# Rapport TP Calcul Numérique
# Résolution de l'équation de Poisson 1D

DIALLO Ibrahima

## 1 Introduction

Dans ce TP, il s'agit de résoudre l'équation de la chaleur en une dimension. Cette équation modélise par exemple la température dans une barre métallique. L'équation s'écrit :

$$-k\frac{\partial^2 T}{\partial x^2} = g$$

avec $T(0) = T_0$ et $T(1) = T_1$ comme conditions aux bords. Dans le cas étudié, on prend $g = 0$, donc il n'y a pas de source de chaleur.

La solution exacte est simple : $T(x) = T_0 + x(T_1 - T_0)$, une droite entre $T_0$ et $T_1$.

Pour résoudre numériquement, le domaine est discrétisé avec des points régulièrement espacés. En utilisant les différences finies, on obtient un système linéaire $Au = f$ où $A$ est une matrice tridiagonale.

## 2 Présentation des méthodes

### 2.1 Exercice 5 : Utilisation de LAPACK

LAPACK est une bibliothèque numérique très utilisée. Deux approches ont été testées :

— `dgbtrf + dgbtrs` : D'abord la matrice est factorisée (décomposition LU), puis le système est résolu.

— `dgbsv` : Une fonction qui réalise les deux opérations en une seule fois (factorisation et résolution).

Ces méthodes sont génériques et fonctionnent pour différentes tailles de matrices.

### 2.2 Exercice 6 : Algorithme LU tridiagonal

Comme la matrice est toujours tridiagonale (seulement 3 diagonales non nulles), un algorithme spécialisé a été implémenté. L'idée est simple : au lieu d'utiliser une bibliothèque générale qui effectue de nombreuses vérifications, seuls les calculs nécessaires sont réalisés.

### 2.3 Un choix important : la vérification des pivots

Dans l'algorithme implémenté, pour vérifier si un pivot est nul (ce qui empêcherait la factorisation), la comparaison exacte avec zéro (`pivot == 0.0`) n'est pas utilisée, mais plutôt un test avec une tolérance :

```
if (fabs(pivot) < 1e-12) {
    *info = i + 1;
    return *info;
}
```

En calcul numérique, les erreurs d'arrondi font que certaines opérations théoriquement égales à zéro peuvent produire des résultats très petits mais non nuls.

Si on testait `pivot == 0.0`, on pourrait :

— Rater un pivot très petit mais pas exactement zéro

— Diviser par un nombre très petit, ce qui donnerait un très grand nombre et amplifierait les erreurs

Avec le test `fabs(pivot) < 1e-12`, on détecte à la fois :

— Les pivots exactement nuls

— Les pivots très proches de zéro

# 3 Résultats et validation

## 3.1 Exactitude des solutions

La première vérification concerne la correction des solutions. La solution numérique a été comparée avec la solution exacte. L'erreur relative obtenue est :

$$\text{relres} = 2.69 \times 10^{-16}$$

Cette valeur est très petite (presque la précision machine), ce qui montre que toutes les méthodes donnent une solution correcte.

## 3.2 Performances comparées

Les temps d'exécution ont été mesurés pour différentes tailles de problèmes. Voici les résultats :

| Taille $n$ | LAPACK (s) | Algorithme spécialisé (s) | Gain |
|---|---|---|---|
| 100 | $9.64 \times 10^{-6}$ | $6.42 \times 10^{-6}$ | +50% |
| 1 000 | $6.16 \times 10^{-5}$ | $5.71 \times 10^{-5}$ | +8% |
| 10 000 | $4.12 \times 10^{-4}$ | $3.86 \times 10^{-4}$ | +7% |
| 100 000 | $3.87 \times 10^{-3}$ | $3.59 \times 10^{-3}$ | +8% |

