Equation chaleur 1D stationnaire

Julien Rigal

Semestre 1

# Partie I

# Introduction

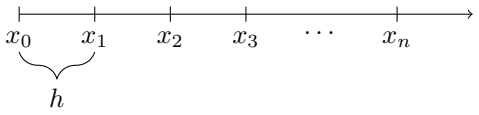
L'objectif de ce TD/TP est d'appliquer un ensemble d'algorithmes étudiés en cours et TD pour la resolution d'un systeme lineaire obtenu par discretisation par la méthode des differences finies de l'équation de la chaleur 1D stationnaire.

Soit à résoudre l'équation de la chaleur dans un milieu immobile linéaire homogène avec terme source et isotrope :

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

Où g est un terme source, $k > 0$ le coefficient de conductivité thermique, et $T_0 < T_1$ les températures au bord du domaine considéré.

Résolvons cette équation avec la méthode des différences finies d'ordre 2. La discrétisation de l'espace sera la suivante :




On peut à présent retourner à un problème de la forme $A * u = b$ en considérant $u = T(x)$.

La première étape consistera en un développement de Taylor à l'ordre 2 qui nous donnera le système d'équation suivant :

$$\begin{cases} u(x_i + h) = u(x_i) + hu'(x_i) + \frac{h^2}{2}u''(x_i) + \mathcal{O}(h^3), \\ u(x_i - h) = u(x_i) - hu'(x_i) + \frac{h^2}{2}u''(x_i) + \mathcal{O}(h^3). \end{cases}$$

En sommant, on obtient :

$$u(x_i + h) + u(x_i - h) = 2u(x_i) + h^2u''(x_i) + \mathcal{O}(h^3).$$

$$\iff$$

$$u''(x_i) + \mathcal{O}(h^3) = \frac{2u(x_i) - u(x_i + h) - u(x_i - h)}{h^2}$$

Remplaçons $u''(x_i)$ par $g(i)$ pour une meilleure lisibilité, on peut alors reformuler l'équation originale.

$$Au = g$$

$$\iff \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = h^2 \begin{bmatrix} g_1 + T_0 \\ g_2 \\ g_3 \\ \vdots \\ g_n + T_1 \end{bmatrix}$$

$$\iff \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = h^2 \begin{bmatrix} T_0 \\ 0 \\ 0 \\ \vdots \\ T_1 \end{bmatrix}$$

Car il n'y a pas de terme source, donc $\forall i \in [0, n]\ g_i = 0$

En reprenant $u = T(x)$, on peut donc conclure quant à la solution analytique :

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

# Partie II

# Methode directe et stockage bande

## Exercice 3 : Référence et utilisation de BLAS/LAPACK

1. **Déclaration et allocation d'une matrice pour BLAS et LAPACK :**
   En C, une matrice doit être déclarée comme un tableau 1D pour BLAS et LAPACK, En C, on peut imaginer la déclaration suivante pour une matrice A∈MxN :

   ```
   #include <stdlib.h>
   int m = 3, n = 4;
   double *matrix = (double *)malloc(m * n * sizeof(double));
   ```

2. **Signification de LAPACK_COL_MAJOR :**
   Le mot clef LAPACK COL MAJOR fait référence à l'ordre de stockage de la matrice. Comme le nom l'indique, cette constante précise que notre matrice à été stockée selon les colonnes (on parcourt les lignes avant de passer à la colonne suivante).
3. **Dimension principale (*ld*) :**
   La dimension principale est le pas en mémoire entre deux colonnes successives de la matrice. Cela revient au nombre de ligne d'une matrice.
4. **Fonction dgbmv :**
   La fonction `dgbmv` calcule le produit matrice-vecteur pour une matrice bande :
   $$y = \alpha Ax + \beta y$$
   où $A$ est une matrice bande définie par ses bandes diagonales supérieures ($ku$) et inférieures ($kl$).
5. **Fonction dgbtrf :**
   La fonction `dgbtrf` vient de la bibliothèque LAPACK qui effectue la factorisation LU pour une matrice en format bande. Elle décompose une matrice $A$ en :

$$A = P \cdot L \cdot U$$

— $P$ est une matrice de permutation.
— $L$ est une matrice triangulaire inférieure avec des 1 sur la diagonale.
— $U$ est une matrice triangulaire supérieure.

