, 0, ..0.., 0, 1)$, car cela revient à sommer les colonnes de A, et $-1 + 2 - 1 = 0$, à part pour les premières et dernières colonnes qui ont un -1 en moins, et donc $2 - 1 = 1$ et $-1 + 2 = 1$.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

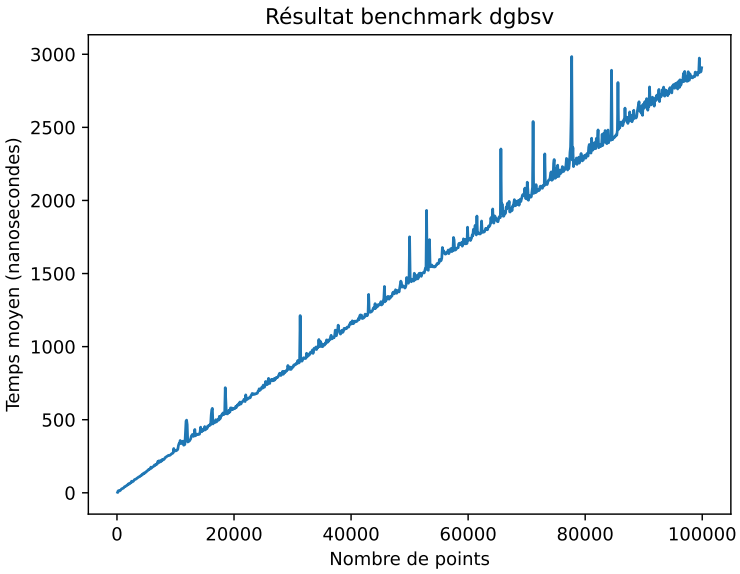
Nous pouvons aussi la multiplier par la matrice identité pour vérifier que le résultat est bien la même matrice.

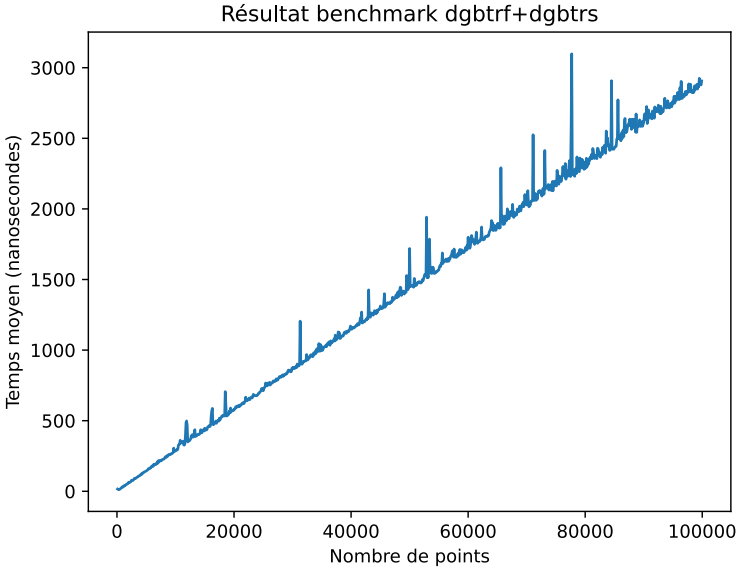
$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

En termes de qualité numérique, les matrices bandes peuvent avoir une précision légèrement meilleure que les matrices pleines, car nous avons moins d'opérations à effectuer pour la plupart des opérations, et aussi nous évitons de stocker des zéros, qui peuvent parfois se retrouver avec des valeurs très petites mais non nulles à cause d'erreur d'arrondis précédants, et donc être source d'erreurs numériques.

2.5: DGBTRF, DGBTRS, DGBSV

Après avoir benchmarké les fonctions DGBTRF+DGBTRS et DGBSV, nous avons obtenu les résultats suivants:





On peut voir que les deux fonctions ont des performances similaires.
Et malgré le bruit on peut remarquer qu'elles ont toutes les deux une complexité linéaire en fonction du nombre de points.