TABLE 1 – Temps d'exécution moyens

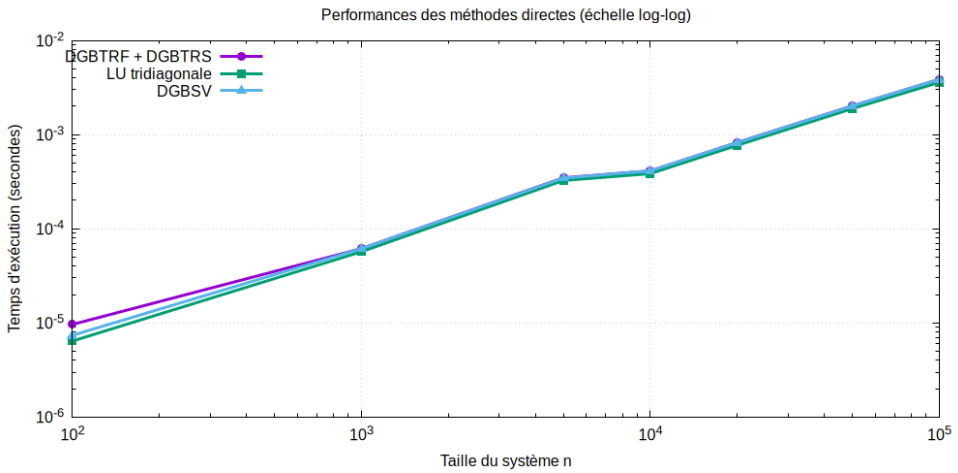


FIGURE 1 – Comparaison des temps d'exécution

# 4 Analyse des résultats

## 4.1 Pourquoi l'algorithme spécialisé est plus rapide ?

### 4.1.1 Pour les petites matrices ($n = 100$)

Le gain est important (+50%) parce que LAPACK effectue beaucoup d'opérations inutiles pour une petite matrice :

— Vérification des arguments
— Allocation de mémoire temporaire

L'algorithme LU tridiagonal va droit au but : il effectue uniquement les calculs nécessaires.

### 4.1.2 Pour les grandes matrices

Le gain est plus petit (7-8%) mais toujours présent. Même pour 100 000 inconnues, l'algorithme spécialisé reste plus rapide. C'est parce qu'il effectue moins d'opérations et accède à la mémoire de façon plus efficace.

## 4.2 Comparaison entre méthodes LAPACK

Les deux méthodes LAPACK donnent exactement les mêmes résultats. `dgbsv` est légèrement plus pratique car une seule fonction est nécessaire au lieu de deux. Pour les grandes matrices, les temps sont presque identiques.

## 4.3 Complexité algorithmique

On observe que quand la taille est multipliée par 10, le temps est à peu près multiplié par 10 aussi. C'est normal car pour une matrice tridiagonale, le nombre d'opérations est proportionnel à $n$. On dit que la complexité est en $O(n)$.

# 5 Conclusion

## 5.1 Exercice 5

La fonction `dgbsv` de LAPACK est une bonne solution générale. Elle est facile à utiliser et donne des résultats précis.

## 5.2 Exercice 6

L'algorithme spécialisé pour les matrices tridiagonales montre qu'on peut faire mieux qu'une bibliothèque générale :

— Il est plus rapide, surtout pour les petits problèmes
— Il reste efficace pour les grands problèmes
— Il est plus robuste numériquement (vérification des pivots avec tolérance)
— Il est plus simple à comprendre et à modifier

## 5.3 Conclusion générale

Ce TP montre l'importance de choisir la bonne méthode selon le problème. Pour des matrices avec une structure simple (comme les tridiagonales), un algorithme spécialisé peut être meilleur qu'une bibliothèque générale.

Il a également été montré l'importance des considérations numériques : vérifier l'égalité exacte avec zéro en calcul numérique est souvent une mauvaise approche. Il vaut mieux utiliser une tolérance comme avec `1e-12`.

## Annexe : Code de l'algorithme

Voici le code complet de la fonction `dgbtrftridiag` :

```
int dgbtrftridiag(int *la, int *n, int *kl, int *ku,
                  double *AB, int *lab, int *ipiv, int *info) {
    int i;
    double pivot, mult;

    *info = 0;

    // Initialisation des pivots
    for (i = 0; i < *n; i++) {
        ipiv[i] = i + 1;
    }

    // Factorisation LU
    for (i = 0; i < *n - 1; i++) {
        // Récupération du pivot (élément diagonal)
        pivot = AB[2 + i * (*lab)];

        // Vérification robuste du pivot
        // Une tolérance (1e-12) est utilisée au lieu de == 0.0
        // car en calcul numérique, un pivot peut être très petit
        // sans être exactement zéro à cause des erreurs d'arrondi
        if (fabs(pivot) < 1e-12) {
            *info = i + 1;  // Échec à l'étape i+1
            return *info;
        }

        // Calcul du multiplicateur
        mult = AB[3 + i * (*lab)] / pivot;

        // Stockage du multiplicateur (devient L[i])
        AB[3 + i * (*lab)] = mult;

        // Mise à jour de la diagonale suivante
        AB[2 + (i + 1) * (*lab)] -= mult * AB[1 + (i + 1) * (*lab)];
    }

    // Vérification du dernier pivot
    if (fabs(AB[2 + (*n - 1) * (*lab)]) < 1e-12) {
        *info = *n;
    }

    return *info;
}
```

**Explication des choix :**

— Ligne 19 : `fabs(pivot) < 1e-12` est plus sûr que `pivot == 0.0`

— Ligne 20 : L'indice de l'étape qui a échoué est retourné

— Ligne 26 : Le multiplicateur est stocké à la place de la sous-diagonale

— Ligne 29 : La diagonale suivante est mise à jour