Université de Perpignan Via Domitia
Master 1 CHPS

Année universitaire 2024-2025
Semestre 2

Rapport de stage

Analyse numérique d'équations aux dérivées partielles par différences finies et implémentation optimisée pour le calcul haute performance

par GAILLOT Jean-Baptiste

Table des matières

Informations et introduction

Pour chaque problème, l'approche sera la suivante :

- Partie mathématiques :
  - concevoir un schéma numérique pour obtenir une solution approchée du problème,
  - s'assurer de l'existence et de l'unicité de la solution du schéma,
  - s'assurer des bonnes propriétés du schéma (consistance, convergence, erreur locale, ...),
  - concevoir des schémas de résolution du système linéaire associé à cette méthode,
- Partie informatique :
  - implémenter la résolution du problème en langage C de différentes manières (séquentiel, OpenMP, MPI, bibliothèque) dans le but de comparer les performances des méthodes,

Bibliographie et supports

— *Rappels de calcul scientifique.* (2008) par Patrick Ciarlet
— *Finite-Difference Approximations to the Heat Equation* (2004) par Gerald W. Recktenwald
— *Numerical Methods for Ordinary Differential Equations* par Habib Ammari et Konstantinos Alexopoulos
— *Lecture 6: Finite difference methods* par Habib Ammari
— Cours de calcul numérique (M1 CHPS) par Serge Dumont
— Cours d'analyse et calcul numérique (L3 Maths) par Francesco Bonaldi
— Cours d'algorithmique et programmation parallèle (M1 CHPS) par David Defour
— Forums d'aides

Organisation du projet en langage C

Lien vers le GitHub du projet : `https://github.com/gaillot18/Stage-EDP.git`

Structure du projet

- **Équation de Poisson en dimension 1** : `Probleme-1D`
  - Version 0 (`base`) : Base
  - Version 1 (`sequentiel-1`) : Méthode itérative, séquentiel
  - Version 2 (`parallele-1`) : Méthode itérative, parallèle OpenMP
  - Version 3 (`parallele-2`) : Méthode itérative, parallèle MPI
  - Version 4 (`sequentiel-2`) : Méthode directe, séquentiel
- **Équation de Poisson en dimension 2** : `Probleme-2D`
  - Version 0 (`base`) : Base
  - Version 1 (`sequentiel-1`) : Méthode itérative, séquentiel
  - Version 2 (`parallele-1`) : Méthode itérative, parallèle OpenMP
  - Version 3 (`parallele-2`) : Méthode itérative, parallèle MPI bloquant
  - Version 4 (`parallele-3`) : Méthode itérative, parallèle MPI non bloquant
  - Version 5 (`sequentiel-2`) : Méthode directe, séquentiel
  - Version 6 (`sequentiel-3`) : Méthode directive, séquentiel, bibliothèque `cholmod`
- **Équation des ondes en dimension 1** : `Probleme-Ondes`
  - Version 1 (`sequentiel-1`) : Méthode itérative, séquentiel
- **Équation de la chaleur en dimension 2** : `Probleme-Chaleur`
  - Version 1 (`sequentiel-1`) : Schéma explicite, séquentiel
  - Version 2 (`parallele-1`) : Schéma explicite, parallèle OpenMP
  - Version 3 (`parallele-2`) : Schéma explicite, parallèle MPI
  - Version 4 (`sequentiel-2`) : Schéma implicite, séquentiel, bibliothèque `cholmod`

Informations

- Le dossier `Fonctions-communes` contient des fichiers de fonctions qui seront appelées pour chaque problèmes (affichages, opérations sur des tableaux, sauvegardes de résultats dans un fichier, ...)
- Pour chaque problème, il y a les dossiers `Sources`, `Objets`, `Librairies` (qui contient les déclarations de fonctions), `Binaires` et `Textes` (qui contient des résultats). A l'intérieur de chaque dossier, il y a des sous-dossiers pour chaque version du problème.
- Pour chaque problème, il y a un Makefile incluant les règles nécessaires pour compiler et/ou nettoyer chaque version et un script pour exécuter chaque version pour les valeurs de pas voulues.
- Pour chaque version d'un problème, il y a les fichiers suivants :
  - `main.c` : programme principal
  - `resolution.c` : fonctions de résolution
  - `parallele.c` : fonctions pour préparer les données MPI (informations sur les noeuds à traiter, données topologie cartésienne, échange des halos) (uniquement pour les versions MPI)
- Certaines fonctions qui sont appelées de nombreuses fois sont mises inline.
- Les flags de compilations utilisés à chaque fois sont `-O3` et `-Wall`.
- Les résultats qui seront présentés ont été exécutés sur une machine ordinaire (8 CPU, 16 Go de mémoire) et non un cluster de calcul.

1 Équation de Poisson en dimension 1

1.1 Analyse numérique

1.1.1 Présentation du problème

Soit $f :]0,1[\to \mathbb{R}$ continue. Soit le problème suivant :
Trouver u de classe C^4 telle que :

$$\begin{cases} -u''(x) = f(x) & \forall\, x \in]0,1[\\ u(0) = 0,\ u(1) = 0 \end{cases}.$$

Un exemple de solution connue est si $f \equiv 1$, alors $u(x) = \frac{1}{2}x(1-x)$, ou si $f(x) = \pi^2 \sin(\pi x)$, alors $u(x) = \sin(\pi x)$.

1.1.2 Schéma numérique

Soient $x, h \in]0,1[$ tels que $[x-h, x+h] \subset [0,1]$. On utilise la formule de Taylor à l'ordre 3 :

$$\exists\, \theta_+ \in]0,1[: u(x+h) = u(x) + hu'(x) + \frac{1}{2}h^2u''(x) + \frac{1}{6}h^3u^{(3)}(x) + \frac{1}{24}h^4u^{(4)}(x+\theta_+h),$$

$$\exists\, \theta_- \in]-1,0[: u(x-h) = u(x) - hu'(x) + \frac{1}{2}h^2u''(x) - \frac{1}{6}h^3u^{(3)}(x) + \frac{1}{24}h^4u^{(4)}(x+\theta_-h).$$

En additionnant, on obtient :

$$\boxed{u(x+h) + u(x-h) = 2u(x) + h^2u''(x) + \frac{1}{24}h^4\left(u^{(4)}(x+\theta_+h) + u^{(4)}(x+\theta_-h)\right)}. \tag{1.1}$$

D'après le théorème des valeurs intermédiaires, on a :

$$\exists\, \theta \in]x+\theta_-h, x+\theta_+h[: u^{(4)}(\theta) = \frac{1}{2}\left(u^{(4)}(x+\theta_+h) + u^{(4)}(x+\theta_-h)\right)$$

ce qui implique que

$$\exists\, \theta \in]-1,1[: u^{(4)}(x+\theta h) = \frac{1}{2}\left(u^{(4)}(x+\theta_+h) + u^{(4)}(x+\theta_-h)\right)$$
$$\Leftrightarrow\ \exists\, \theta \in]-1,1[: 2u^{(4)}(x+\theta h) = u^{(4)}(x+\theta_+h) + u^{(4)}(x+\theta_-h).$$

En injectant dans (1.1), on obtient :

$$u(x+h) + u(x-h) = 2u(x) + h^2u''(x) + \frac{1}{12}h^4u^{(4)}(x+\theta h)$$
$$\Leftrightarrow -h^2u''(x) = 2u(x) - u(x+h) - u(x-h) + \frac{1}{12}h^4u^{(4)}(x+\theta h)$$
$$\Leftrightarrow \boxed{-u''(x) = \frac{1}{h^2}(-u(x+h) + 2u(x) - u(x-h)) + E_h}. \tag{1.2}$$

avec

$$E_h := \frac{1}{12}h^2u^{(4)}(x+\theta h).$$

On peut utiliser l'égalité (1.2) sans l'erreur de troncature E_h et poser $x_i := ih$ pour $i \in \{0, \ldots, N\}$ (on utilise $N+1$ noeuds avec $h = 1/N$), $u_i :\approx u(x_i)$ et $f_i := f(x_i)$ pour obtenir le schéma numérique suivant :

$$\boxed{\frac{1}{h^2}(-u_{i+1} + 2u_i - u_{i-1}) = f_i}. \tag{1.3}$$

Soient $u := \begin{pmatrix} u_1 & \cdots & u_{N-1} \end{pmatrix}^T$ le vecteur de la solution approchée ($u_0 = 0$ et $u_N = 0$ sont connus) et $f := \begin{pmatrix} f_1 & \cdots & f_{N-1} \end{pmatrix}^T$ le vecteur du second membre exact. Alors la forme matricielle du schéma numérique est la suivante (pour $N = 6$) :

$$\boxed{Au = f} \tag{1.4}$$

$$\Leftrightarrow \underbrace{\frac{1}{h^2}\begin{pmatrix} 2 & -1 & \cdot & \cdot & \cdot \\ -1 & 2 & -1 & \cdot & \cdot \\ \cdot & -1 & 2 & -1 & \cdot \\ \cdot & \cdot & -1 & 2 & -1 \\ \cdot & \cdot & \cdot & -1 & 2 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_{N-1} \end{pmatrix}}_{=u} = \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_{N-1} \end{pmatrix}}_{=f}.$$

Remarques

— La valeur en un point du maillage dépend de valeurs d'au plus 3 points du maillage.

— A est une matrice creuse : elle comporte 3 diagonales (centrales).

1.1.3 Existence et unicité de la solution approchée

Proposition A est définie-positive et $Au = f$ admet une unique solution.

Démonstration Montrons que A est définie-positive :
Soit $x \in \mathbb{R}^N$, alors :

$$x^T A x = \begin{pmatrix} x_1 & \cdots & x_{N-1} \end{pmatrix} A \begin{pmatrix} x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} = \frac{1}{h^2} \begin{pmatrix} x_1 & \cdots & x_{N-1} \end{pmatrix} \begin{pmatrix} 2x_1 - x_2 \\ \vdots \\ -x_{N-2} + 2x_{N-1} \end{pmatrix}$$

$$= \frac{1}{h^2} \left(2x_1^2 - x_1x_2 - x_2x_1 + 2x_2^2 - x_2x_3 + ... - x_{N-2}x_{N-3} + 2x_{N-2}^2 - x_{N-2}x_{N-1} - x_{N-1}x_{N-2} + 2x_{N-1}^2\right)$$

$$= \frac{1}{h^2}\left(2\sum_{i=1}^{N-1} x_i^2 - 2\sum_{i=1}^{N-2} x_i x_{i+1}\right) = \frac{1}{h^2}\left(\sum_{i=1}^{N-2} x_i^2 + 2x_{N-1}^2 - 2\sum_{i=1}^{N-2} x_i x_{i+1}\right)$$

$$= \frac{1}{h^2}\left(\sum_{i=1}^{N-2} x_i^2 + \sum_{i=0}^{N-3} x_{i+1}^2 + 2x_{N-1}^2 - 2\sum_{i=1}^{N-2} x_i x_{i+1}\right)$$

$$= \frac{1}{h^2}\left(\sum_{i=1}^{N-2} x_i^2 + \sum_{i=1}^{N-2} x_{i+1}^2 - 2\sum_{i=1}^{N-2} x_i x_{i+1} + x_1^2 x_{N-1}^2\right) = \frac{1}{h^2}\left(\sum_{i=1}^{N-2} (x_i - x_{i+1})^2 + x_1^2 + x_{N-1}^2\right) \geq 0.$$

Si $\sum_{i=1}^{N-2} (x_i - x_{i+1})^2 + x_1^2 + x_{N-1}^2 = 0$, alors $x = 0_{\mathbb{R}^N}$.
Donc A est définie-positive, donc A est inversible et $Au = f$ admet une unique solution.

