Calcul Numérique
Méthodes d'Algèbre Linéaire
appliquées au problème de Poisson 1D

Matéo Pasquier

Janvier 2024

Contents

1 Introduction et généralités

L'utilisation d'ordinateurs pour résoudre des équations n'est pas nouvelle, le principe existe depuis les débuts de l'informatique et cette pratique a même une utilité grandissante avec les années, les problèmes que nous souhaitons résoudre devenants de plus en plus complexes. En effet, l'atout principal des machines est qu'elles peuvent effectuer des calculs à des vitesses inatteignables par des humains.
Cela est bien sûr utile dans de nombreuses situations. Nous pouvons vouloir résoudre des équations dont nous ne possédons pas de solution en utilisant par exemple des méthodes de discrétisation (Euler...). Bien que le calcul soit théoriquement faisable à la main par une personne, obtenir des résultats précis en un temps raisonnable n'est en pratique pas réalisable, sauf par une machine.

L'objectif est donc d'écrire des algorithmes de résolution qui approximeraient une solution réelle du problème voulu, en faisant attention aux imperfections des ordinateurs modernes (notamment les questions de précision flottante) et le tout bien sûr en un temps le plus court possible, même si de nos jours il devient question de préférer des algorithmes moins énergivores aux plus rapides.
Le développement de telles méthodes numériques n'est pas une activité triviale et énormément de paramètres doivent être pris en compte afin d'obtenir un algorithme aux performances et résultats idéaux. Ici, nous allons présenter plusieurs façons différentes de résoudre l'équation de la chaleur en une dimension, ou problème de Poisson 1D.

Nous présenterons d'abord le problème susmentionné, puis nous proposerons plusieurs méthodes numériques permettants sa résolution que nous avons implementé dans le langage C. Nous discuterons alors des résultats obtenus avec chaque méthode ainsi que de leurs points positifs et négatifs respectifs.

2 Équation de la chaleur

2.1 Présentation

L'équation de la chaleur cherche à résoudre l'évolution de la température d'un milieu donné selon des valeurs de température initiales à certains endroits. ici, nous travaillerons en un dimension, dasn un milieu immobile, linéaire et homogène. Nous avons alors le problème différentiel suivant :

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

avec $T_0 < T_1$ les températubes aux bords du domaine étudié, g terme source, $k > 0$ le coefficient de conductivité thermique.
Nous considérerons pour la suite qu'il n'y a pas de source de chaleur, ce qui nous donne la solution analytique suivante :

$$T(x) = T_0 + x(T_1 - T_0) \tag{2}$$

2.2 Discrétisation

Pour la résolution, nous allons discrétiser l'équation par une méthode de différence finie centrée d'ordre 2, selon $n+2$ nœuds x_i espacés d'un pas $h = \frac{1}{n+1}$ tel que $x_0 = 0, x_{i+1} = x_i + h$. L'équation s'écrit donc au nœud i :

$$-k(\frac{\partial^2 T}{\partial x^2})_i = g_i \tag{3}$$

On notera donc pour la suite le système $Au = f$, $A \in \mathbb{R}^{n\times n}, u, f \in \mathbb{R}^n$

2.2.1 Approximation de la dérivée seconde

Nous souhaitons maintenant approximer la dérivée seconde de l'équation. Soit la formule de Taylor-Young qui nous indique que f admet un développement limité à l'ordre n en x tel que :

$$f(x+h) = f(x) + f'(x)h + ...\frac{f^{(n)}(x)}{n!} + o(h^n) \tag{4}$$

Appliqué à notre situation, nous obtenons :

$$T(x+h) = T(x) + hT'(x) + \frac{h^2}{2}T''(x) + o(h^3) \tag{5}$$

$$T(x+h) = T(x) - hT'(x) + \frac{h^2}{2}T''(x) + o(h^3) \tag{6}$$

Nous obtenons alors :

$$T(x+h) + T(x-h) = 2T(x) + h^2T''(X) + o(h^3) \tag{7}$$

Ce qui donne après substitution :

$$T''(x) = \frac{1}{h^2}(T(x+h) - 2T(x) + T(x+h) + o(h)) \tag{8}$$

$$T''(x) \approx \frac{1}{h^2}(T(x+h) - 2T(x) + T(x+h)) \tag{9}$$

Nous avons alors comme résultat :

$$\frac{\partial^2 T}{\partial x^2}(x) = \frac{1}{h^2}(T(x+h) - 2T(x) + T(x+h)) \tag{10}$$

2.2.2 Système linéaire

Nous voulons maintenant représenter le problème sous forme d'un système linéaire de dimension n.

Soit $T(0) = T_0 = g_0$.