6. **Fonction dgbtrs :**
   La fonction `dgbtrs` résout un système linéaire $Ax = b$, où $A$ a été factorisée au préalable par `dgbtrf`.
7. **Fonction dgbsv :**
   La fonction `dgbsv` effectue une résolution directe d'un système $Ax = b$ pour une matrice bande $A$, combinant les opérations de `dgbtrf` et `dgbtrs`.
8. **Norme du résidu relatif :**
   La norme du résidu relatif $r = b - Ax$ peut être calculée avec les fonctions BLAS :
   (a) Calculer $Ax$ avec `dgemv` ou `dgbmv`.
   (b) Calculer $r = b - Ax$.
   (c) Calculer la norme $\|r\|$ avec `dnrm2`.
   (d) La norme relative est donnée par Relatif $= \|r\|/\|b\|$.

# Exercice 4 : Stockage GB et appel à dgbmv

1. **Stockage GB pour la matrice de Poisson 1D :**
   Pour une matrice tridiagonale $A$ de Poisson 1D, seules les bandes non nulles sont stockées. Par exemple :

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

   devient :

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

2. **Utilisation de dgbmv :**
   Pour calculer $y = Ax$ avec `dgbmv`, il faudra ajouter la ligne suivante au coeur de la fonction :

```
dgbmv("N", n, n, kl, ku, alpha, AB, ldab, x, incx, beta, y, incy);
```

   Nous utiliserons un environnement Docker, possédant les bibliothèques BLAS et LAPACK.

3. **Validation :**
   Une méthode de validation consisterait à comparer les résultats de `dgbmv` avec un produit matrice-vecteur effectuée sans la méthode, et avec la même matrice.

# Exercice 5 : DGBTRF, DGBTRS, DGBSV

1. **Résolution directe :**
   Pour résoudre un système linéaire de type $Ax = b$ avec une méthode directe, et en faisant appel à LAPACK, nous procéderons en trois étapes :
   — `DGBTRF` : Se chargera de la factorisation LU de la matrice bande $A$ (comme vu à la question cinq de l'exercice 3).
   — `DGBTRS` : Utilise cette factorisation $P \cdot L \cdot U$ pour résoudre le système linéaire (pour un ou plusieurs seconds membres).
   — `DGBSV` : Combine les deux étapes précédentes pour une résolution directe du système.
2. **Évaluation des performances :**
   Ces fonctions exploitent la structure bande de la matrice, et la complexité de la factorisation est $\mathcal{O}(n \cdot (K_L + K_U)^2)$, où $kl$ et $ku$ représentent respectivement le nombre de bandes sous et au-dessus de la diagonale.

   Pour une matrice dense de taille $n \times n$, la factorisation LU a une complexité de $\mathcal{O}(n^3)$, et $\mathcal{O}(n^2)$ pour la résolution. Par conséquent, l'utilisation de ces méthodes spécialement optimisées pour les matrices bandes est bien plus efficace.

# Exercice 6 : LU pour les matrices tridiagonales

1. **Implémentation de la méthode de factorisation LU pour les matrices tridiagonales avec le format GB.**

   Nous travaillerons donc sur une matrice bande (GB), afin d'optimiser l'algorithme. Une méthode directe de factorisation LU sera implémentée, nous stockerons L et U dans le même format pour une meilleure optimisation. Nous devrions avoir une complexité de $\mathcal{O}(n)$.
2. **Validation.**

   Nous pourrions procéder de la sorte :

   (a) Factoriser $A$ en $L \cdot U$.
   (b) Résoudre le système $L \cdot U \cdot x = b$ en deux fois :
      — $L \cdot y = b$ pour $y$ (résolution avant),
      — $U \cdot x = y$ pour $x$ (résolution arrière).