Remarque A est symétrique et définie-positive donc l'utilisation de la méthode de Cholesky est possible.

1.1.4 Consistance du schéma et majoration de l'erreur de troncature

Proposition Le schéma (1.4) est consistant : $\lim_{h\to 0} |E_h| = 0$ et $|E_h| \leq \frac{1}{12} h^2 \sup_{x\in[0,1]} |f''(x)|$.

Démonstration Comme on a $x + \theta h \in [0,1]$ et $-u^{(4)} = -f''$, alors on peut majorer l'erreur de troncature comme ceci :

$$|u^{(4)}(x + \theta h)| = |f''(x + \theta h)| \leq \sup_{x\in[0,1]} |f''(x)|,$$

on obtient :

$$\left|\frac{1}{12} h^2 u^{(4)}(x + \theta h)\right| \leq \frac{1}{12} h^2 \sup_{x\in[0,1]} |f''(x)| \underset{h\to 0}{\longrightarrow} 0.$$

Remarque Le schéma (1.4) est d'ordre 2 pour x.

1.1.5 Convergence du schéma et majoration de l'erreur locale

Proposition *(admise)* $\forall\, i,j \in \{0,\dots,N\} : a_{i,j}^{-1} \geq 0$.

Proposition Soit $e := (u_i - u(x_i))_{0\leq i\leq N}$. Alors, le schéma (1.4) est convergent : $\lim\limits_{h\to 0}\|e\|_\infty = 0$ et

$$\|e\|_\infty \leq \frac{1}{96}h^2 \sup_{x\in[0,1]} |f''(x)|.$$

Démonstration
Soit $i \in \{1,\dots,N-1\}$. Alors,

$$Ae = \frac{1}{h^2}\begin{pmatrix} 2e_1 - e_2 \\ -e_1 + 2e_2 - e_3 \\ \vdots \\ -e_{N-3} + 2e_{N-2} - e_{N-1} \\ -e_{N-2} + 2e_{N-1} \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} -\underbrace{e_0}_{=0} + 2e_1 - e_2 \\ -e_1 + 2e_2 - e_3 \\ \vdots \\ -e_{N-3} + 2e_{N-2} - e_{N-1} \\ -e_{N-2} + 2e_{N-1} - \underbrace{e_N}_{=0} \end{pmatrix}$$

$$\Leftrightarrow (Ae)_i = \frac{1}{h^2}(-e_{i-1} + 2e_i - e_{i+1}) = \frac{1}{h^2}(-u_{i-1} + 2u_i - u_{i+1}) + \frac{1}{h^2}(u(x_{i-1}) - 2u(x_i) - u(x_{i+1}))$$

$$= (Au)_i + \frac{1}{h^2}(u(x_{i-1}) - 2u(x_i) + u(x_{i+1})) = f(x_i) + \frac{1}{h^2}(u(x_{i-1}) - 2u(x_i) + u(x_{i+1}))$$

$$= \underbrace{f(x_i) + u''(x_i)}_{=0} + \frac{1}{12}h^2 u^{(4)}(x_i + \theta_i h) = \frac{1}{12}h^2 u^{(4)}(x_i + \theta_i h) = \frac{1}{12}h^2 f''(x_i + \theta_i h)$$

$$\Rightarrow |(Ae)_i| \leq \frac{1}{12}h^2 \sup_{x\in[0,1]} |f''(x)|$$

$$\Rightarrow \|Ae\|_\infty = \max_{0\leq i\leq N}\left|\frac{1}{12}h^2 \sup_{x\in[0,1]} |f''(x)|\right| = \frac{1}{12}h^2 \sup_{x\in[0,1]} |f''(x)|$$

Soit $\varepsilon := Ae$ ($A^{-1}\varepsilon = e$). Alors,

$$|e_i| = |(A^{-1}\varepsilon)_i| = \left|\sum_{j=0}^{N} a_{i,j}^{-1}\varepsilon_j\right| \leq \sum_{j=0}^{N}|a_{i,j}^{-1}||\varepsilon_j| \leq \frac{1}{12}h^2 \sup_{x\in[0,1]} |f''(x)| \sum_{j=0}^{N} a_{i,j}^{-1}.$$

Soit $\delta_j := \begin{pmatrix}1 & \cdots & 1\end{pmatrix}^T$. Alors, $\sum\limits_{j=0}^{N} a_{i,j}^{-1} = \sum\limits_{j=0}^{N} a_{i,j}^{-1}\delta_j$. Soit $u_0 := A^{-1}\delta$. Alors, $\delta = Au_0$ donc u_0 est solution de l'équation (1.4) avec $f = \delta$. Dans ce cas, on connait la solution exacte : $u_0(x) = \frac{1}{2}x(1-x)$ et $\sup\limits_{x\in[0,1]} u_0(x) = \frac{1}{8}$ donc $\sup\limits_{x\in[0,1]} (u_0)_i = \frac{1}{8}$ et $\|u_0\|_\infty = \frac{1}{8}$. Donc,

$$\|A^{-1}\delta\|_\infty = \|u_0\|_\infty = \max_{0\leq i\leq N}\sum_{j=0}^{N}|a_{i,j}^{-1}\delta_j| = \max_{0\leq i\leq N}\sum_{j=0}^{N}|a_{i,j}^{-1}| = \frac{1}{8}$$

donc $|e_i| \leq \frac{1}{12}h^2 \sup\limits_{x\in[0,1]} |f''(x)|\frac{1}{8} = \frac{1}{96}h^2 \sup\limits_{x\in[0,1]} |f''(x)|$.

Remarques

— On vérifiera cette propriété à la fin de la sous-sous-section 1.2.4 avec un exemple qui utilise une implémentation utilisant une méthode de résolution directe.

— On a aussi montré que $\|A^{-1}\|_\infty \leq \frac{1}{8}$.

Parfois, la solution u peut être moins régulière. On cherche à assurer la stabilité du schéma si u est de classe C^2.

Proposition Si u est de classe C^2, alors le schéma (1.4) est convergent : $\lim_{h\to 0}\|e\|_\infty = 0$.

Démonstration
Soient $\varepsilon := Ae$, $i \in \{1, \cdots, N-1\}$ et $u_{sol} = \begin{pmatrix} u(x_0) & \cdots & u(x_N) \end{pmatrix}^T$ le vecteur de la solution exacte. Alors,

$$\varepsilon_i = (Ae)_i = (A(u - u_{sol}))_i = (Au - Au_{sol})_i = (Au)_i - (Au_{sol})_i = f(x_i) - (Au_{sol})_i .$$

$$Au_{sol} = \frac{1}{h^2}\begin{pmatrix} -u(x_0) + 2u(x_1) - u(x_2) \\ \vdots \\ -u(x_{N-2}) + 2u(x_{N-1}) - u(x_N) \end{pmatrix} \Rightarrow (Au_{sol})_i = \frac{1}{h^2}\left(-u(x_{i-1}) + 2u(x_i) - u(x_{i+1})\right).$$

On a :

$$u(x_{i-1}) = u(x_i) - hu'(x_i) + \frac{1}{2}h^2u''(x_i + \theta_- h) \quad \text{et} \quad u(x_{i+1}) = u(x_i) + hu'(x_i) + \frac{1}{2}h^2u''(x_i + \theta_+ h).$$

En injectant dans $(Au_{sol})_i$, en simplifiant et en utilisant le théorème des valeurs intermédiaires, on obtient :

$$(Au_{sol})_i = -\frac{1}{2}\left(u''(x_i + \theta_- h) + u''(x_i + \theta_+ h)\right) = \frac{1}{2}\left(f(x_i + \theta_- h) - f(x_i + \theta_+ h)\right) = f(x_i + \theta h)$$

donc

$$\varepsilon_i = f(x_i) - f(x_i + \theta h).$$

f est continue sur $[0,1]$ (compact) donc f est uniformément continue donc :

$$\forall\ \eta > 0,\ \exists\ h_\eta > 0,\ \forall\ x, y \in [0,1] : |x - y| < h_\eta \Rightarrow |f(x) - f(y)| < \eta,$$

en particulier :

$$\forall\ \eta > 0,\ \exists\ h_\eta > 0 : |x_i - (x_i + \theta h)| < h_\eta \Rightarrow |f(x_i) - f(x_i + \theta_h)| < \eta.$$

Soit $\eta > 0$. Alors,

$$\exists\ h_\eta > 0 : |x_i - (x_i + \theta h)| < h_\eta \Rightarrow |f(x_i) - f(x_i + \theta_h)| < \eta$$
$$\Leftrightarrow |\varepsilon_i| < \eta \Rightarrow \|\varepsilon\|_\infty < \eta$$

donc $\|\varepsilon\|_\infty \underset{h\to 0}{\longrightarrow} 0$.

$$\|e\|_\infty = \|A^{-1}\varepsilon\|_\infty \leq \|A^{-1}\|_\infty\|\varepsilon\|_\infty \leq \frac{1}{8}\|\varepsilon\|_\infty \underset{h\to 0}{\longrightarrow} 0$$

donc $\lim_{h\to 0}\|e\|_\infty = 0$.

1.1.6 Méthode de résolution itérative

Soient $D := \operatorname{diag}(A)$, $-E$ la partie triangulaire inférieure stricte de A, $-F$ la partie triangulaire supérieure stricte de A et k l'itération. Alors, le schéma de la méthode Jacobi pour $Au = f$ est le suivant :

$$u^{(k+1)} = D^{-1}(E + F)u^{(k)} + D^{-1}f.$$

Pour ne pas calculer d'inverse, on utilise :

$$Du^{(k+1)} = (E + F)u^{(k)} + f.$$

A est creuse, on peut donc développer ces termes :

$$Du^{(k+1)} = \frac{1}{h^2}\begin{pmatrix} 2 & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & 2 \end{pmatrix}\begin{pmatrix} u_1^{(k+1)} \\ \vdots \\ \vdots \\ u_{N-1}^{(k+1)} \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} 2u_1^{(k+1)} \\ \vdots \\ \vdots \\ 2u_{N-1}^{(k+1)} \end{pmatrix}$$

et

$$(E+F)\,u^{(k)} + f = \frac{1}{h^2}\begin{pmatrix} \cdot & 1 & \cdot & \cdot \\ 1 & \cdot & \ddots & \cdot \\ \cdot & \ddots & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot \end{pmatrix}\begin{pmatrix} u_1^{(k)} \\ \vdots \\ \vdots \\ u_{N-1}^{(k)} \end{pmatrix} + \begin{pmatrix} f_1^{(k)} \\ \vdots \\ \vdots \\ f_{N-1}^{(k)} \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} u_0^{(k)} + u_2^{(k)} \\ \vdots \\ \vdots \\ u_{N-2}^{(k)} + u_N^{(k)} \end{pmatrix} + \begin{pmatrix} f_1^{(k)} \\ \vdots \\ \vdots \\ f_{N-1}^{(k)} \end{pmatrix}.$$

Soit $i \in \{1, \cdots, N-1\}$. Alors, en identifiant chaque terme, on obtient :

$$Du^{(k+1)} = (E+F)\,u^{(k)} + f \Leftrightarrow \frac{2}{h^2}u_i^{(k+1)} = \frac{1}{h^2}\left(u_{i-1}^{(k)} + u_{i+1}^{(k)}\right) + f_i$$

$$\Leftrightarrow \boxed{u_i^{(k+1)} = \frac{1}{2}\left(u_{i-1}^{(k)} + u_{i+1}^{(k)} + h^2 f_i\right)}. \tag{1.5}$$

1.1.7 Méthode de résolution directe

Soit L tel que $A = LL^T$. On rappelle la formule de la factorisation de Cholesky :

$$\ell_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} \ell_{i,k}^2} \quad \text{et} \quad \ell_{i,j} = \frac{1}{\ell_{j,j}}\left(a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k}\right).$$

On calcule d'abord en colonnes puis en lignes.