Soit à résoudre le système linéaire $Ax = b$ tel que

$$b = \begin{pmatrix} g_0 \\ \vdots \\ g_{n+1} \end{pmatrix}$$

et

$$x = \begin{pmatrix} T_0 \\ \vdots \\ T_{n+1} \end{pmatrix}$$

En remplaçant nous avons :

$$A \begin{pmatrix} T_0 \\ \vdots \\ T_{n+1} \end{pmatrix} = \begin{pmatrix} g_0 \\ \vdots \\ g_{n+1} \end{pmatrix}$$

D'après cette identité, nous en déduisons que :

$$g_1 = \frac{-T_0 + 2T_1 + T_2}{h^2} \tag{11}$$

et donc de manière plus générale :

$$g_n = \frac{-T_{n-1} + 2T_n + T_{n+1}}{h^2} \tag{12}$$

Or, nous connaissons la valeur de g_0 qui vaut T_0. Nous pouvons donc substituer dans (11) :

$$g_1 = \frac{-g_0 + 2T_1 + T_2}{h^2} \tag{13}$$

Nous avons alors :

$$\frac{2T_1 - T_2}{h^2} = g_1 \frac{kg_0}{h^2} \tag{14}$$

et donc

$$g_i = \frac{-g_{i-1} + 2T_n + T_{n+1}}{h^2} \tag{15}$$

Nous en déduisons alors la forme finale :

$$\frac{k}{h_2} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots \\ \vdots & & & & \\ \cdots & 0 & -1 & 2 & -1 \\ \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} T_1 \\ \vdots \\ T_n \end{pmatrix} = \begin{pmatrix} \frac{g_i k g_{i-1}}{h^2} \\ \vdots \\ \frac{g_n k g_{n-1}}{h^2} \end{pmatrix} \tag{16}$$

3 BLAS et LAPACK

3.1 Présentation

Nous utiliserons deux bibliothèques très répandues, Blas(Basic Linear Algebra Subprograms) et Lapack (Linear Algebra Package). Ces librairies, apparues dans les années 1970, ont pour objectif de simplifier et d'optimiser les opérations d'algèbre linéaire classiques.

- Blas se concentre sur des opérations de base (produit matriciel, matrice-vecteur, produit scalaire...) et tente de les optimiser à un très bas niveau (par exemple en utilisant la vectorisation, qui permet de travailler sur plusieurs éléments contigues d'une liste en même temps). De plus, Blas est régulièrement mis à jour afin de tirer profit des nouvelles avancées en informatique afin d'être sûr que les routines restent les plus performantes possible. Il en ressort donc un code très optimisé qui est probablement l'un des plus efficaces pour ce genre d'opérations matricielles de base.

- Lapack implémente différentes méthodes et algorithmes utiles à la résolution de système linéaire, comme par exemple la décomposition de Cholesky, méthode de remontée etc... Il est commun de trouver des implémentations de Lapack qui utilisent intensivement Blas pour les opérations matricielles de bas niveau que l'on retrouve régulièrement en algèbre linéaire et obligatoires dans la plupart des algorithmes de résolution.

Blas et Lapack ont tous deux d'abord été développés en Fortran avant d'avoir des implémentations en C. Cependant, ces deux langages ont quelques différences quant à leur représentation en mémoire de matrices, ce qui n'est pas sans conséquence dans l'utilisation de Blas et Lapack.

3.2 Stockage mémoire

Une matrice classique contient deux dimensions, qui peuvent être stockés soit ligne par ligne, soit colonne par colonne. Connaître ce détail d'implémentation dans les technologies qu'on utilise est important car il peut fortement impacter les performances d'un programme si mal utilisé (notamment pour les questions d'accès au cache qui est sensé réduire les pertes liées à la lecture en mémoire). Là où le Fortran stocke ses matrices colonne par colonne (ColMajor), le C préfère un stockage par ligne (RowMajor). Lors de la transition du Fortran vers le C, il fallait donc mettre en place des sécurités pour s'assurer que les algorithmes n'effectuaient pas leurs opérations matricielles "à l'envers", ie parcourir les lignes au lieu des colonnes dans une boucle.

Cela se traduit généralement par un paramètre à donner en entrée des fonctions pour indiquer le format de stockage utilisé, les méthodes peuvent alors si nécessaire transposer les matrices d'entrées avant de débuter les vrais calculs et ainsi obtenir des résultats corrects.

Ce paramètre est souvent nommé "Leading Dimension" ou ld et peut être indiqué grâce à des constantes déjà présentes dans les librairies telles que LAPACK_COL_MAJOR qui indique un stockage par colonne.

D'autres questions plus générales se posent à propos du stockage des matrices. En effet, il va souvent être question de stocker des matrices très volumineuses et cela pose problème car nous ne disposons que d'un espace mémoire limité. Cependant, il est possible de tirer profit du format spécial de certaines matrices pour optimiser leur stockage en mémoire, et éventuellement aussi améliorer la rapidité des algorithmes. Blas et Lapack peuvent prendre en compte plusieurs types de formats, il en revient donc à l'utilisateur de les formater au préalable afin de tirer profit de cette fonctionnalité. Les formats les plus populaires sont pensés pour les matrices creuses qui par définition contiennent beaucoup d'éléments nuls et qu'il est donc inutile de stocker.