   Nous obtiendrons donc un x, qu'il faudra comparer avec une solution exacte par exemple.

Nous pouvons calculer l'erreur relative à l'aide de la formule suivante :

$$\frac{\|A \cdot x - b\|_2}{\|b\|_2}.$$

Et vérifier que cette erreur est inférieure à un seuil donné.

# Partie III

# Méthode de résolution itérative

À présent, nous allons travailler sur trois méthodes de résolution itératives. Richardson, Jacobi, et Gauss Seidel. Ces dernières trouvent leur utilité lorsque les matrices avec lesquelles nous travaillons sont de taille trop importantes, ou creuses. Dans ces cas là, les méthodes directes deviennent inefficaces.

Lors de cette partie, je vais traiter chacune des trois approches à part. Les expliquer, puis mesurer leur efficacité après implémentation en C.

## Exercice 7 : Richardson

Tout d'abord, introduisons une notion importante, le résidu $r$. Il est défini par la formule suivante :

$$r = b - Ax$$

Cette notion est centrale dans la méthode de Richardson. En effet, le but sera d'itérer sur une solution x approximative afin de réduire le résidu. Nous ferons cela après avoir défini au préalable une variable appelée "tolérance". Dès lors que le résidu sera inférieur à sa valeur, nous considérerons qu'il y a convergence.

L'approximation à l'itération k sera de cette forme :

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

$\alpha$ est appelé paramètre d'accélération. Cette constante est strictement positive, et ne dépasse pas une certaine valeur, liée à la matrice A. L'encadrement est le suivant :

$$0 < \alpha < \frac{2}{\|A\|}$$

Où $\|A\|$ est la plus grande valeur propre de A.

À noter, la méthode de Richardson est une méthode "générale", où $\alpha$ n'a pas d'autre caractéristique que son domaine d'existence. Il joue un rôle très important dans l'efficacité de cette dernière, aussi c'est sur lui qu'il faut travailler. Les deux méthodes suivantes seront en fait des cas plus spécifiques, où l'on fera varier ce terme.

Pour cet exercice, l'idée était donc de calculer le $\alpha$ optimal. Pour cela, il nous a fallu implémenter des fonctions qui trouvaient les valeurs propres maximales et minimales. Une fois cela fait, il n'y avait plus qu'à poser :

$$\alpha = \frac{2}{\lambda_{min} + \lambda_{max}}$$

Une fois cela fait, nous pouvons mesurer les performance d'une telle méthode. Pour ce faire, penchons nous sur les opérations effectuées lors de chaque itération :

— **Multiplication matrice vecteur** $(Ax^{(k)})$ :
La complexité pour une telle opération est la suivante : $\mathcal{O}(n^2)$ pour une matrice dense, ou $\mathcal{O}(nnz)$ (nz étant le nombre d'éléments non nuls) pour une matrice creuse.

— **Calcul du résidu** $(b - Ax^{(k)})$ : $\mathcal{O}(n)$.

— **Mise à jour** $(x^{(k+1)} = x^{(k)} + \alpha r^{(k)})$ : $\mathcal{O}(n)$.

**Au total, par itération** :
Au final, ce sera la complexité de la multiplication qui domine le reste. On aura donc $\mathcal{O}(n * nz)$ **pour une matrice creuse**, ou bien $\mathcal{O}(n^2)$ **pour une matrice dense.**

Pour ce qui est du stockage :

— **La matrice** $A$ :

$\mathcal{O}(n^2)$ pour une matrice dense, $\mathcal{O}(n * nz)$ pour une matrice creuse.

— **Les vecteurs** $x^{(k)}$**,** $r^{(k)}$ :

$\mathcal{O}(n)$.

**Le total à stocker en espace** :

$\mathcal{O}(n^2 + n)$ pour une matrice dense, $\mathcal{O}(n * nz + n)$ pour une matrice creuse.

## Exercice 8 : Jacobi

Comme vu précédemment, la méthode itérative de Jacobi est en fait une application de celle de Richardson pour un $\alpha$ spécifique. Ici, on aura :

$$\alpha = D^{-1}$$

Où $D^{-1}$ est l'inverse de la matrice diagonale de A.