Proposition Soit A une matrice tridiagonale, symétrique et définie-positive. Alors, la matrice L de la décomposition de Cholesky de A est bidiagonale inférieure.

Démonstration On a : Si $i > j+1$, alors $a_{i,j} = 0$.
On calcule la colonne $j = 1$:

$$\ell_{2,1} = \frac{1}{\ell_{0,1}}\left(\underbrace{a_{1,1}}_{=-1} - \underbrace{\sum_{k=1}^{-1} \ell_{1,k}\ell_{0,k}}_{=0}\right) \neq 0.$$

Si $d > 2$, alors :

$$\ell_{d,1} = \frac{1}{\ell_{0,1}}\left(\underbrace{a_{d,1}}_{=0} - \sum_{k=0}^{-1} \underbrace{\ell_{d,k}}_{=0}\,\ell_{0,k}\right) = 0$$

donc $L|_{j=1} = \begin{pmatrix} \ell_{0,1} & \ell_{1,1} & 0 & \cdots & 0 \end{pmatrix}^T$.
On calcule la colonne $j = 2$: Si $d > 3$, alors

$$\ell_{d,2} = \frac{1}{\ell_{1,2}}\left(\underbrace{a_{d,2}}_{=0} - \sum_{k=0}^{0} \underbrace{\ell_{d,k}}_{=0}\,\ell_{0,k}\right) = 0$$

donc $L|_{j=2} = \begin{pmatrix} 0 & \ell_{1,2} & \ell_{2,2} & 0 & \cdots & 0 \end{pmatrix}^T$.
Ainsi de suite (pour les colonnes suivantes, il y a $a_{d,j} = 0\ \forall\ d > j+1$ donc $\ell_{d,k} = 0\ \forall\ k \in \{1, \ldots, j-1\}$).

A est symétrique, définie-positive et tridiagonale donc $\exists\, L : LL^T = A$ avec :

$$L = \begin{pmatrix} \ell_{1,1} & \cdot & \cdot & \cdot \\ \ell_{2,1} & \ddots & \cdot & \cdot \\ \cdot & \ddots & \ddots & \cdot \\ \cdot & \cdot & \ell_{N-1,N-2} & \ell_{N-1,N-1} \end{pmatrix}.$$

On peut résoudre $Au = f$ en résolvant $Ly = f$ puis $L^T u = y$ avec :

$$y_1 = \frac{f_1}{\ell_{1,1}} \quad \text{et} \quad \text{pour } i \text{ de 2 à } N-1 : y_i = \frac{f_i - \ell_{i,j-1} y_{i-1}}{\ell_{i,i}}$$

et

$$\boxed{u_{N-1} = \frac{y_{N-1}}{\ell_{N-1,N-1}} \quad \text{et} \quad \text{pour } i \text{ de } N-2 \text{ à } 1 : u_i = \frac{y_i - \ell_{i,j+1} u_{i+1}}{\ell_{i,i}}}. \tag{1.6}$$

1.2 Implémentation

Pour la suite, la fonction à approcher sera $f(x) := \pi^2 \sin(\pi x)$ et $\varepsilon := 1 \cdot 10^{-10}$.

1.2.1 Version de base

Implémentation *(version 0 résumée, voir les détails dans `/Probleme-1D/base`)*
Pour commencer, on implémente la version de base non optimisée de la résolution.
Étapes :

- créer des fonctions qui font le travail :
  - construire la matrice `A` (voir la fonction `construire_matrice`),
  - calculer le second membre `f`,
  - calculer la solution approchée `u` (voir la fonction `resoudre_gauss`).

Commentaires

- Pour calculer `u`, on résout le système linéaire avec la méthode de Gauss.
- On note ces résultats :

N	5	10	50	100	300	500	1500
$\|e\|_\infty$	0.031916	0.008265	0.000329	0.000082	0.000009	0.000003	<0.000001
Temps d'exécution (s)	<0.01	<0.01	<0.01	<0.01	0.01	0.05	1.00

- A est de taille $O(N)$ et la méthode de Gauss est $O(N^3)$ donc la complexité algorithmique est $O(N^3)$.

1.2.2 Version avec méthode de résolution itérative en séquentiel

Implémentation *(version 1 résumée, voir les détails dans `/Probleme-1D/sequentiel-1`)*
Ensuite, on implémente une version séquentielle du schéma (1.5).
Étapes :

- créer des fonctions qui font le travail :
  - calculer le second membre `f`,
  - calculer la solution approchée `u` :

```
void calculer_u_jacobi(double *f, double *u){

    h_carre = 1.0 / (N * N);
    int nb_iteration_max = INT_MAX;
    double norme = DBL_MAX;
    double *u_anc; double *permut;
```

```
    init_u_anc(&u_anc);

    for (int iteration = 0 ; iteration < nb_iteration_max && norme > 1e-10 ;
        iteration ++){

        for (int j = 1 ; j < nb_pt - 1 ; j ++){
            for (int i = 1 ; i < nb_pt - 1 ; i ++){
                u[IDX(i, j)] = schema(f, u, u_anc, i, j);
            }
        }

        norme = norme_infty_iteration(u, u_anc);

        permut = u; u = u_anc; u_anc = permut; nb_iteration ++;

    }

    terminaison(&permut, &u, &u_anc);

}
```

Fonction qui applique le schéma à un point :

```
static inline __attribute__((always_inline))
double schema(double *f, double *u_anc, int i){

    double res = 0.5 * ((u_anc[i - 1] + u_anc[i + 1]) + h_carre * f[i]);

    return res;

}
```

Fonction pour calculer la norme infinie relative :

```
static inline __attribute__((always_inline))
double norme_infty_iteration(double *u, double *u_anc){

    double norme_nume = 0.0;
    double norme_deno = 0.0;
    double norme;

    for (int i = 0 ; i < nb_pt * nb_pt ; i ++){
        double diff = fabs(u[i] - u_anc[i]);
        if (diff > norme_nume){
            norme_nume = diff;
        }
        if (fabs(u_anc[i]) > norme_deno){
            norme_deno = fabs(u_anc[i]);
        }
    }

    norme = norme_nume / norme_deno;

    return norme;

}
```

Fonction pour terminer :

```
void terminaison(double **permut, double **u, double **u_anc){

    if (nb_iteration % 2 != 0){
```

```
            *permut = *u; *u = *u_anc; *u_anc = *permut;
        }

        free(*u_anc);

}
```

Commentaires

- A la fin d'une itération, on calcule la norme infinie avec la fonction `norme_infty_iteration`.
- Pour éviter la copie du tableau `u` dans `u_div` à la fin de chaque itération, on permute les pointeurs. A la fin, selon la parité du nombre d'itérations, on libère l'allocation du bon pointeur avec la fonction `terminaison`.
- Une version avec la méthode de Gauss-Seidel a été testée. Le schéma est : $u_i^{(k+1)} = \frac{1}{2}\left(u_{i-1}^{(k+1)} + u_{i+1}^{(k)} + h^2 f_i\right)$.
- On note ces résultats :

N	5	10	50	100	300	500	1500
Nombre d'itérations avant arrêt	102	400	8506	31227	241002	617699	4557543
$\|e\|_\infty$	0.031916	0.008265	0.000329	0.000082	0.000007	0.000002	0.000045
Temps d'exécution (s)	<0.01	<0.01	<0.01	0.01	0.14	0.54	11.83

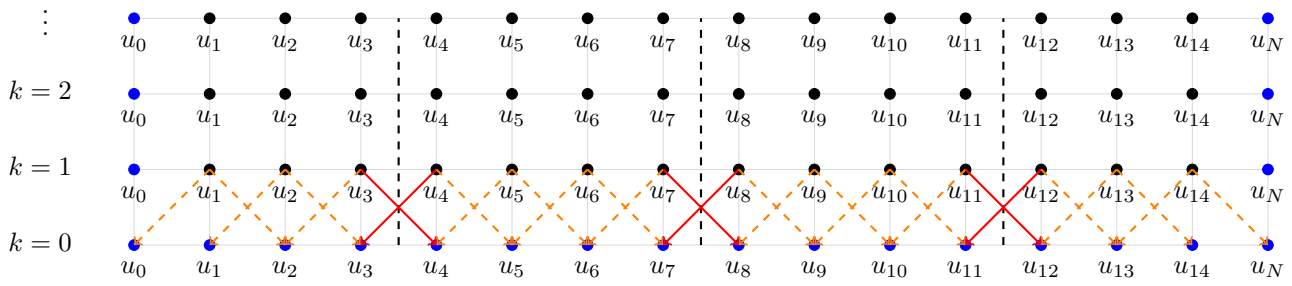
1.2.3 Versions avec méthode de résolution itérative en parallèle avec OpenMP et MPI

Implémentation *(version 2 résumée, voir les détails dans `/Probleme-1D/parallele-1`)*
Ensuite, on implémente une version parallèle du schéma (1.5) avec OpenMP.

Commentaire Cette implémentation reprend exactement le même code que pour `/Probleme-1D/sequentiel-1`) en ajoutant une directive `for` dans la boucle de la fonction `calculer_u_jacobi` et une directive `for` dans la boucle du calcul de la norme relative.

Implémentation *(version 3 résumée, voir les détails dans `/Probleme-1D/parallele-2`)*
Ensuite, on implémente une version parallèle du schéma (1.5) avec MPI. On décompose le domaine discrétisé en part égales (à une cellule près en fonction de la divisibilité).

Illustration Schéma des dépendances pour 4 processus et $N = 15$:



- Un noeud en bleu représente un noeud connu (le vecteur de départ pour $k = 0$ et : ($u_0 = 0$ et $u_N = 0$)). Un noeud en noir représente un noeud à calculer.
- Une ligne noire pointillée représente la séparation entre deux processus.
- Une flèche représente la dépendance entre un noeud et les noeuds sur lesquels ils pointent. Si la flèche est orange pointillée, alors la dépendance fait intervenir deux noeuds sur le même processus. Sinon, si la flèche est rouge, alors la dépendance fait intervenir deux noeuds sur un processus différent.