3.2.1 COO

La méthode de stockage optimisée la plus simple est appelée COO (pour CO-Ordinates). Son principe est de simplement stocker les informations dans trois tableaux différents :

- Le premier contient simplement les valeurs non-nulles de la matrice, stockées de préférence dans leur ordre d'apparation dans ladite matrice mais cela n'est pas obligatoire.
- La seconde liste "JR" contient les indices de ligne de chacun de ces éléments.
- La troisème et dernière liste "JC" contient les indices de colonne des éléments.

Ce formet est simple d'utilisation si bien organisé mais n'est pas optimal car il va présenter certaines répétitions, notamment stocker plusieurs fois les mêmes indices de ligne pour tous les éléments présents sur une même ligne (il en va de même pour les colonnes). La méthode de stockage suivante permet de mitiger ce problème.

3.2.2 Compressed Sparse

Il existe deux versions de ce stockage, le Compressed Sparse Row (CSR) et le Compressed Sparse Column (CSC). Ils partagent le même principe, seulement CSR travaille sur les lignes, CSC sur les colonnes.

Le format CS va, similairement au COO, stocker tous les éléments non nuls dans un tableau "AA" de manière ordonnée. Cette fois cependant, seulement l'un des deux indices sera stocké pour chaque valeur dans un tableau "JA". L'autre indice est déterminé grâce à un tableau "IA" qui fonctionne selon le principe suivant :

- Toutes les lignes (ou colonnes) sont stockées de manière continue dans AA.

- Chaque entrée de donne l'indice de début de chaque ligne (ou colonne) dans AA.

Concrètement, si la première ligne de la matrice contient deux valeurs non nulles, la seconde valeur de IA, correspondant au début de la deuxième ligne de la matrice, indiquera "3" car cette deuxième ligne commence à partir de la troisième valeur stockée dans AA.

Ces méthodes de stockage sont efficaces mais demandent de gérer trois listes différentes, ce qui est malheuresement inévitable dans ce cas-là. Cependant il existe une méthode plus efficace pour les matrices diagonales qui est toujours stockée dans un tableau 2D.

3.2.3 General Band

Dans un stockage general Band (GB), nous gardons un format type tableau à double dimension, seulement nous parvenons à éliminer une ou plusieurs lignes de la matrice diagonale originale. Il en suit donc que les éléments restent à leur indice de colonne initial, seuls les indices de ligne changent. Le principe part du fait que comme chaque colonne contient un nombre de ligne inférieur à la dimension n de la matrice, il est possible de les "aligner" dans une matrice au nombre de lignes réduit.
Si nous prenons l'exemple d'une matrice diagonale quelconque "A":

$$A = \begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

Figure 1. T. Dufaud, CM4

sa représentation en General Band "AB" sera de la forme :

$$AB = \begin{pmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{pmatrix}$$

Figure 2. T. Dufaud, CM4

où les éléments $A(i,j)$ sont stockés dans $AB(ku+1+i-j,j)$ pour un ombre de lignes finales de $ku+kl+1$ avec ku le nombre de sur-diagonales de A et kl les sous-diagonales de A. Les matrices stockées en bandes sont indiquées dans Lapack par un nom se terminant en "B".

Les fonctions de Blas et Lapack prennent en compte ces formats de stockage et il faut savoir quelles fonctions utiliser selon la méthode de stockage choisie. Il est maintenant temps de discuter plus précisément de certaines de ces fonctions et des formats de stockage qu'elles utilisent.

3.3 Méthodes

3.3.1 Nomenclature

Les fonctions Lapack suivent une logique de dénomination précise qui permet, quand on la maîtrise, de reconnaître facilement la fonctionnalité d'une fonction. Les noms suivent le format "pmmaaa" :

- p indique le type de donnée utilisé; S et D pour simple/double real floating-point arithmetic, C et Z pour simple/double complex arithmetic.
- mm indique le type de matrice utilisé. C'est ici que nous pouvons utiliser plusieurs types de stockage différents (GB pour General Band, DI pour Diagonal Matrix, GT pour General Tridiagonal Matrix...).
- aaa dénote l'algorithme utilisé par la fonction (SV pour la résolution de système linéaire).

3.3.2 Méthodes courantes

Voici plusieurs routines de base fréquemment utilisées (notons que le gb dans leurs noms précise qu'elles attendent toutes en entrée des matrices en format bande):

- $dgbmv$ effectue un produit matrice vecteur.
- $dgbtrf$ effectue une factorisation LU de la matrice d'entrée en utilisant un pivot partiel.
- $dgbtrs$ résoue un système linéaire $Ax = b$. Cette résolution utilise une factorisation LU effectuée par $dgbtrf$.
- $dgbsv$ calcule la solution d'un système linéaire $Ax = b$ en utilisant la décomposition LU de la matrice A.

Il est important de prendre connaissance de ces fonctions de base car nous allons en avoir besoin pour la suite, elles sont à la base des méthodes de résolution en algèbre linéaire.