Ce qui est attendu car la matrice est tri-diagonale, et donc la factorisation LU est en O(n) et la résolution du système est en O(n), car chaque colonne de la matrice ne possède que 3 éléments non nuls, et donc chaque étape de la factorisation LU et de la résolution du système ne nécessite que 3 opérations au maximum au lieu de n opérations pour une matrice pleine.

2.6: LU pour les matrices tri-diagonales

Pour la décomposition LU, l'algorithme de Gauss avec pivot partiel peut être optimisé pour les matrices tri-diagonales.
Notamment lors de la recherche du pivot, car nous savons que le pivot optimal est toujours sur
un des voisins de la ligne sur laquelle nous travaillons, car nous savons qu'en dessous et au dessus il n'y aura que des zéros.

2.6.1 : Validation

On peut valider notre implémentation de la factorisation LU en multipliant la matrice L et U, que nous devons extraire car elles sont fusionées dans la matrice A après l'appel, et en vérifiant que le résultat est bien la matrice originale, car par définition de la factorisation LU, $A = LU$.
En plus, on peut utiliser des tests unitaires avec d'autres méthodes de factorisation LU sur les matrices générales pour vérifier que notre implémentation est correcte.
Par exemple,

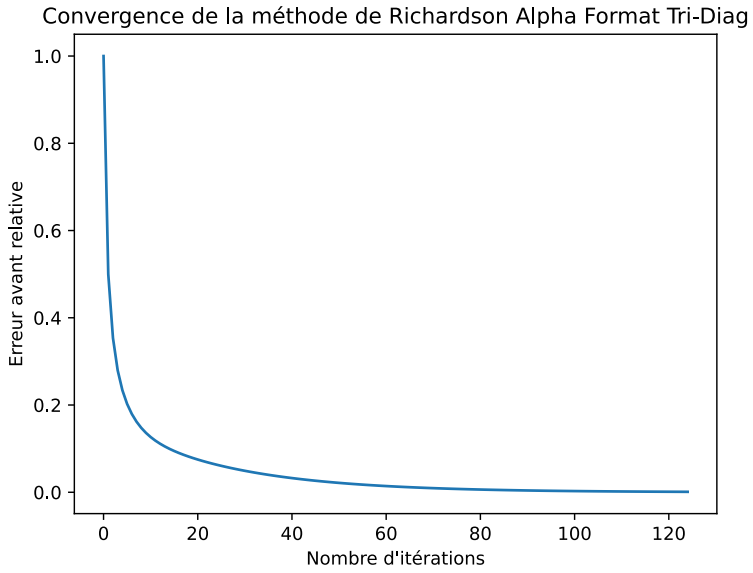
$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

doit être factorisée en

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0 & -0.66... & 1 \end{bmatrix} U = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1.5 & -1 \\ 0 & 0 & 1.33 \end{bmatrix}$$

3 Méthode de resolutions itératives

3.7: Implémentation C - Richardson



Résultats :

Index	X found	X exact
0	6.337976	6.363636
1	7.679237	7.727273
2	9.022074	9.090909
3	10.373724	10.454545
4	11.728027	11.818182
5	13.093872	13.181818
6	14.462603	14.545455
7	15.841942	15.909091
8	17.223485	17.272727
9	18.611332	18.636364

Convergence à 10^{-3} en 125 itérations.