Étapes :

— créer une topologie 1D non courbée :

```
void creer_topologie(){

    int tore = 0;
    dims = 0;

    MPI_Dims_create(nb_cpu, 1, &dims);

    MPI_Cart_create(MPI_COMM_WORLD, 1, &dims, &tore, 0, &comm_1D);

    MPI_Barrier(comm_1D);

}
```

— créer des fonctions qui, pour chaque processus, fait connaître sa position dans la topologie en déterminant :
  — le nombre de points à traiter `nb_div` (dans le cas où $N+1$ n'est pas divisible par le nombre de processus),
  — les noeuds de départ et d'arrivé `i_debut` et `i_fin`,
  — les voisins à gauche et en bas `voisins`,
  — les tableaux `deplacements` et `nb_elements_recus` pour effectuer le regroupement final sur le rang 0 :

```
void infos_topologie(){

    MPI_Cart_coords(comm_1D, rang, 1, &coords);

    MPI_Cart_shift(comm_1D, 0, 1, &(voisins[0]), &(voisins[1]));

    bord = 2;
    for (int i = 0 ; i < 2 ; i ++){
        if (voisins[i] == -1){
            bord --;
        }
    }

    MPI_Barrier(comm_1D);

}
```

```
void infos_processus(){

    i_debut = (coords * nb_pt) / dims;
    i_fin = ((coords + 1) * nb_pt) / dims - 1;
    nb_pt_div = i_fin - i_debut + 1;

    MPI_Barrier(comm_1D);

}
```

  — créer des fonctions qui font le travail partagé et les communications sur chaque processus :
    — calculer le second membre en parallèle `f_div`,
    — calculer la solution approchée en parallèle `u_div` :
    Fonction principale :

```
void calculer_u_jacobi(double *f_div, double *u_div){

    nb_iteration = 0;
    h_carre = 1.0 / pow(N, 2);
    int nb_iteration_max = INT_MAX;
    double norme = DBL_MAX;
```

```
    int i_boucle_debut; int i_boucle_fin;
    double *u_div_anc; double *permut;

    // Vecteur de depart
    init_u_div_anc(&u_div_anc);
    for (int i = 0 ; i < nb_pt_div + 2 ; i ++){
        u_div[i] = 0.0;
    }

    // Bornes des boucles
    infos_bornes_boucles(&i_boucle_debut, &i_boucle_fin);

    // Iterations
    for (int iteration = 0 ; iteration < nb_iteration_max && norme > 1e-10
         ; iteration ++){

        // Communication
        echanger_halos(u_div_anc);

        for (int i = i_boucle_debut ; i < i_boucle_fin ; i ++){
            u_div[i] = schema(f_div, u_div, u_div_anc, i);
        }

        // Test d'arret
        norme = norme_infty_iteration(u_div, u_div_anc);

        permut = u_div; u_div = u_div_anc; u_div_anc = permut;
            nb_iteration ++;

    }

    terminaison(&permut, &u_div, &u_div_anc);

}
```

Fonction pour obtenir les indices de départ et d'arrivé de la boucle principale du schéma (adaptés pour itérer sur les bords locaux qui ne sont pas globaux) :

```
void infos_bornes_boucles(int *i_boucle_debut, int *i_boucle_fin){

    *i_boucle_debut = 1;
    *i_boucle_fin = nb_pt_div + 1;

    if (i_debut == 0){
        (*i_boucle_debut) ++;
    }
    if (i_fin == nb_pt - 1){
        (*i_boucle_fin) --;
    }

}
```

Fonction pour échanger les halos :

```
void echanger_halos(double *u_div){

    // Envoi gauche, reception droite
    MPI_Sendrecv(&(u_div[1]), 1, MPI_DOUBLE, voisins[0], etiquette, &(
        u_div[nb_pt_div + 1]), 1, MPI_DOUBLE, voisins[1], etiquette,
        comm_1D, &statut);

    // Envoi droite, reception gauche
```

```
    MPI_Sendrecv(&(u_div[nb_pt_div]), 1, MPI_DOUBLE, voisins[1], etiquette
        , &(u_div[0]), 1, MPI_DOUBLE, voisins[0], etiquette, comm_1D, &
        statut);

}
```

— regrouper sur le rang 0 `u_div` et le stocker dans `u`.

Commentaire Les tableaux partiels possèdent 2 cases supplémentaires pour accueillir les halos. Au début de chaque itération, l'échange des halos est effectué dans la fonction `echanger_halos`.

1.2.4 Version avec méthode de résolution directe en séquentiel

Implémentation *(version 4 résumée, voir les détails dans `/Probleme-1D/sequentiel-2`)*
Ensuite, on implémente une version séquentielle du schéma (1.6).
Étapes :

— créer la structure pour stocker une matrice de la même forme que L :

```
struct mat_2bandes{
    int N;
    double *diag; // taille n
    double *sous_diag; // taille n - 1
};
```

— créer des fonctions pour allouer et libérer la structure :
Fonction pour allouer la structure :

```
void init_mat_2bandes(struct mat_2bandes *A){

    A -> N = N;
    A -> diag = (double *)malloc(idx_max * sizeof(double));
    A -> sous_diag = (double *)malloc((idx_max - 1) * sizeof(double));

}
```

Fonction pour libérer la structure :

```
void liberer_mat_2bandes(struct mat_2bandes *A){

    free(A -> diag);
    free(A -> sous_diag);

}
```

— créer la fonction pour obtenir la décomposition de Cholesky en utilisant la structure :

```
void calculer_cholesky(struct mat_2bandes *L){

    h_carre = 1.0 / pow(N, 2);
    double alpha = 2.0 / h_carre;
    double beta = -1.0 / h_carre;

    (L -> diag)[0] = sqrt(alpha);
    (L -> sous_diag)[0] = beta / (L -> diag)[0];

    for (int i = 1 ; i < idx_max - 1 ; i ++){
        (L -> diag)[i] = sqrt(alpha - pow((L -> sous_diag[i - 1]), 2));
        (L -> sous_diag)[i] = beta / (L -> diag[i]);
    }
```

```
    (L -> diag)[idx_max - 1] = sqrt(alpha - pow((L -> sous_diag[idx_max - 2]), 2))
        ;

}
```

Test (affichage effectué automatiquement en exécutant `Probleme-2D/Binaires/sequentiel-2`) pour avoir un aperçu de la compression :

```
Illustration de la structure mat_2bandes (exemple pour N petit) :

Structure mat2_bandes :
N = 7
diag      =  9.899495   8.573214   8.082904   7.826238   7.668116   7.560864
sous_diag = -4.949747  -5.715476  -6.062178  -6.260990  -6.390097

Matrice reelle correspondante :
  9.899495   0.000000   0.000000   0.000000   0.000000   0.000000
 -4.949747   8.573214   0.000000   0.000000   0.000000   0.000000
  0.000000  -5.715476   8.082904   0.000000   0.000000   0.000000
  0.000000   0.000000  -6.062178   7.826238   0.000000   0.000000
  0.000000   0.000000   0.000000  -6.260990   7.668116   0.000000
  0.000000   0.000000   0.000000   0.000000  -6.390097   7.560864
```

— créer des fonctions pour résoudre le système linéaire en utilisant la structure :
Fonction principale :

```
void resoudre_cholesky(double *f, double *u){

    struct mat_2bandes L;
    double *y = (double *)malloc(idx_max * sizeof(double));

    u[0] = 0; u[nb_pt - 1] = 0;

    init_mat_2bandes(&L);
    calculer_cholesky(&L);

    resoudre_cholesky_descente(&L, &(f[1]), y); // Laisser f[0] pour le bord
    resoudre_cholesky_remontee(&L, y, &(u[1])); // Laisser u[0] pour le bord

    liberer_mat_2bandes(&L);
    free(y);

}
```

Fonction pour résoudre $Ly = f$:

```
void resoudre_cholesky_descente(struct mat_2bandes *L, double *f, double *y){

    y[0] = f[0] / (L -> diag)[0];

    for (int i = 1 ; i < idx_max ; i ++){
        y[i] = (f[i] - (L -> sous_diag)[i - 1] * y[i - 1]) / (L -> diag)[i];
    }

}
```

Fonction pour résoudre $L^T u = y$:

```
void resoudre_cholesky_remontee(struct mat_2bandes *L, double *y, double *u){

    u[idx_max - 1] = y[idx_max - 1] / (L -> diag)[idx_max - 1];
```

```
    for (int i = idx_max - 2 ; i >= 0 ; i --){
        u[i] = (y[i] - (L -> sous_diag)[i] * u[i + 1]) / (L -> diag)[i];
    }

}
```

Commentaires

— Cette méthode est impossible à paralléliser à cause des dépendances.

— A possède $O(N)$ colonne. Pour chaque colonne, il y a $O(1)$ lignes à calculer. Pour chaque case, il y a $O(1)$ opérations. Donc la complexité algorithmique est $O(N)$.

On profite du fait que la méthode de Cholesky donne une solution exacte de $Au = f$ pour vérifier la proposition énoncée en sous-sous-section 1.1.5 avec $f(x) = \pi^2 \sin(\pi x)$ dont on connait la solution exacte. On a $f''(x) = -\pi^4 \sin(\pi x)$, $\sup_{x \in [0,1]} |f''(x)| = \pi^4$ donc d'après la proposition : $\|e\|_\infty \leq \frac{1}{96}\pi^4 h^2$. En exécutant pour différentes valeurs de N pour cette méthode, on compare les erreurs obtenues avec celle maximales théoriques :

N	2	3	4	5	10	100	1000
$\|e\|_\infty$ maximal théorique	0.253669	0.112742	0.063417	0.040587	0.010146	0.000101	0.000001
$\|e\|_\infty$ observé	0.233701	0.083678	0.053029	0.031916	0.008265	0.000082	0.000001

Donc l'inégalité est vérifiée pour cet exemple. On remarque que si $f \equiv 1$, alors cette méthode donne $\|e\|_\infty = 0 \ \forall \ N$, ce qui est attendu car $f'' \equiv 0$.

1.2.5 Comparaison des performances des méthodes

— Les deux méthodes ont l'avantage de ne pas utiliser de matrice pleine.

— La méthode itérative possède une complexité algorithmique quadratique (car on itère un certain nombre de fois un traitement en $O(n)$). Le parallélisme aurait pu donner de bons résultats si le nombre de points est très élevé, mais un nombre de points élevé implique un nombre d'itérations externes élevés.

— La méthode directe possède une complexité algorithmique linéaire et une structure dense. Même si on utilise cette fois-ci une structure stockée, elle reste bonne pour la localité mémoire (cache) car les données sont contigues et si le cache possède au moins 2 voies, alors il peut plus facilement stocker les 2 diagonales en même temps (moins de cache-miss).

— Pour $N = 1000$, la méthode directe, en plus de donner une erreur moins élevée, est 50000 fois plus rapide que la méthode itérative.

2 Équation de Poisson en dimension 2

2.1 Analyse numérique

2.1.1 Présentation du problème

Soit $f :]0,1[\times]0,1[\to \mathbb{R}$ continue Soit $D :=]0,1[\times]0,1[$. Soit le problème suivant :
Trouver u de classe C^4 telle que :

$$\begin{cases} -\Delta u(x,y) = f(x,y) & \forall \ (x,y) \in D \\ u(x,y) = 0 & \forall \ (x,y) \in \partial D \end{cases}.$$

Un exemple de solution connue est si $f(x,y) = \sin(2\pi x)\sin(2\pi y)$, alors $u(x,y) = \frac{1}{8\pi^2}(2\pi x)\sin(2\pi y)$.