4 Performances et résultats

Il existe plusieurs types d'algorithmes qui permettent de résoudre des système linéaires, nous pouvons séparer ce que nous verrons pour le moment en deux parties :

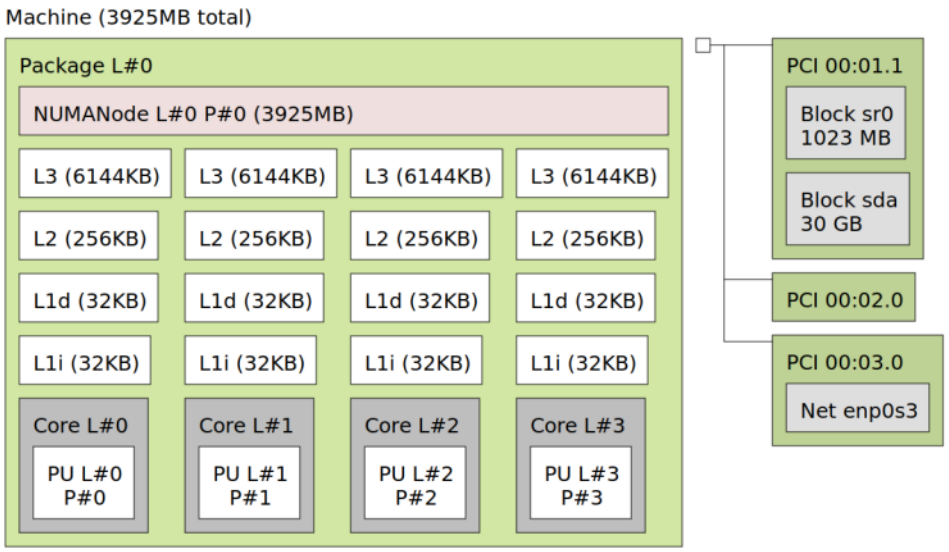
- Les méthodes directes qui tentent de résoudre le système en une seule passe.
- Les méthodes itératives qui construisent une solution approchée à chaque étape qui converge petit à petit vers la solution réelle.

Nous avons implémenté certains de ces algorithmes et nous souhaitons étudier leurs performances sur le problème de Poisson 1D selon plusieurs tailles de matrices différentes, les métriques principales à étudier étant la vitesse d'exécution ainsi que la convergence de la solution calculée.

Nous avons pris plusieurs mesures afin d'assurer que les mesures des temps d'exécution se fassent dans un environnement le plus stable possible. Notamment, avant chaque lancement, nous avons utilisé les commandes suivantes :

- **cpupower** qui permet de fixer la fréquence du processeur pour par exemple empêcher le système d'exploitation de rentrer dans des modes économies d'énergie ce qui affecterait le temps mesuré.
- **taskset** qui permet de fixer le processus de notre programme sur un cœur de calcul unique et éviter de potentielles migrations pendant l'exécution.

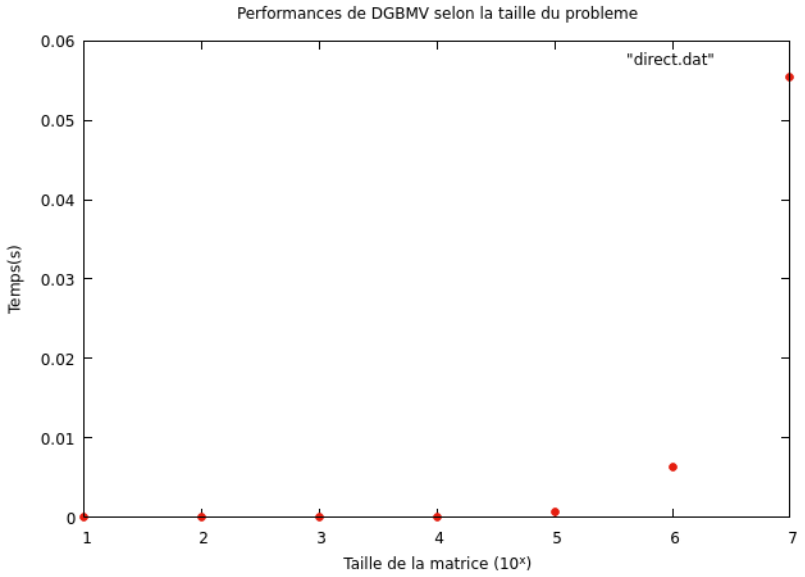
Les exécutions ont été répétées 10 fois afin d'obtenir une moyenne stable pour chaque mesure. Le code a été utilisé sur l'architecture NUMA suivante, qui correspond à un processeur i5-8265U (1.6Ghz) :