Erreur par rapport à la solution exacte: 0.005673

3.8 Implémentation C - Jacobi

On a A =

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

On la sépare en D, L et U:

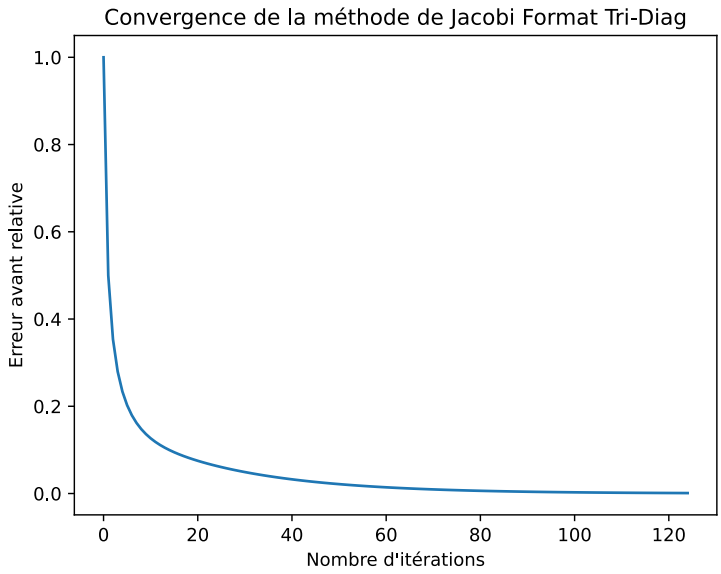
$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} L = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} U = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

La matrice d'itération de Jacobi est:

$$M = D^{-1} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}$$

On peut voir que la matrice d'itération de Jacobi est une matrice diagonale, et donc stockable en format General Band.

On peut aussi remarquer que cette matrice revient au même résultat que de faire richardson avec un $\alpha = 0.5$, car $M = D^{-1} = 0.5 * I$.



Résultats :

Index	X found	X exact
0	6.337976	6.363636
1	7.679237	7.727273
2	9.022074	9.090909
3	10.373724	10.454545
4	11.728027	11.818182
5	13.093872	13.181818
6	14.462603	14.545455
7	15.841942	15.909091
8	17.223485	17.272727
9	18.611332	18.636364

Convergence à 10^{-3} en 125 itérations.

Erreur par rapport à la solution exacte: 0.005673

On a exactement la même convergence et résultats que pour Richardson avec un $\alpha = 0.5$, ce qui confirme que ces deux méthodes sont équivalentes.

3.9: Implémentation C - Gauss-Seidel

On a

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

On la sépare en D, L et U:

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} L = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} U = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

La matrice d'itération de Gauss-Seidel est:

$$M = D - E = D + L = \begin{bmatrix} 2 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

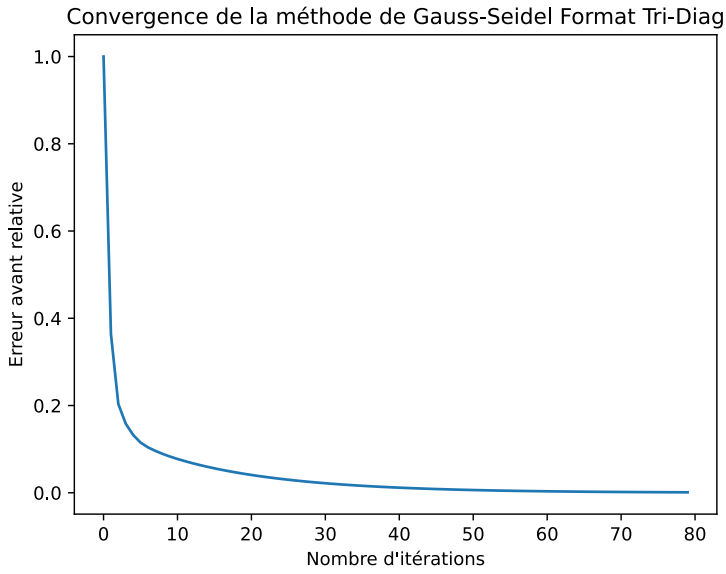
On inverse M pour obtenir la matrice d'itération de Gauss-Seidel:

$$M^{-1} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0.25 & 0.5 & 0 & 0 \\ 0.125 & 0.25 & 0.5 & 0 \\ 0.0625 & 0.125 & 0.25 & 0.5 \end{bmatrix}$$