2.1.2 Schéma numérique

Soient $x, y, h_x, h_y \in]0,1[$ tels que $[x - h_x, x + h_x]$, $[y - h_y, y + h_y] \subset [0,1]$. On utilise la formule de Taylor à l'ordre 3 :

$$\exists\ \theta_{x_+} \in]0,1[: u(x+h_x,y) = u(x,y) + h_x \frac{\partial u}{\partial x}(x,y) + \frac{1}{2}h_x^2 \frac{\partial^2 u}{\partial x^2}(x,y) + \frac{1}{6}h_x^3 \frac{\partial^3 u}{\partial x^3}(x,y) + \frac{1}{24}h_x^4 \frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_+}h_x,y\right),$$

$$\exists\ \theta_{x_-} \in]-1,0[: u(x-h_x,y) = u(x,y) - h_x \frac{\partial u}{\partial x}(x,y) + \frac{1}{2}h_x^2 \frac{\partial^2 u}{\partial x^2}(x,y) - \frac{1}{6}h_x^3 \frac{\partial^3 u}{\partial x^3}(x,y) + \frac{1}{24}h_x^4 \frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_-}h_x,y\right).$$

En additionnant, on obtient :

$$\Leftrightarrow \boxed{\begin{array}{c} u(x+h_x,y) + u(x-h_x,y) \\ = 2u(x,y) + h_x^2 \frac{\partial^2 u}{\partial x^2}(x,y) + \frac{1}{24}h_x^4 \left(\frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_+}h_x,y\right) + \frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_-}h_x,y\right) \right) \end{array}} \tag{2.1}$$

D'après le théorème des valeurs intermédiaires, on a :

$$\exists\ \theta_x \in]-1,1[: 2\frac{\partial^4 u}{\partial x^4}(\theta_x h_x,y) = \left(\frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_+}h_x,y\right) + \frac{\partial^4 u}{\partial x^4}\left(x+\theta_{x_-}h_x,y\right) \right).$$

En injectant dans (2.1), on obtient :

$$u(x+h_x,y) + u(x-h_x,y) = 2u(x,y) + h_x^2 \frac{\partial^2 u}{\partial x^2}(x,y) + \frac{1}{12}\frac{\partial^4 u}{\partial x^4}(x+\theta_x h_x,y)$$

$$\Leftrightarrow \boxed{-\frac{\partial^2 u}{\partial x^2}(x,y) = \frac{1}{h_x^2}\left(-u(x-h_x,y) + 2u(x,y) - u(x+h_x,y)\right) + \frac{1}{12}h_x^2 \frac{\partial^4 u}{\partial x^4}(x+\theta_x h_x,y)}. \tag{2.2}$$

De même : $\exists\ \theta_y \in]-1,1[$:

$$\boxed{-\frac{\partial^2 u}{\partial y^2}(x,y) = \frac{1}{h_y^2}\left(-u(x,y-h_y) + 2u(x,y) - u(x,y+h_y)\right) + \frac{1}{12}h_y^2 \frac{\partial^4 u}{\partial y^4}(x,y+\theta_y h_y)}. \tag{2.3}$$

En additionnant (2.2) et (2.3), on obtient :

$$\boxed{-\Delta u(x,y) = \frac{1}{h_x^2}\delta_x^2 + \frac{1}{h_y^2}\delta_y^2 + E_{h_x,h_y}} \tag{2.4}$$

avec :

$$\delta_x^2 := -u(x-h_x,y) + 2u(x,y) - u(x+h_x,y) \quad \text{et} \quad \delta_y^2 := -u(x,y-h_y) + 2u(x,y) - u(x,y+h_y)$$

et

$$E_{h_x,h_y} := \frac{1}{12}\left(h_x^2 \frac{\partial^4 u}{\partial x^4}(x+\theta_x h_x,y) + h_y^2 \frac{\partial^4 u}{\partial y^4}(x,y+\theta_y h_y) \right).$$

Si $h := h_x = h_y$, alors :

$$\boxed{-\Delta u(x,y) = \frac{1}{h^2}\left(-u(x-h,y) - u(x,y-h) + 4u(x,y) - u(x+h,y) - (x,y+h)\right) + E_h} \tag{2.5}$$

avec

$$E_h := \frac{1}{12}h^2 \left(\frac{\partial^4 u}{\partial x^4}(x+\theta_x h,y) + \frac{\partial^4 u}{\partial y^4}(x,y+\theta_y h) \right).$$

On peut utiliser l'égalité (2.5) sans l'erreur de troncature E_h et poser $x_i := ih$ et $y_j := jh$ pour $i,\ j \in \{0,\ \ldots,\ N\}$ (on utilise $N+1$ noeuds avec $h = 1/N$), $u_{i,j} :\approx u(x_i,y_j)$ et $f_{i,j} := f(x_i,y_j)$ pour obtenir le schéma numérique suivant :

$$\boxed{\frac{1}{h^2}\left(-u_{i-1,j} - u_{i,j-1} + 2u_{i,j} - u_{i+1,j} - u_{i,j+1}\right) = f_{i,j}}. \tag{2.6}$$

Soient $u := \begin{pmatrix} u_1 & \cdots & u_{N-1} \end{pmatrix}^T$ (avec $u_j := \begin{pmatrix} u_{1,j} & \cdots & u_{N-1,j} \end{pmatrix}^T$) le vecteur de la solution approchée linéarisé ($u_{0,\cdot} = 0, u_{\cdot,0} = 0, u_{N,\cdot} = 0$ et $u_{\cdot,N} = 0$ sont connus) et $f := \begin{pmatrix} f_1 & \cdots & f_{N-1} \end{pmatrix}^T$ (avec $f_j := \begin{pmatrix} f_{1,j} & \cdots & f_{N-1,j} \end{pmatrix}^T$) le vecteur du second membre exact linéarisé. Alors la forme matricielle du schéma numérique est la suivante (pour $N = 4$) :

$$\boxed{Au = f} \tag{2.7}$$

$$\Leftrightarrow \underbrace{\frac{1}{h^2}\begin{pmatrix} 4 & -1 & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 4 & -1 & \cdot & -1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 4 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot \\ -1 & \cdot & \cdot & 4 & -1 & \cdot & -1 & \cdot & \cdot \\ \cdot & -1 & \cdot & -1 & 4 & -1 & \cdot & -1 & \cdot \\ \cdot & \cdot & -1 & \cdot & -1 & 4 & \cdot & \cdot & -1 \\ \cdot & \cdot & \cdot & -1 & \cdot & \cdot & 4 & -1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & \cdot & -1 & 4 & -1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & -1 & 4 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{pmatrix}}_{=u} = \underbrace{\begin{pmatrix} f_{1,1} \\ f_{2,1} \\ f_{3,1} \\ f_{1,2} \\ f_{2,2} \\ f_{3,2} \\ f_{1,3} \\ f_{2,3} \\ f_{3,3} \end{pmatrix}}_{=f}.$$

La forme matricielle du schéma numérique est la suivante (pour N quelconque) :

$$\underbrace{\frac{1}{h^2}\begin{pmatrix} \boxed{M} & \boxed{-I} & \cdot & \cdot \\ \boxed{-I} & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \boxed{-I} \\ \cdot & \cdot & \boxed{-I} & \boxed{M} \end{pmatrix}}_{=A} \underbrace{\begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ u_{N-1} \end{pmatrix}}_{=u} = \underbrace{\begin{pmatrix} f_1 \\ \vdots \\ \vdots \\ f_{N-1} \end{pmatrix}}_{=f} \quad \text{avec} \quad M := \begin{pmatrix} 4 & -1 & \cdot & \cdot \\ -1 & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & -1 \\ \cdot & \cdot & -1 & 4 \end{pmatrix}.$$

Remarques

— La valeur en un point du maillage dépend de valeurs d'au plus 5 points du maillage.

— A est une matrice creuse : elle comporte 5 diagonales (dont 3 centrales) et 3 blocs de diagonales, où les blocs sont M et $-I$.

2.1.3 Existence et unicité de la solution approchée

Proposition A est définie-positive et $Au = f$ admet une unique solution.

Démonstration Montrons que A est définie-positive (on utilise la structure en bloc de A) :
Soit $x \in \mathbb{R}^{(N-1)\times(N-1)}$ avec $x = (x_j)_{1\le j\le N-1}$ et $x_j = (x_{i,j})_{1\le i\le N-1}$, alors :

$$x^T A x = \frac{1}{h^2}\begin{pmatrix} x_1^T & \cdots & \cdots & x_{N-1}^T \end{pmatrix} \begin{pmatrix} \boxed{M} & \boxed{-I} & \cdot & \cdot \\ \boxed{-I} & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \boxed{-I} \\ \cdot & \cdot & \boxed{-I} & \boxed{M} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_{N-1} \end{pmatrix}$$

$$= \frac{1}{h^2}\begin{pmatrix} x_1^T & x_2^T & \cdots & x_{N-2}^T & x_{N-1}^T \end{pmatrix} \begin{pmatrix} Mx_1 - x_2 \\ -x_1 + Mx_2 - x_3 \\ \vdots \\ -x_{N-3} + Mx_{N-2} - x_{N-1} \\ -x_{N-2} + Mx_{N-1} \end{pmatrix}$$

$$= \frac{1}{h^2}\left(x_1^T(Mx_1 - x_2) + x_2^T(-x_1 + Mx_2 - x_3) + \cdots + x_{N-2}^T(-x_{N-3} + Mx_{N-2} - x_{N-1}) + x_{N-1}^T(-x_{N-2} + Mx_{N-1})\right)$$

$$= \frac{1}{h^2}\left(x_1^T M x_1 - 2x_1^T x_2 + x_2^T M x_2 - x_2^T x_3 + \cdots + x_{N-2}^T x_{N-3} + x_{N-2}^T M x_{N-2} - 2x_{N-2}^T x_{N-1} + x_{N-1}^T M x_{N-1}\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-1} x_j^T M x_j - 2\sum_{j=1}^{N-2} x_j^T x_{j+1}\right).$$

De plus,

$$x_j^T M x_j = x_j^T\left(h^2 B + 2I\right)x = h^2 x_j^T B x_j + 2\sum_{i=1}^{N-1} x_{i,j}^2 = h^2 x_j^T B x_j + 2\|x_j\|^2$$

avec

$$B := \frac{1}{h^2}\begin{pmatrix} 2 & -1 & \cdot & \cdot \\ -1 & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & -1 \\ \cdot & \cdot & -1 & 2 \end{pmatrix}.$$