4.1 Résultats

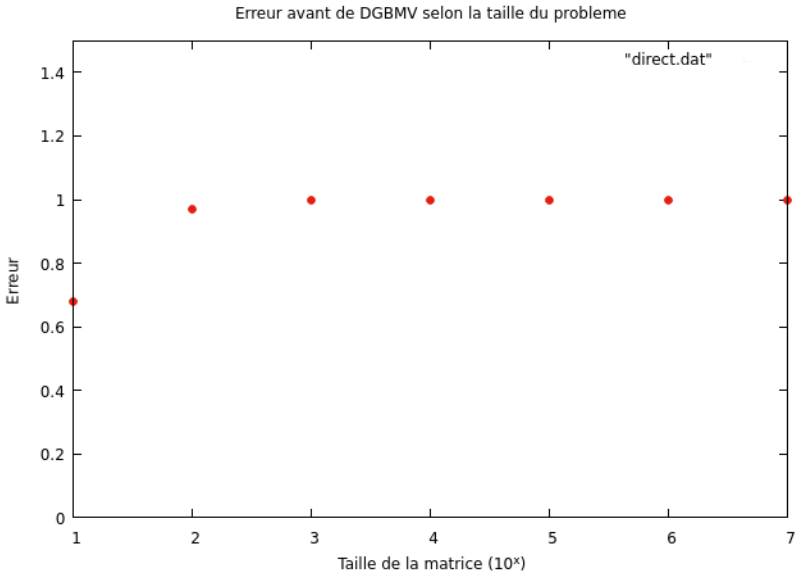
4.1.1 Méthodes directes

Nous avons d'abord écrit une méthode de conversion de matrice en stockage General Band, que l'on a ensuite utilisé avec *dgbmv*. Comme indiqué plus haut, nous avons mesuré le temps d'exécution selon plusieurs n différents (de 10^1 a 10^7). Comme nous disposons de la solution analytique du problème, nous pouvons l'utiliser afin d'estimer la précision de notre solution.



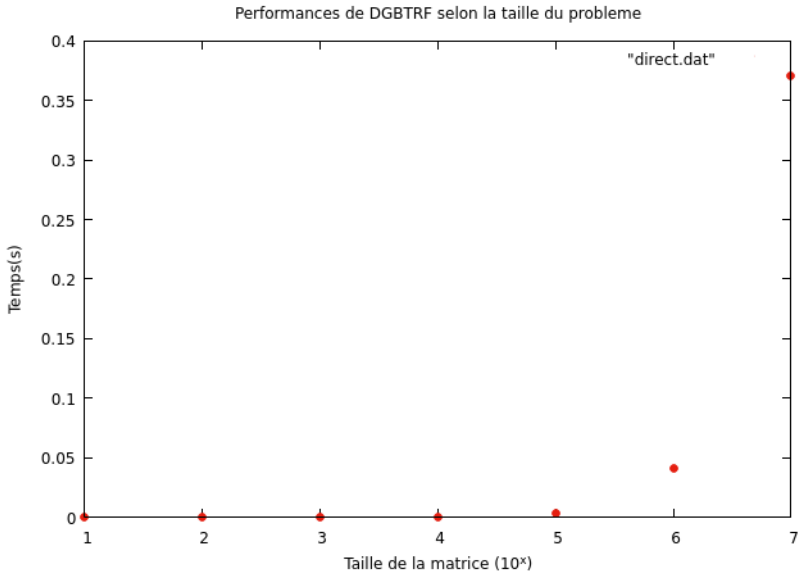
Pour les plus petites tailles de matrice, nous constatons une évolution assez inégale du temps d'exécution. La raison principale pouvant être que le temps de résolution du problème reste très faible et est donc difficile à mesurer avec précision, certains artéfacts brouillent alors la mesure car ils prennent une part importante relativement à l'exécution de la fonction (appel de fonction, accès initiaux lents à la mémoire...). Ce problème se répètera d'ailleurs pour toutes le sprochaines mesures.

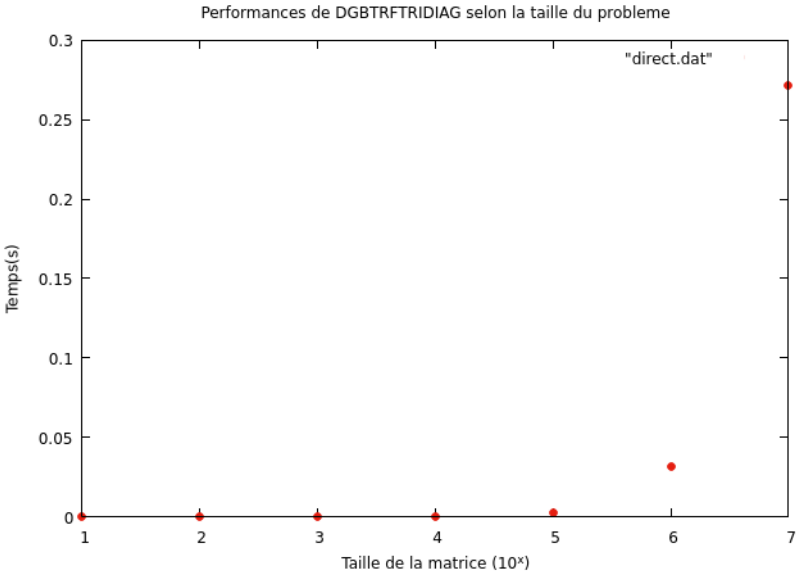
Si nous voulons avoir une idée plus nette des performances de l'algorithme, il faut donc regarder les temps d'exécution pour de plus grande taille de problème. Nous voyons en effet une évolution nette et non négligeable du temps d'exécution à partir de 10^5, qui semble "exploser". En observant les valeurs de plus près, nous observons une augmentation du temps relativement linéaire (x10) correspondant à l'augmentation de la taille de la matrice. Il aurait été intéressant de voir si cette observation reste valide pour de plus grandes taille de matrices mais nous sommes limités par la mémoire à notre disposition et des programmes plus ambitieux ne parvenaient pas à terminer leur exécution.