On peut voir un pattern dans les coefficients de la matrice d'itération de Gauss-Seidel, en effet, chaque coefficient est la moitié du coefficient au dessus, et le premier coefficient est 0.5 à la diagonale principale.
Cela est cohérent avec la méthode de Gauss-Seidel, car on fait la moyenne des valeurs des voisins pour obtenir la valeur de la cellule courante, et à chaque itération on fait la moyenne des valeurs des voisins de la valeur précédente, et donc on divise par 2 à chaque itération.

Cependant, pour avoir une matrice tri-diagonale, nous allons devoir tronquer la matrice d'itération de Gauss-Seidel, car elle est triangulaire inférieure.
Nous nous retrouvons avec une matrice M^{-1} un peu moins précise, mais qui est tri-diagonale:

$$M^{-1} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0.25 & 0.5 & 0 & 0 \\ 0 & 0.25 & 0.5 & 0 \\ 0 & 0 & 0.25 & 0.5 \end{bmatrix}$$



Résultats:

Index	X found	X exact
0	6.336293	6.363636
1	7.674070	7.727273
2	9.015502	9.090909
3	10.362511	10.454545
4	11.716634	11.818182
5	13.078825	13.181818
6	14.449478	14.545455
7	15.828158	15.909091
8	17.214033	17.272727
9	18.605124	18.636364

Convergence à 10^{-3} en 80 itérations.

Erreur par rapport à la solution exacte: 0.005856

4: Autres formats de stockage

4.10: Stockage CSR et CSC

D'abord, rappelons ce que sont les formats CSR et CSC.
Le format CSR (Compressed Sparse Row) est un format de stockage de matrices creuses, qui consiste à stocker uniquement les éléments non nuls de la matrice.
On stocke 3 vecteurs:

- Un vecteur des valeurs non nulles de la matrice, dans l'ordre des lignes.
- Un vecteur des colonnes des valeurs non nulles de la matrice, dans l'ordre des lignes.
- Un vecteur des indices de début de ligne, qui contient le nombre d'éléments non nuls avant la ligne i, à lire par fenêtre de 2 éléments pour obtenir les indices de début et de fin des éléments non nuls de la ligne i.

Par exemple, prennons cette matrice compressée en format CSR:

$$values = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$columns = \begin{bmatrix} 1 & 0 & 2 & 1 \end{bmatrix}$$

$$rows = \begin{bmatrix} 0 & 1 & 3 & 4 \end{bmatrix}$$

On lit rows pour savoir que la première ligne contient les élements d'indices [0, 1[dans values et columns.
Il y a donc 1 élément non nul dans la première ligne, qui est values[0] = 1.
Et on lit columns[0] pour savoir que cet élément est à la colonne 1.

La deuxième ligne contient les éléments d'indices [1, 3[dans values et columns.
Il y a donc 2 éléments non nuls dans la deuxième ligne, qui sont values[1] = 2 et values[2] = 3.
Et on lit columns[1] et columns[2] pour savoir que ces éléments sont à la colonne 0 et 2.

On fait ça pour chaque ligne, et on peut reconstruire la matrice originale, qui est :

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 0 & 4 & 0 \end{bmatrix}$$

On peut d'ailleurs recréer la matrice originale depuis son format CSR avec un algorithme très simple:

```
A = zeros(N,N)
for (i = 0; i < row.size() - 1; i++)
  for (j = row[i]; j < row[i+1]; j++)
    A(i, col[j]) = val[j]
  end
end
```

Le format CSC (Compressed Sparse Column) est similaire au format CSR, mais on stocke les valeurs non nulles par colonne au lieu de par ligne, et on stocke les indices de début de colonne au lieu de début de ligne.

Nous n'allons donc pas détailler le format CSC, surtout car dans notre contexte, la matrice de Poisson 1D est symétrique, et donc le format CSR et CSC sont équivalents.