La méthode va donc reposer sur l'extraction de la diagonale principale de A, pour cela nous implémentons une fonction qui effectura cette tâche, et que nous appelerons dans la fonction principale.

La fonction principale quant à elle, devra à chaque itération mettre à jour x (en utilisant les valeurs de la diagonale de A stockée dans MB) et les valeurs actuelles de x.

Cela ce fait en deux temps :

1. `cblas_dgbmv` calcule la multiplication $(A - M)x$ et met à jour $Bx$ en utilisant $Bx = b - (A - M)x$.
2. Les nouvelles valeurs de $x$ sont calculées selon la méthode de Jacobi :

$$x_i^{(k+1)} = \frac{Bx_i}{M_{ii}}$$

Pour ce qui est de la complexité, nous pourrions penser de prime abord qu'elle soit identique, cependant il faut prendre en compte le fait que ce qui joue le rôle d'$\alpha$ est une matrice tridiagonale, ce qui allège grandement la complexité.

Pour la complexité arithmétique, le terme dominant sera la multiplication matrice-vecteur (*cblas dgbmv*), par conséquent on aura :

$$O((kl + ku) * la)$$

Pour la complexité en espace, la nature des objets à stocker est la suivante : Matrices (matrice bande $AB$) et Vecteurs ($RHS$, $BX$, $X$, $MB$). Cela resultera en :

$$O(\text{la} \cdot (\text{kl} + \text{ku})) + O(\text{la}) = O(\text{la} \cdot (\text{kl} + \text{ku}))$$

# Exercice 9 : Gauss Seidel

Pour la méthode itérative de Gauss Seidel, l'idée est de réécrire le système $A\mathbf{x} = \mathbf{b}$ sous la forme :

$$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j - \sum_{j=i+1}^{n} a_{ij} x_j \right), \quad i = 1, 2, \ldots, n$$

Où :
— $x_i$ est le $i$-ième composant du vecteur $\mathbf{x}$.
— $a_{ij}$ est un élément de la matrice $A$.

La différence majeure ici est que les valeurs déjà calculées, c'est à dire $(x_1, x_2, \ldots, x_{i-1})$, sont immédiatement utilisées.

À chaque itération, il se passe donc la chose suivante : On parcourt les lignes $i = 1, 2, \ldots, n$ de la matrice $A$ et on met à jour les valeurs de $x_i$ en utilisant les valeurs nouvellement calculées.

Comme pour les autres méthodes, le processus est répété jusqu'à ce que $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$ devienne inférieur à un seuil fixé (convergence).

Nous pouvons déjà supposer une performance meilleure, puisque les valeurs de x sont corrigées plus fréquemment.

Décomposons la complexité :

## Complexité arithmétique

Lors de la mise à jour des composantes $(x_i^{(k+1)})$
— Pour chaque itération, et dans le cas d'une matrice tridiagonale, seules les composantes $j = i - 1$ et $j = i + 1$ sont utilisées.
— Sur une ligne, il y a deux multiplications et additions. Donc pour $n$ lignes, la complexité est $\mathcal{O}(n)$ par itération.

Lors du calcul du résidu

$$r^{(k)} = b - Ax^{(k)}$$

est calculé avec une multiplication matrice-vecteur $Ax^{(k)}$. La matrice étant tridiagonale, la complexité est également $\mathcal{O}(n)$.

Total par itération : $\mathcal{O}(n)$.

## Complexité en espace

Pour le stockage des données :

— Pour une matrice tridiagonale au format GB, il y a besoin de $\mathcal{O}(n)$.
— Pour chacun des vecteurs, également $\mathcal{O}(n)$.

Au total, $\mathcal{O}(n)$.

En conclusion, la méthode de Gauss Seidel est très efficace vis à vis de la complexité arithmétique et en espace pour des matrices tridiagonales. Elle converge plus rapidement que Jacobi, à condition que les lignes soient parcourues judicieusement.