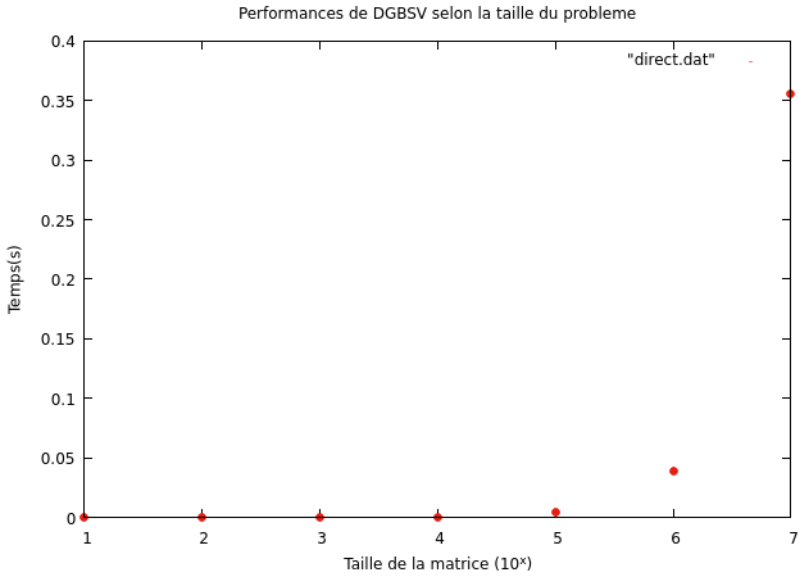
Concernant l'erreur avant de la solution approchée, nous observons clairement qu'elle semble stagner aux alentours de $9.9e-01$, une erreur loin d'être négligeable et qui montre que notre solution présente quelques défauts de précision, peut-être dûs aux problèmes du calcul flottant et des optimisations intensives de Blas.

Les observations sur l'évolution du temps d'exécution de $dgbtrf$ sont similaires, il est cependant intéressant de comparer ces mesures avec celles de $dgbtrftridiag$.





Il est clair que $dgbtrftridiag$ est plus rapide que $dgbtrf$ avec une amélioration de près de 30% pour un n de 10^7. Cela montre donc qu'une factorisation LU pour un système tridiagonal est plus efficace que pour un système triangulaire ou plein.



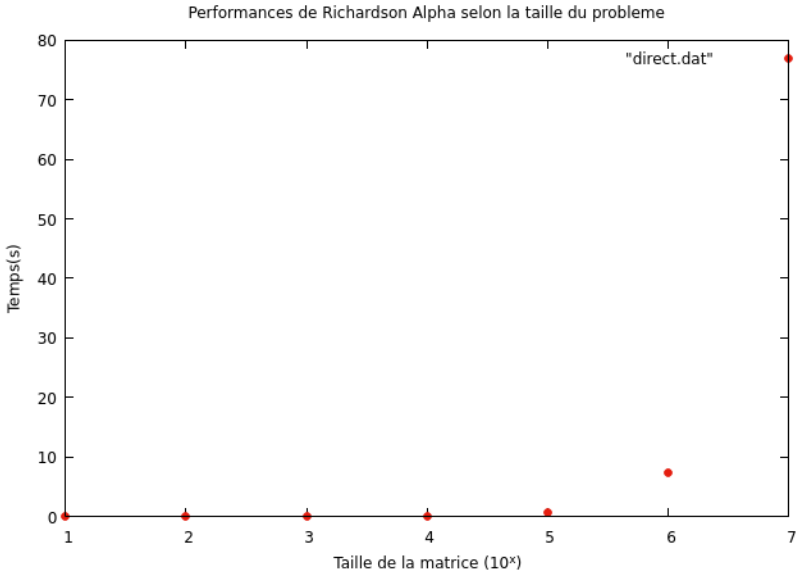
dgbsv calcule la solution d'un système linéaire $Ax = b$ en utilisant la décomposition LU de A, calculée à partir d'autres fonctions que nous avons mentionnés, comme $dgbtrftridiag$, il est donc logique de s'attendre à des temps d'exécutions évoluant de manières similaires. Nous voyons donc que les résultats coincident avec ceux de $dgbtrftridiag$, *dgbsv* étant naturellement plus lent car il doit en plus résoudre le système linéaire.