4.10.1 : Stockage CSR pour la matrice de Poisson 1D.

Pour la matrice de Poisson 1D, qui est de cette forme:

$$P = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

On commence par stocker les valeurs non nulles en continu, dans le sens de lecture classique (gauche puis droite, haut puis bas):

$$values = \begin{bmatrix} 2 & -1 & -1 & 2 & -1 & -1 & 2 & -1 & -1 & 2 \end{bmatrix}$$

Il suffit de construire un tableau de taille 3*n - 2 (la première et la dernière ligne ont 2 éléments non nuls, les autres en ont 3),

On rempli les 2 premiers éléments de la première ligne 2 et -1,

Puis on répète n-2 fois les 3 éléments -1, 2 et -1,

Et on finit par les 2 derniers éléments de la dernière ligne -1 et 2.

Ensuite, on stocke les colonnes des valeurs non nulles:

$$columns = \begin{bmatrix} 0 & 1 & 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 \end{bmatrix}$$

On construit un tableau de même taille, on stocke 0 et 1 pour la première ligne.

Et pour chaque ligne i avant la dernière, on stocke i-1, i et i+1 (car chaque ligne a 3 éléments non nuls consécutifs, tous décalés d'une colonne).

Pour la dernière ligne, on stocke n-2 et n-1 pour les deux derniers éléments.

Enfin, on stocke les indices de début de ligne:

$$rows = \begin{bmatrix} 0 & 2 & 5 & 8 & 10 \end{bmatrix}$$

On construit un tableau de taille n+1, et on stocke les indices [0, 2[pour la première ligne.

Pour chaque ligne i avant la dernière, on ajoute 3 à l'indice de fin de la ligne i-1 pour obtenir l'indice de fin de la ligne i, car chaque ligne a 3 éléments non nuls.

Et pour la dernière ligne, on ajoute 2 à l'indice de fin de la ligne n-2 pour obtenir l'indice de fin de la dernière ligne, car la dernière ligne a 2 éléments non nuls.

4.10.2 : Stockage CSC pour la matrice de Poisson 1D.

On repart de la matrice P de Poisson 1D de taille 4x4:

$$P = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

On stocke les valeurs non nulles en continu, dans le sens de lecture priorité colonne (haut puis bas, gauche puis droite):

$$values = \begin{bmatrix} 2 & -1 & -1 & 2 & -1 & -1 & 2 & -1 & -1 & 2 \end{bmatrix}$$

On stocke les lignes des valeurs non nulles:

$$rows = \begin{bmatrix} 0 & 1 & 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 \end{bmatrix}$$

Et on stocke les indices de début de colonne:

$$columns = \begin{bmatrix} 0 & 2 & 5 & 8 & 10 \end{bmatrix}$$

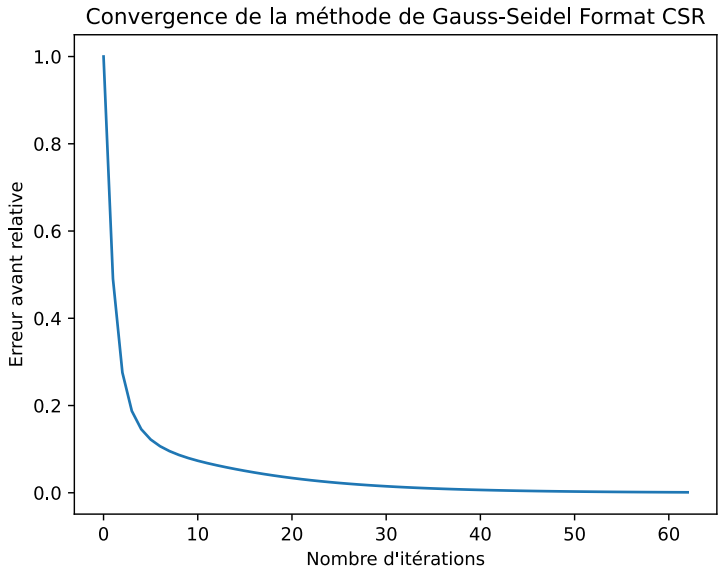
Les 3 membres de la matrice en format CSC sont les mêmes que pour le format CSR, mais la matrice est symétrique, donc cela reste cohérent.