Donc on obtient :

$$x^T A x = \frac{1}{h^2}\left(\sum_{j=1}^{N-1}\left(h^2 x_j^T B x_j + 2\|x_j\|^2\right) - 2\sum_{j=1}^{N-2} x_j^T x_{j+1}\right) = \frac{1}{h^2}\left(\sum_{j=1}^{N-1} 2\|x_j\|^2 + h^2\sum_{j=1}^{N-1} x_j^T B x_j - 2\sum_{j=1}^{N-2} x_j^T x_{j+1}\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-1}\|x_j\|^2 + \sum_{j=1}^{N-1}\|x_j\|^2 - 2\sum_{j=1}^{N-2} x_j^T x_{j+1} + h^2\sum_{j=1}^{N-1} x_j^T B x_j\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-1}\|x_j\|^2 + \sum_{j=0}^{N-2}\|x_{j+1}\|^2 - 2\sum_{j=1}^{N-2} x_j^T x_{j+1} + h^2\sum_{j=1}^{N-1} x_j^T B x_j\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-2}\|x_j\|^2 + \|x_{N-1}\|^2 + \sum_{j=1}^{N-2}\|x_{j+1}\|^2 + \|x_1\|^2 - 2\sum_{j=1}^{N-2} x_j^T x_{j+1} + h^2\sum_{j=1}^{N-1} x_j^T B x_j\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-2}\left(\|x_j\|^2 - 2x_j^T x_{j+1} + \|x_{j+1}\|^2\right) + \|x_1\|^2 + \|x_{N-1}\|^2 + h^2\sum_{j=1}^{N-1} x_j^T B x_j\right)$$

$$= \frac{1}{h^2}\left(\sum_{j=1}^{N-2}\|x_j + x_{j+1}\|^2 + \|x_1\|^2 + \|x_{N-1}\|^2 + h^2\sum_{j=1}^{N-1} x_j^T B x_j\right)$$

et d'après la sous-sous-section 1.1.3, B est définie-positive donc $\forall\, x_j \in \mathbb{R}^N : x_j^T B x_j \geq 0$, donc $x^T A x \geq 0$.
Si $\sum_{j=1}^{N-2}\|x_j + x_{j+1}\|^2 + \|x_1\|^2 + \|x_{N-1}\|^2 + h^2\sum_{j=1}^{N-1} x_j^T B x_j = 0$, alors $x_j = 0_{\mathbb{R}^N}\ \forall\, j \in \{1,\ldots,N-1\}$ donc $x = 0_{\mathbb{R}^{(N-1)\times(N-1)}}$.
Donc A est définie-positive, donc A est inversible et $Au = f$ admet une unique solution.

2.1.4 Consistance du schéma et majoration de l'erreur de troncature

Notation Soit $d \in \{1,\ldots,4\}$. Alors, $C_{u,d} := \max\left\{\sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^d u}{\partial x^d}(x,y)\right|, \sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^d u}{\partial y^d}(x,y)\right|\right\}$.

Proposition Le schéma (2.7) est consistant : $\lim_{h\to 0}|E_h| = 0$ et $|E_h| \leq \frac{1}{6}h^2\,|C_{u,4}|$.

Démonstration

$$|E_{h_x,h_y}| = \frac{1}{12}\left|h_x^2\frac{\partial^4 u}{\partial x^4}(x+\theta_x h_x, y) + h_y^2\frac{\partial^4 u}{\partial y^4}(x, y+\theta_y h_y)\right|$$

$$\leq \frac{1}{12}\left|h_x^2 \sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^4 u}{\partial x^4}(x,y)\right| + h_y^2 \sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^4 u}{\partial y^4}(x,y)\right|\right|$$

$$\leq \frac{1}{12}\left|\left(h_x^2+h_y^2\right)\ \max\left\{\sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^4 u}{\partial x^4}(x,y)\right|, \sup_{(x,y)\in[0,1]^2}\left|\frac{\partial^4 u}{\partial y^4}(x,y)\right|\right\}\right| = \frac{1}{12}\left(h_x^2+h_y^2\right)|C_{u,4}|.$$

Si $h := h_x = h_y$ alors :

$$|E_h| \leq \frac{1}{6}h^2\,|C_{u,4}| \underset{h\to 0}{\longrightarrow} 0.$$

Remarque Le schéma (2.7) est d'ordre 2 pour x et pour y.

2.1.5 Convergence du schéma et majoration de l'erreur locale

Proposition *(admise)* Soit $e_j := \left(\|u_j - u(x_j)\|_\infty\right)_{0\leq j\leq N}$. Alors, le schéma utilisé est convergent :

$$\forall\ j \in \{1,\ldots,N-1\} : \lim_{h\to 0}\|e_j\|_\infty = 0$$

et

$$\exists\ C>0,\ \forall\ j\in\{1,\ldots,N-1\} : \|e_j\|_\infty \leq Ch^2\left(C_{u,4} + hC_{u,3}\right).$$

2.1.6 Méthode de résolution itérative

Soient $D := \mathrm{diag}(A)$, $-E$ la partie triangulaire inférieure stricte de A, $-F$ la partie triangulaire supérieure stricte de A et k l'itération. Alors, le schéma de la méthode Jacobi pour $Au = f$ est le suivant :

$$Du^{(k+1)} = (E+F)\,u^{(k)} + f,$$

avec :

$$D = \frac{1}{h^2}\begin{pmatrix} \boxed{D_0} & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & \boxed{D_0} \end{pmatrix} \quad \text{où} \quad D_0 := 4I,$$

$$E = \frac{1}{h^2}\begin{pmatrix} \boxed{D_-} & \cdot & \cdot & \cdot \\ \boxed{I} & \ddots & \cdot & \cdot \\ \cdot & \ddots & \ddots & \cdot \\ \cdot & \cdot & \boxed{I} & \boxed{D_-} \end{pmatrix} \quad \text{où} \quad D_- := \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \end{pmatrix},$$

$$F = \frac{1}{h^2}\begin{pmatrix} \boxed{D_+} & \boxed{I} & \cdot & \cdot \\ \cdot & \ddots & \ddots & \cdot \\ \cdot & \cdot & \ddots & \boxed{I} \\ \cdot & \cdot & \cdot & \boxed{D_+} \end{pmatrix} \quad \text{où} \quad D_+ := \begin{pmatrix} \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

donc

$$E+F = \frac{1}{h^2}\begin{pmatrix} \boxed{D_\star} & \boxed{I} & \cdot & \cdot \\ \boxed{I} & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \boxed{I} \\ \cdot & \cdot & \boxed{I} & \boxed{D_\star} \end{pmatrix} \quad \text{où} \quad D_\star := D_- + D_+.$$

A est creuse et par blocs, on peut donc développer ces termes :

$$Du^{(k+1)} = \frac{1}{h^2}\begin{pmatrix} \boxed{D_0} & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & \boxed{D_0} \end{pmatrix}\begin{pmatrix} u_1^{(k+1)} \\ \vdots \\ \vdots \\ u_{N-1}^{(k+1)} \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} D_0 u_1^{(k+1)} \\ \vdots \\ \vdots \\ D_0 u_{N-1}^{(k+1)} \end{pmatrix}.$$

Soit $j \in \{1, \dots, N-1\}$, alors :

$$D_0 u_j^{(k+1)} = 4I\begin{pmatrix} u_{1,j}^{(k+1)} \\ \vdots \\ \vdots \\ u_{N-1,j}^{(k+1)} \end{pmatrix} = \begin{pmatrix} 4u_{1,j}^{(k+1)} \\ \vdots \\ \vdots \\ 4u_{N-1,j}^{(k+1)} \end{pmatrix}.$$

En identifiant chaque terme, on obtient :

$$\boxed{\left(Du^{(k+1)}\right)_{i,j} = \frac{1}{h^2} 4u_{i,j}^{(k+1)}}. \tag{2.8}$$

$$(E+F)\,u^{(k)} + f = \frac{1}{h^2}\begin{pmatrix} \boxed{D_\star} & \boxed{I} & \cdot & \cdot \\ \boxed{I} & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \boxed{I} \\ \cdot & \cdot & \boxed{I} & \boxed{D_\star} \end{pmatrix}\begin{pmatrix} u_1^{(k+1)} \\ \vdots \\ \vdots \\ u_{N-1}^{(k+1)} \end{pmatrix} + f = \frac{1}{h^2}\begin{pmatrix} u_0^{(k)} + D_\star u_1^{(k)} + u_2^{(k)} \\ \vdots \\ \vdots \\ u_{N-2}^{(k)} + D_\star u_{N-1}^{(k)} + u_N^{(k)} \end{pmatrix} + f.$$

Soit $j \in \{1, \dots, N-1\}$, alors :

$$u_{j-1}^{(k)} + D_\star u_j^{(k)} + u_{j+1}^{(k)} = \begin{pmatrix} u_{1,j-1}^{(k)} \\ \vdots \\ \vdots \\ u_{N-1,j-1}^{(k)} \end{pmatrix} + \begin{pmatrix} \cdot & 1 & \cdot & \cdot \\ 1 & \cdot & \ddots & \cdot \\ \cdot & \ddots & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot \end{pmatrix}\begin{pmatrix} u_{1,j}^{(k)} \\ \vdots \\ \vdots \\ u_{N-1,j}^{(k)} \end{pmatrix} + \begin{pmatrix} u_{1,j+1}^{(k)} \\ \vdots \\ \vdots \\ u_{N-1,j+1}^{(k)} \end{pmatrix}$$

$$= \begin{pmatrix} u_{1,j-1}^{(k)} + u_{0,j}^{(k)} + u_{2,j}^{(k)} + u_{1,j+1}^{(k)} \\ \vdots \\ \vdots \\ u_{N-1,j-1}^{(k)} + u_{N-2,j}^{(k)} + u_{N,j}^{(k)} + u_{N-1,j+1}^{(k)} \end{pmatrix}.$$

En identifiant chaque terme, on obtient :

$$\boxed{\left((E+F)\,u^{(k)} + f\right)_{i,j} = \frac{1}{h^2}\left(u_{i,j-1}^{(k)} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right) + f_{i,j}}. \tag{2.9}$$

En imposant l'égalité entre (2.8) et (2.9), on obtient :

$$\frac{1}{h^2} 4u_{i,j}^{(k+1)} = \frac{1}{h^2}\left(u_{i,j-1}^{(k)} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right) + f_{i,j}$$

$$\Leftrightarrow \boxed{u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)} + h^2 f_{i,j}\right)}. \tag{2.10}$$

2.1.7 Méthode de résolution directe

Soit L tel que $A = LL^T$.

Proposition Soit A une matrice avec N diagonales inférieures, symétrique et définie-positive. Alors, la matrice L de la décomposition de Cholesky de A de A possède N diagonales inférieures.

Démonstration Même démonstration que dans la sous-sous-section 1.1.7 appliquée à N diagonales inférieures (On construit L par colonne. Si $j = 1$ et $i > j + N - 1$, alors $\underbrace{a_{i,j}}_{=0} - \underbrace{\sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k}}_{=0} = 0$ donc $\ell_{i,j} = 0$. Si $j > 1$ et $i > j + N - 1$, alors $\underbrace{a_{i,j}}_{=0} - \sum_{k=1}^{j-1} \underbrace{\ell_{i,k}}_{=0} \ell_{j,k} = 0$ donc $\ell_{i,j} = 0$).