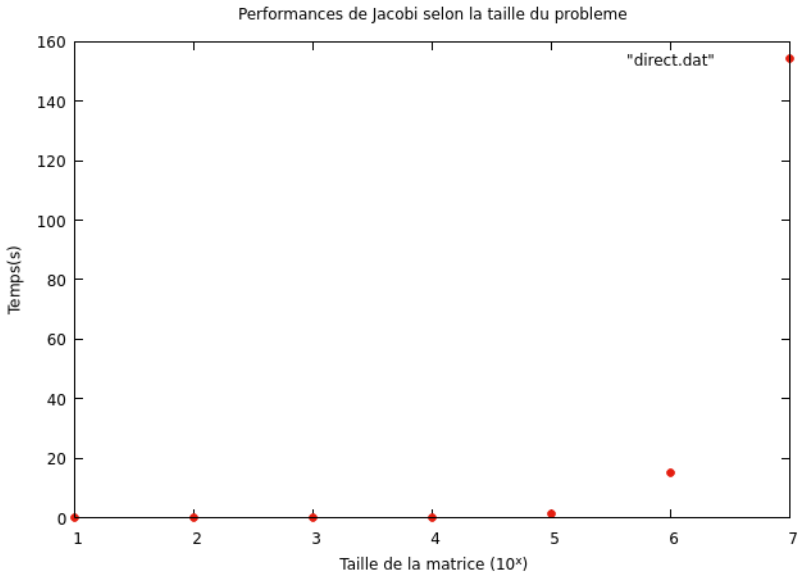
4.1.2 Méthodes itératives

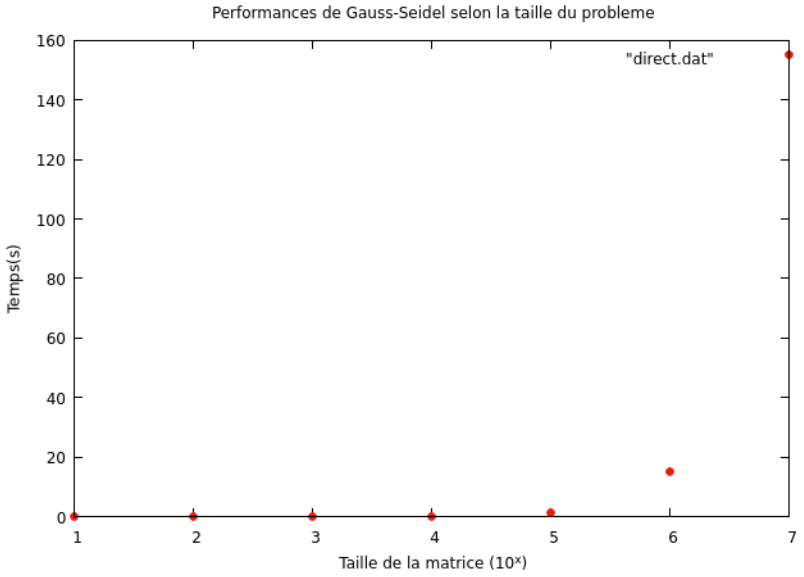
Passons maintenant aux méthodes itératives qui ont pour but de résoudre un système linéaire:

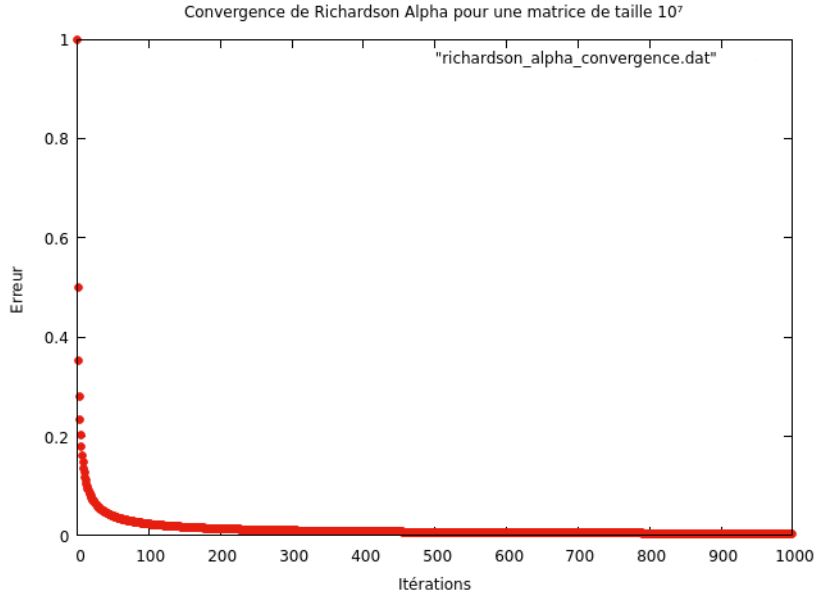
- Richardon Alpha
- Jacobi
- Gauss-Seidel