# Partie IV

# Autres formats de stockage

Cette partie est une ouverture sur d'autres formes de stockage. En effet, jusqu'à présent nous avons implémenté diverses méthodes de résolution de systèmes linéaires (*LU*, *Richardson*, *Jacobi* et *Gauss-Seidel*).

Que ces méthodes soient directes ou itératives, elles sont dans les deux cas spécifiquement adaptées aux **matrices tridiagonales** stockées sous un format ***GB* (General Band)**.

Le problème principal ici, est que même si ces formats sont bel et bien efficaces pour des matrices ayant une structure bien définie/régulières, ce ne sera pas forcément le cas général. Il sera possible d'avoir une matrice beaucoup plus large et creuse, avec des éléments non nuls dispersés de manière irrégulière.

Viennent alors les formats *CSR* (Compressed Sparse Row) et *CSC* (Compressed Sparse Column) trouvent une grande utilité pour réduire la complexité en mémoire et améliorer les performances des algorithmes.

## Les formats CSR et CSC

L'idée est de stocker de manière compacte uniquement les éléments non nuls d'une matrice creuse, avec un grand nombre d'éléments nuls. De cette façon, on n'utilise que les éléments "pertinents".

### CSR (Compressed Sparse Row)

Le format *CSR* est un format adapté aux opérations où l'on doit souvent parcourir les lignes de la matrice (méthodes itératives). Une matrice en format *CSR* est stockée dans trois tableaux (à une dimension) distincts :

- **Valeurs** : un tableau contenant tous les éléments non nuls de la matrice, ligne par ligne.
- **Colonnes** : un tableau contenant les indices de colonnes correspondants pour chacun d'entre eux.
- **Index de ligne** : un tableau indiquant où commence chaque ligne dans les tableaux des valeurs et des colonnes. Permet d'identifier les éléments

d'une ligne donnée sans avoir à tout parcourir.

Les tailles de ces trois tableaux sont respectivement :

— `values` : le nombre d'éléments non nuls.
— `columns` : idem que `values` (correspondant aux indices des colonnes).
— `row_ptr` : une taille de $N + 1$, où $N$ est le nombre de lignes.

Un exemple pour mieux comprendre :

$$\begin{bmatrix} 2 & 0 & 0 & 7 \\ 0 & 2 & 5 & 0 \\ 0 & 0 & 3 & 0 \\ 7 & 0 & 0 & 8 \end{bmatrix}$$

La représentation *CSR* serait :
— `values = [2, 7, 2, 5, 3, 7, 8]`
— `columns = [0, 3, 1, 2, 2, 0, 3]`
— `row_ptr = [0, 2, 4, 5, 7]`

Le gain au niveau de l'espace mémoire requis apparait ici assez clairement, surtout si l'on prend en compte le fait que la dimension des matrices étudiées sera bien plus importante.

#### CSC (Compressed Sparse Column)

Le format *CSC* est similaire au format *CSR*, mais avec les colonnes au lieu des lignes comme unité de stockage. Il est plus efficace lorsque l'on souhaite accéder aux éléments d'une colonne plus directement. Le format *CSC* contient également trois tableaux :

— **Valeurs** : un tableau des éléments non nuls, stockés colonne par colonne.
— **Lignes** : un tableau contenant les indices de lignes des éléments non nuls.
— **Index de colonne** : un tableau indiquant où chaque colonne commence dans les tableaux de valeurs et de lignes.

Comme pour le stockage *CSR*, la représentation *CSC* est une représentation compacte qui permet un accès efficace aux éléments non nuls d'une matrice en utilisant les indices de ligne et de colonne.

#### Dans le contexte du TP

À présent, nous allons réimplémenter ces méthodes (LU, Richardson, Jacobi, Gauss-Seidel) dans le cas de matrices stockées en *CSR* et *CSC*.

L'objectif sera de pouvoir, à terme, également appliquer ces méthodes efficacement sur des matrices plus grandes et sans structure définie, tout en maintenant des performances optimales.