Soient $u := \begin{pmatrix} u_1 & \cdots & u_{(N-1)^2} \end{pmatrix}$ et $f := \begin{pmatrix} f_1 & \cdots & f_{(N-1)^2} \end{pmatrix}$. A est symétrique, définie-positive et possède N diagonales inférieures donc $\exists\, L : LL^T = A$.
On remarque que la structure de A est telle que :

$$a_{i,j} = \begin{cases} \alpha & \text{si } i = j \\ \beta & \text{si } i = j + 1 \text{ et } j \not\equiv 0 \pmod{N-1} \\ \gamma & \text{si } i = j + N - 1 \\ 0 & \text{sinon} \end{cases}.$$

avec

$$\alpha := \frac{4}{h^2} \quad \text{et} \quad \beta := \gamma := -\frac{1}{h^2}.$$

Soit $d := i - j$ la diagonale de A ($i = d + j$, $d \in \{0, \ldots, N-1\}$). On calcule : pour j de 1 à $(N-1)^2$ puis pour d de 0 à $N-1$. Si $d = 0$, alors on calcule $\ell_{i,i}$, sinon, on calcule $\ell_{i,j}$. Avec la structure de A, on obtient :

$$\boxed{\begin{cases} \ell_{i,i} = \sqrt{\alpha - \displaystyle\sum_{k=\max\{1,j-N+d+1\}}^{i-1} \ell_{i,k}^2} & \text{si } d = 0 \\ \ell_{i,j} = \left(a_{i,j} - \displaystyle\sum_{k=\max\{1,j-N+d+1\}}^{j-1} \ell_{i,k}\ell_{j,k} \right) / \ell_{j,j} & \text{si } d > 0. \end{cases}} \tag{2.11}$$

Remarque Si A était pleine, la complexité algorithmique du calcul de L aurait été $O(N^6)$. Ici, on calcule $O(N^2)$ colonnes comportants $O(N)$ diagonales. Le calcul d'une case est $O(N)$. Donc on a réduit la complexité algorithmique du calcul de L à $O(N^4)$.

On peut résoudre $Au = f$ en résolvant $Ly = f$ puis $L^T u = y$ avec :

$$y_1 = \frac{f_1}{\ell_{1,1}} \quad \text{et} \quad \text{pour } i \text{ de 2 à } (N-1)^2 : y_i = \left(f_i - \sum_{k=\max\{1,i-N+1\}}^{i-1} \ell_{i,k} y_k \right) / \ell_{i,i}$$

et

$$\boxed{u_{(N-1)^2} = \frac{y_{(N-1)^2}}{\ell_{(N-1)^2,(N-1)^2}} \quad \text{et} \quad \text{pour } i \text{ de } (N-1)^2 - 1 \text{ à } 1 : u_i = \left(y_i - \sum_{k=i+1}^{\min\{i+N-1,(N-1)^2\}} \ell_{k,i} u_k \right) / \ell_{i,i}}. \tag{2.12}$$

2.2 Implémentation

Pour la suite, la fonction à approcher sera $f(x, y) := \sin(2\pi x) \sin(2\pi y)$ et $\varepsilon := 1 \cdot 10^{-10}$.

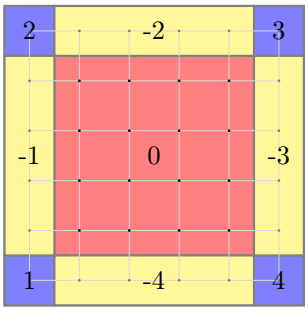
2.2.1 Version de base

Implémentation *(version 0 résumée, voir les détails dans `/Probleme-2D/base`)*
On implémente d'abord la version de base non optimisée de la résolution.
Étapes :

- créer des fonctions qui font le travail :
  - construire la matrice `A` (voir les fonctions `connaitre_bord` et `construire_matrice`),
  - calculer le second membre `f`,
  - calculer la solution approchée `u` (voir la fonction `resoudre_gauss`).

Commentaires

- Tout les tableaux utilisés sont linéarisés pour garantir la contiguité.
- Les numéros du type de bord de la fonction `connaitre_bord` sont les suivants :



- Pour calculer `u`, on résout le système linéaire avec la méthode de Gauss.
- On note ces résultats :

N	10	50	100
$\|e\|_\infty$	0.00038444	0.00001661	0.00000417
Temps d'exécution (s)	$<$0.1	4.1	278.9

- A est de taille $O(N^2)$ et la méthode de Gauss est $O(N^3)$ donc la complexité algorithmique est $O(N^6)$.

2.2.2 Version avec méthode de résolution itérative en séquentiel

Implémentation *(version 1 résumée, voir les détails dans `/Probleme-2D/sequentiel-1`)*
Pour commencer, on implémente la version séquentielle du schéma (2.10).
Étapes :

— créer des fonctions qui font le travail :
  — calculer le second membre `f`,
  — calculer la solution approchée `u` :
  Fonction qui applique le schéma à un point :

```
static inline __attribute__((always_inline))
double schema(double *f, double *u_anc, int i, int j){

    double res = 0.25 * (
    u_anc[IDX(i - 1, j)]
    + u_anc[IDX(i, j - 1)]
    + u_anc[IDX(i + 1, j)]
    + u_anc[IDX(i, j + 1)]
    + h_carre * f[IDX(i, j)]);

    return res;

}
```

Commentaire On note ces résultats :

N	10	50	100	300	500	700
Nombre d'itérations	102	2298	8506	66569	171980	320379
$\|e\|_\infty$	0.00038444	0.00001661	0.00000417	0.00000046	0.00000015	0.00000005
Temps d'exécution (s)	<0.1	<0.1	0.1	4.8	34.6	128.4

2.2.3 Version avec méthode de résolution itérative en parallèle avec OpenMP et MPI

Implémentation *(version 2 résumée, voir les détails dans `/Probleme-2D/parallele-1`)*
Ensuite, on implémente une version parallèle avec OpenMP du schéma (2.10).

Commentaires

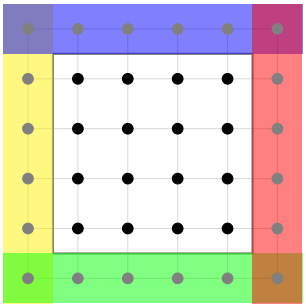
— Cette implémentation reprend exactement le même code que pour `/Probleme-2D/sequentiel-1` en ajoutant une directive `for` dans la boucle interne de la fonction `calculer_u_jacobi` et une directive `for` dans la boucle du calcul de la norme relative.

— On note ces résultats du temps d'exécution (en s) en fonction de N et du nombre de threads :

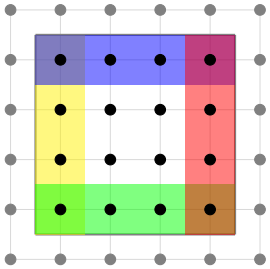
↓ Nombre de threads $N \rightarrow$	300	500	700
1	5.1	35.7	130.0
2	4.2	21.4	80.0
4	3.1	13.3	53.5
6	3.6	18.0	62.5
8	4.0	15.9	56.3

Implémentation *(version 3 résumée, voir les détails dans `/Probleme-2D/parallele-2`)*
On implémente ensuite une version parallèle du schéma (2.10) avec MPI. On décompose le domaine discrétisé en part égales (à une ligne et/ou colonne près en fonction de la divisibilité) de telle sorte que le périmètre de chaque sous-domaine soit minimal (en décomposant prioritairement en carrés plutôt qu'en bandes). Chaque processus a son sous-domaine avec un contour pour stocker les halos.

Illustration Schéma de la structure de `u_div` :



Un rectangle jaune (*resp.* bleu, rouge, vert) représente une zone à recevoir de la gauche (*resp.* du dessus, de la droite, du dessous).

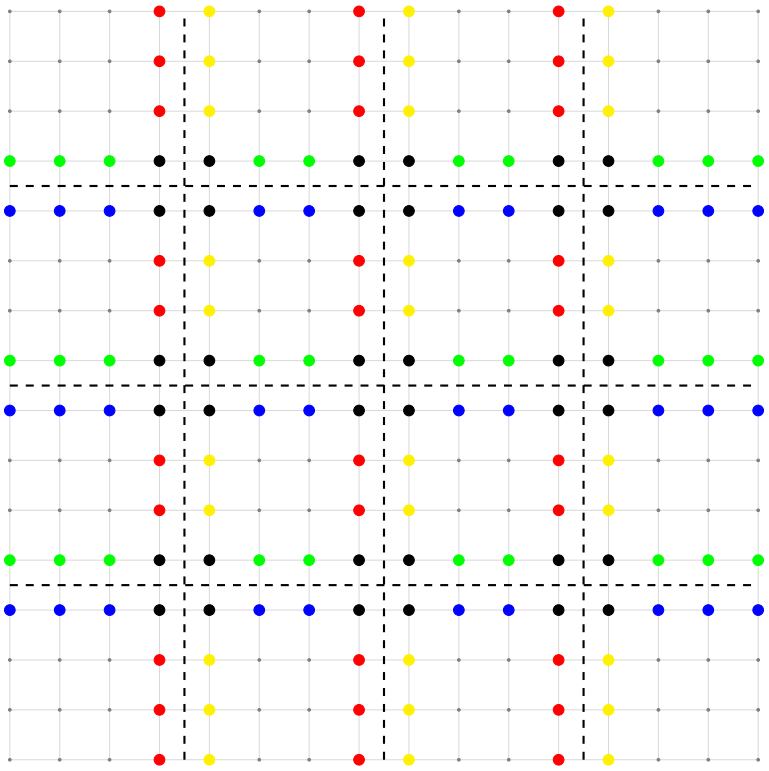


Un rectangle jaune (*resp.* bleu, rouge, vert) représente une zone à envoyer à gauche (*resp.* au dessus, à droite, en dessous).

— Un noeud en noir représente un noeud qui appartient au sous-domaine du processus, à calculer.

— Un noeud en gris représente un noeud à recevoir d'un voisin.

Illustration Schéma des noeuds envoyés ou reçus pour 16 processus et $N = 15$:



— Un noeud en jaune (*resp.* bleu, rouge, vert) représente un noeud à échanger avec son voisin de gauche (*resp.* du dessus, de droite, du dessous).

— Un noeud en noir représente un noeud à échanger avec plusieurs voisins.