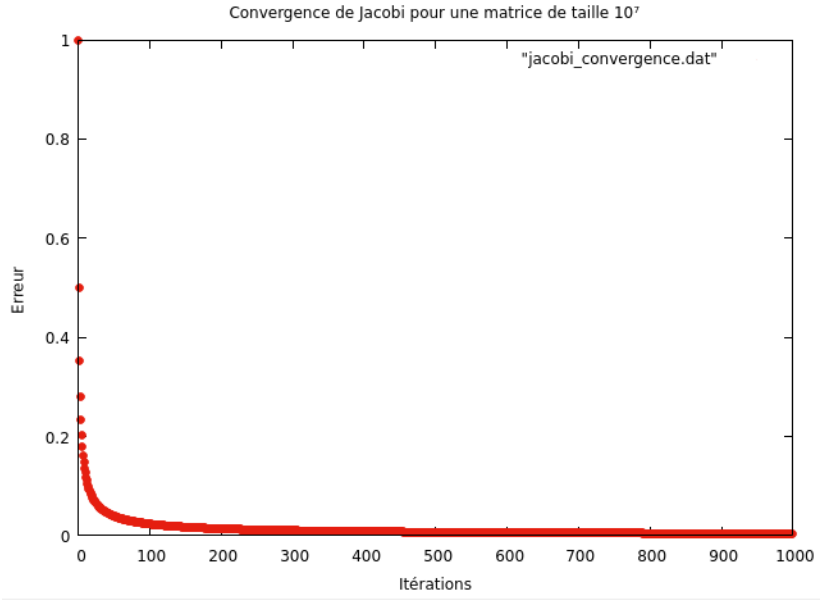
En plus de comparer les temps d'exécutions, nous allons étudier la convergence de ces algorithmes qui est un autre point très important à considérer pour des méthodes numériques.

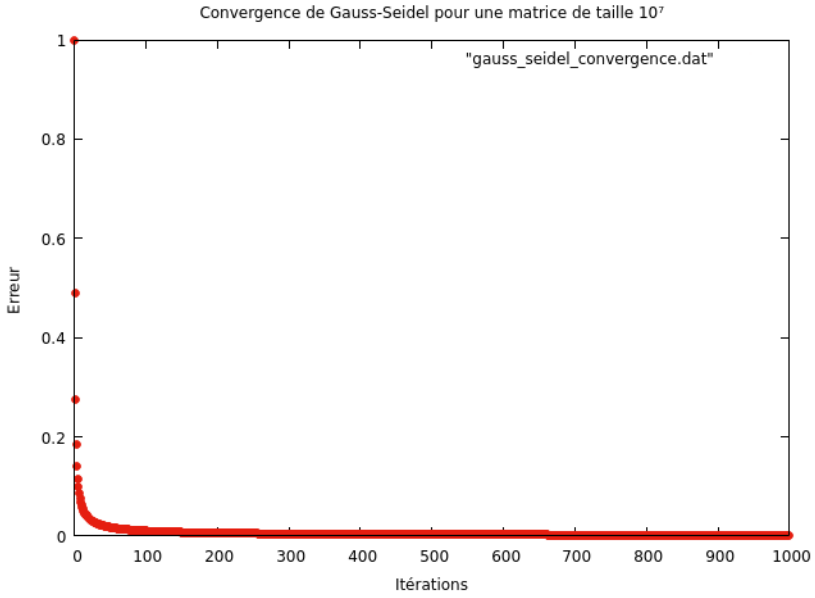



Performances de Jacobi selon la taille du probleme
"direct.dat"
Temps(s)
Taille de la matrice (10ˣ)
0
20
40
60
80
100
120
140
160
1
2
3
4
5
6
7


Performances de Gauss-Seidel selon la taille du probleme
"direct.dat"
Temps(s)
Taille de la matrice (10ˣ)
0
20
40
60
80
100
120
140
160
1
2
3
4
5
6
7


Convergence de Richardson Alpha pour une matrice de taille 10^7
"richardson_alpha_convergence.dat"
Erreur
Itérations
0
0.2
0.4
0.6
0.8
1
0
100
200
300
400
500
600
700
800
900
1000


Convergence de Jacobi pour une matrice de taille 10^7
"jacobi_convergence.dat"
Erreur
Itérations
0
0.2
0.4
0.6
0.8
1
0
100
200
300
400
500
600
700
800
900
1000



Nous observons dans les trois cas que la vitesse de convergence diminie d'abord rapidement, puis semble stagner car les améliorations successives deviennent de plus en plus faibles, comme s'il devenait plus difficile de s'approcher de la solution.
En analysant plus en détails ces historiques de convergence, nous voyons que la méthode de Gauss-Seidel semble avoir une convergence initiale beaucoup plus rapide que les deux autres, mais c'est aussi celle qui a le plus long temps d'exécution.

5 Conclusion

Nous avons discuté des différentes problématiques survenant lors de l'utilisation de l'informatique pour la résolution de problème numérique, comment les reconnaître et les mitiger au possible. Ensuite, nous avons parlé de différentes méthodes classiques très utilisées en algèbre linéaire et de comment nous pouvons les optimiser. Nous avons finalement implementé nous-même certaines de ces méthodes et avons mésuré plusieurs métriques de performance afin de déduire si les résultats expérimentaux concordent avec la théorie.