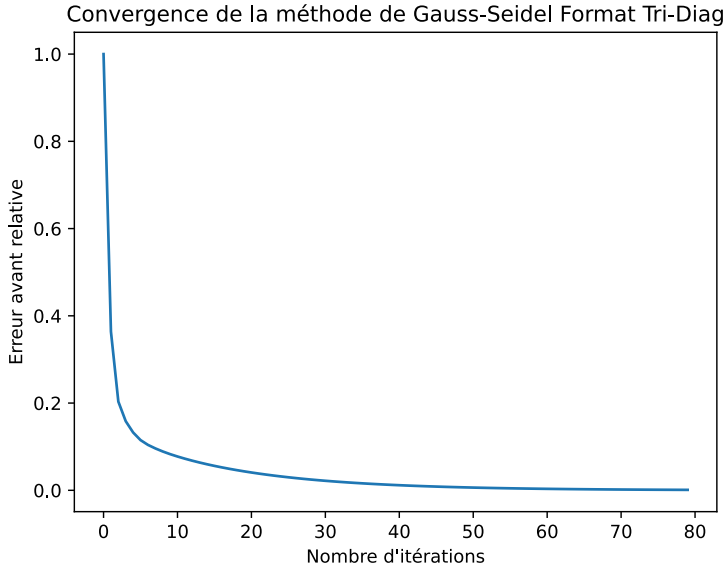
4.10.4 : Différents algorithmes pour ces formats.

Les algorithmes restant les mêmes, on peut juste copier les algorithmes des formats tridiagonaux en changeant les appels dgbsv par dcsrmv ou dcscmv.

On peut cependant noter quelques différences:

- Le format CSR, n'étant plus limité à une matrice tri-diagonale, converge plus rapidement que le format tridiagonal pour Gauss-Seidel, car nous n'avons plus à tronquer la matrice d'itération.
  Cela vient au prix de la complexité spatiale et temporelle plus importante, car nous devons stocker plus d'éléments, et faire plus d'opérations pour chaque itération.





Ici, la méthode avec le format tri-diagonal converge en 80 itérations, alors que la méthode avec le format CSR n'en prend que 63.

Résultats Gauss-Seidel (format CSR):

Index	X found	X exact
0	6.329861	6.363636
1	7.665084	7.727273
2	9.007498	9.090909
3	10.358217	10.454545
4	11.717607	11.818182
5	13.085318	13.181818
6	14.460364	14.545455
7	15.841259	15.909091
8	17.226168	17.272727
9	18.613084	18.636364

Convergence à 10^{-3} en 63 itérations.
Erreur par rapport à la solution exacte: 0.005673

Pour Richardson "alpha" et Jacobi, il n'y a aucune différence avec le format general band, car pour l'un on utilise un scalaire, et pour l'autre une matrice diagonale, et les 2 formats arrivent à les stocker sans perte d'information.
Les résultats de Richardson alpha sont les mêmes que pour Jacobi, car les deux méthodes sont équivalentes pour alpha = 0.5. Je ne vais donc pas les réécrire.

Résultats Jacobi (format CSR):

Index	X found	X exact
0	6.329861	6.363636
1	7.665084	7.727273
2	9.007498	9.090909
3	10.358217	10.454545
4	11.717607	11.818182
5	13.085318	13.181818
6	14.460364	14.545455
7	15.841259	15.909091
8	17.226168	17.272727
9	18.613084	18.636364

Convergence à 10^{-3} en 125 itérations.
Erreur par rapport à la solution exacte: 0.005673

Entre Gauss-Seidel et Jacobi, cette fois nous avons bien les mêmes résultats pour les deux méthodes, avec Jacobi qui converge en 2 fois plus d'itérations que Gauss-Seidel, ce qui est cohérent avec la théorie.

5 : Conclusion

5.1 : Différentes méthodes de résolution

Méthode	Complexité Spatiale	Complexité Temporelle	Précision numérique	Taux de Convergence	Parallélisable?
Richardson	O(3n)	O(3n/iter)	Tolérance choisie	(λmax - λmin)/(λmax + λmin)	Oui
Jacobi	O(4n)	O(6n/iter)	Tolérance choisie	ρ(cos(pi * h))	Oui
Gauss-Seidel	O(4n)	O(6n/iter)	Tolérance choisie	ρ(cos²(pi * h))	Moyennement
DGBSV (Factorisation LU)	O(3n)	O(9n)	10^{-15}	X	Non

Ici, nous pouvons voir que parmi les méthodes utilisées, la méthode directe est la plus précise, et souvent la plus rapide, de plus, comme nous faisons moins de calculs, il y a moins de risque d'erreurs d'arrondis.

Parmi les méthodes itératives, Gauss-Seidel prend le moins de temps, notamment car il converge plus rapidement que les autres méthodes. Cependant, il n'est pas parallélisable, contrairement à Richardson et Jacobi, qui peuvent être parallélisés un peu plus facilement, ce qui peut être un avantage pour des matrices plus grandes.

De plus, la matrice d'itération de Jacobi est plus simple à calculer, car elle est diagonale, et donc trouver l'inverse revient à mettre l'inverse des éléments de la diagonale, ce qui est plus simple que de trouver l'inverse d'une matrice tridiagonale.

5.2 : Différents formats de stockage

Format	Contraintes sur les 0?	Place mémoire	Temps Matrice * Vecteur	Résolution directe A*x = b
Dense	Non	O(n * m)	O(n * m)	$O(N^3)$
General Band	Oui	O(n * (kl + ku))	O(n * (kl + ku))	O(3N)
CSR	Non	O(n + nombre éléments =/= 0)	O(nombre éléments =/= 0)	$O((\text{nombre éléments} =/= 0)^3)$
CSC	Non	O(m + nombre éléments =/= 0)	O(nombre éléments =/= 0)	$O((\text{nombre éléments} =/= 0)^3)$

Nous pouvons voir que les formats compressés sont plus efficaces en terme de place mémoire, et aussi en temps de calculs car ils évitent de faire des calculs sur des zéros, et donc réduisent le nombre d'opérations à effectuer.
Entre le format CSR et CSC, il n'y a pas aucune différence.
Par contre, le format General Band est plus efficace que le format CSR/CSC pour les matrices tri-diagonales, car nous n'avons pas à stocker les indices des colonnes, mais il est cepandant plus limité, contrairement au format CSR/CSC qui ne perd pas d'information sur la matrice.

Pour l'algorithme de Richardson avec alpha et la résolution directe, le format General Band est plus efficace dans notre cas, car nous avons une matrice tri-diagonale remplie.
Pour la méthode de Gauss-Seidel, le format CSR/CSC est plus efficace, car nous n'avons pas à tronquer la matrice d'itération, et donc nous avons une convergence plus rapide.

5.3 : Conclusion

Nous avons vu dans ce TP plusieurs méthodes de résolution de systèmes linéaires, et plusieurs formats de stockage de matrices, et nous avons pu comparer les performances de ces méthodes.

Pour les méthodes de résolution, nous avons vu que les méthodes itératives sont souvent plus rapides et plus précises que les méthodes directes.

Cependant, la résolution analytique reste la meilleure, ce qui est normal.
Nous avons quand même pu explorer les différentes méthodes de résolution, et comparer les avantages et inconvénients de chacune.