Étapes :

— créer une topologie 2D non courbée,

— créer des fonctions qui, pour chaque processus, fait connaître sa position dans la topologie en déterminant :

  — le nombre de points à traiter `nb_div_i` et `nb_div_j`,

  — les noeuds de départ et d'arrivé `i_debut`, `j_debut`, `i_fin` et `j_fin`,

  — les voisins à gauche, en haut, à droite et en bas `voisins`,

— les types dérivés pour échanger les halos `ligne` et `colonne` et pour effectuer le regroupement final sur le rang 0 `bloc_send` :

```
void creer_types(){

    int taille_send[2] = {nb_pt_div_j + 2, nb_pt_div_i + 2};
    int sous_taille_send[2] = {nb_pt_div_j, nb_pt_div_i};
    int debut_send[2] = {1, 1};

    MPI_Type_contiguous(nb_pt_div_i, MPI_DOUBLE, &ligne);
    MPI_Type_vector(nb_pt_div_j, 1, nb_pt_div_i + 2, MPI_DOUBLE, &colonne);

    MPI_Type_create_subarray(2, taille_send, sous_taille_send, debut_send,
        MPI_ORDER_C, MPI_DOUBLE, &bloc_send);

    MPI_Type_commit(&ligne);
    MPI_Type_commit(&colonne);
    MPI_Type_commit(&bloc_send);

    MPI_Barrier(comm_2D);

}
```

— création des fonctions qui font le travail partagé sur chaque processus :

— calculer le second membre en parallèle `f_div`,

— calculer la solution approchée en parallèle `u_div` :

Fonction principale :

```
void calculer_u_jacobi(double *f_div, double *u_div){

    nb_iteration = 0;
    h_carre = 1.0 / pow(N, 2);
    int nb_iteration_max = INT_MAX;
    double norme = DBL_MAX;
    int i_boucle_debut; int j_boucle_debut;
    int i_boucle_fin; int j_boucle_fin;
    double *u_div_anc; double *permut;

    init_u_anc(&u_div_anc);
    for (int i = 0 ; i < (nb_pt_div_i + 2) * (nb_pt_div_j + 2) ; i ++){
        u_div[i] = 0.0;
    }

    infos_bornes_boucles(&i_boucle_debut, &j_boucle_debut, &i_boucle_fin, &
        j_boucle_fin);

    for (int iteration = 0 ; iteration < nb_iteration_max && norme > 1e-10 ;
        iteration ++){

        echanger_halos(u_div_anc);

        for (int j = j_boucle_debut ; j < j_boucle_fin ; j ++){
            for (int i = i_boucle_debut ; i < i_boucle_fin ; i ++){
                    u_div[IDX(i, j)] = schema(f_div, u_div, u_div_anc, i, j);
            }
        }

        norme = norme_infty_iteration(u_div, u_div_anc);

        permut = u_div; u_div = u_div_anc; u_div_anc = permut; nb_iteration
            ++;
```

```
    }

    terminaison(&permut, &u_div, &u_div_anc);

}
```

Fonction pour échanger les halos :

```
void echanger_halos(double *u_div){

    // Envoi haut, reception bas
    MPI_Sendrecv(&(u_div[IDX(1, nb_pt_div_j)]), 1, ligne, voisins[1],
        etiquette, &(u_div[IDX(1, 0)]), 1, ligne, voisins[3], etiquette,
        comm_2D, &statut);

    // Envoi bas, reception haut
    MPI_Sendrecv(&(u_div[IDX(1, 1)]), 1, ligne, voisins[3], etiquette, &(
        u_div[IDX(1, nb_pt_div_j + 1)]), 1, ligne, voisins[1], etiquette,
        comm_2D, &statut);

    // Envoi gauche, reception droite
    MPI_Sendrecv(&(u_div[IDX(1, 1)]), 1, colonne, voisins[0], etiquette, &(
        u_div[IDX(nb_pt_div_i + 1, 1)]), 1, colonne, voisins[2], etiquette,
        comm_2D, &statut);

    // Envoi droite, reception gauche
    MPI_Sendrecv(&(u_div[IDX(nb_pt_div_i, 1)]), 1, colonne, voisins[2],
        etiquette, &(u_div[IDX(0, 1)]), 1, colonne, voisins[0], etiquette,
        comm_2D, &statut);

}
```

— regrouper sur le rang 0 `u_div` et le stocker dans `u`.

Commentaires

— Cette implémentation reprend, pour de nombreuses fonctions, le même principe que `/Probleme-1D/parallele-1`, adaptées pour 2 variables. La principale différence se situe dans la gestion des halos.

— Les tableaux partiels possèdent 2 lignes et 2 colonnes supplémentaires pour accueillir les halos. Au début de chaque itération, l'échange des halos est effectué dans la fonction `echanger_halos`.

— Pour regrouper les résultats sur le rang 0, on utilise un type dérivé `bloc_recv` crée dynamiquement par le rang 0 (voir la fonction `regrouper_u`).

— On note ces résultats du temps d'exécution (en s) en fonction de N et du nombre de processus :

↓ Nombre de processus $N \rightarrow$	300	500	700
1	5.3	37.1	133.4
2	3.0	20.8	76.8
4	1.9	12.8	47.1
6	2.7	18.1	62.5
8	2.5	18.9	110.4

Implémentation *(version 4 résumée, voir les détails dans `/Probleme-2D/parallele-3`)*
Ensuite, on implémente une version parallèle avec MPI en utilisant des communications non bloquantes du schéma (2.10).

Commentaires

— Cette implémentation reprend exactement le même code que pour `/Probleme-2D/parallele-2`) en modifiant le mode de communication.

— Dès que la communication est lancée, on fait les calculs sur l'intérieur du sous-domaine (en excluant les bords locaux), après on vérifie / attend que la communication soit terminée et on fait les calculs sur les bords locaux avec la fonction `test_fin_echange_halos`.

— Pour calculer sur les bords du sous-domaine (2 bandes verticales, 2 bandes horizontales et 4 coins), on utilise la fonction `calculer_u_jacobi_bords`.

— On note ces résultats du temps d'exécution (en s) en fonction de N et du nombre de processus :

↓ Nombre de processus $N \rightarrow$	300	500	700
1	5.3	37.3	136.5
2	3.0	21.4	79.0
4	1.8	14.6	56.4
6	2.6	19.6	65.5
8	3.8	28.2	142.2

2.2.4 Version avec méthode de résolution directe en séquentiel

Implémentation *(version 5 résumée, voir les détails dans `/Probleme-1D/sequentiel-2`)*
Ensuite, on implémente une version séquentielle du schéma (2.12). On calcule la décomposition de Cholesky avec le schéma (2.11). On travaille par diagonales `d` et par colonnes `j`. On utilise une structure qui stocke l'entier N et N pointeurs de pointeurs de flottants doubles. Chaque pointeur pointe vers le premier élément d'une diagonale. On a :
$\ell_{i,j}$ qui correspond à `(L -> diags)[i-j][j]`.
Étapes :

— créer la structure pour stocker une matrice de la même forme que L :

```
struct mat_Nbandes{
    int N;
    double **diags;
};
```

— créer des fonctions pour allouer et libérer la structure :
Fonction pour allouer la structure :

```
void init_mat_Nbandes(struct mat_Nbandes *A){

    A -> N = N;
    A -> diags = (double **)malloc(N * sizeof(double *));
    for (int i = 0 ; i < N ; i ++){
        (A -> diags)[i] = (double *)malloc((idx_max - i) * sizeof(double));
    }

}
```

Fonction pour libérer la structure :

```
void liberer_mat_Nbandes(struct mat_Nbandes *A){

    int N = A -> N;
    for (int i = 0 ; i < N ; i ++){
        free((A -> diags)[i]);
    }
    free(A -> diags);

}
```

— créer la fonction pour obtenir la décomposition de Cholesky en utilisant la structure :

```
void calculer_cholesky(struct mat_Nbandes *L){

    h_carre = 1.0 / pow(N, 2);
    double alpha = 4.0 / h_carre;

    for (int j = 0 ; j < idx_max ; j ++){
        for (int d = 0 ; d < N && j + d < idx_max ; d ++){

            int i = d + j;

            if (d == 0){
                (L -> diags)[0][j] = alpha;
                for (int k = max(0, j - N + d + 1) ; k < i ; k ++){
                    int d_1 = i - k;
                    (L -> diags)[0][j] -= pow((L -> diags)[d_1][k], 2);
                }
                (L -> diags)[0][j] = sqrt((L -> diags)[0][j]);
            }

            else{
                (L -> diags)[d][j] = valeur_a(i, j);
                for (int k = max(0, j - N + d + 1) ; k < j ; k ++){
                    int d_1 = i - k;
                    int d_2 = j - k;
                    (L -> diags)[d][j] -= (L -> diags)[d_1][k] * (L -> diags)[d_2
                        ][k];
                }
                (L -> diags)[d][j] /= (L -> diags)[0][j];
            }

        }
    }

}
```

Fonction pour obtenir la valeur de $a_{i,j}$ en fonction des paramètres i et j :

```
static inline __attribute__((always_inline)) double valeur_a(int i, int j){

    double res;

    if (i == j){
        res = 4.0 / h_carre;
    }
    else if (i == j + 1 && j % (N - 1) != (N - 2)){
        res = -1.0 / h_carre;
    }
    else if (i == j + N - 1){
        res = -1.0 / h_carre;
    }
    else{
        res = 0.0;
    }

    return res;

}
```

Test (affichage effectué automatiquement en exécutant `Probleme-2D/Binaires/sequentiel-2`) pour avoir un aperçu de la compression :

```
Illustration de la structure mat_Nbandes (exemple pour N petit) :

Structure mat_Nbandes :
N = 4
diag[0] =  8.0000  7.7460  7.7287  7.7275  7.3829  7.3668  7.6995  7.3261  7.3139
diag[1] = -2.0000 -2.0656 -0.1380 -2.2184 -2.3331 -0.2074 -2.2717 -2.4056
diag[2] =  0.0000 -0.5164 -0.5521 -0.0370 -0.6222 -0.6863 -0.0585
diag[3] = -2.0000 -2.0656 -2.0702 -2.0705 -2.1672 -2.1719

Matrice reelle correspondante :
 8.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
-2.0000  7.7460  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
 0.0000 -2.0656  7.7287  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
-2.0000 -0.5164 -0.1380  7.7275  0.0000  0.0000  0.0000  0.0000  0.0000
 0.0000 -2.0656 -0.5521 -2.2184  7.3829  0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000 -2.0702 -0.0370 -2.3331  7.3668  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000 -2.0705 -0.6222 -0.2074  7.6995  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000 -2.1672 -0.6863 -2.2717  7.3261  0.0000
 0.0000  0.0000  0.0000  0.0000  0.0000 -2.1719 -0.0585 -2.4056  7.3139
```

— créer des fonctions pour résoudre le système linéaire en utilisant la structure :
Fonction principale :

```c
void resoudre_cholesky(double *f, double *u){

    struct mat_Nbandes L;
    double *y, *u_int, *f_int;

    init_u_bord(u);
    u_int = (double *)malloc(idx_max * sizeof(double));
    f_int = (double *)malloc(idx_max * sizeof(double));
    extraire_interieur(u, u_int, nb_pt);
    extraire_interieur(f, f_int, nb_pt);
    y = (double *)malloc(idx_max * sizeof(double));

    init_mat_Nbandes(&L);
    calculer_cholesky(&L);

    resoudre_cholesky_descente(&L, f_int, y);
    resoudre_cholesky_remontee(&L, y, u_int);
    inserer_interieur(u_int, u, nb_pt);

    liberer_mat_Nbandes(&L);
    free(u_int);
    free(f_int);
    free(y);

}
```

Fonction pour résoudre $Ly = f$:

```c
void resoudre_cholesky_descente(struct mat_Nbandes *L, double *f, double *y){

    y[0] = f[0] / (L -> diags)[0][0];

    for (int i = 1 ; i < idx_max ; i ++){
        y[i] = f[i];
        for (int k = max(0, i - N + 1) ; k < i ; k ++){
            int d = i - k;
            y[i] -= (L -> diags)[d][k] * y[k];
        }
```

```
        y[i] /= (L -> diags)[0][i];
    }

}
```

Fonction pour résoudre $L^T u = y$:

```
void resoudre_cholesky_remontee(struct mat_Nbandes *L, double *y, double *u){

    u[idx_max - 1] = y[idx_max - 1] / (L -> diags)[0][idx_max - 1];

    for (int i = idx_max - 2 ; i >= 0 ; i --){
        u[i] = y[i];
        for (int k = i + 1 ; k < min(i + N, idx_max) ; k ++){
            int d = k - i;
            u[i] -= (L -> diags)[d][i] * u[k];
        }
        u[i] /= (L -> diags)[0][i];
    }

}
```

Commentaires

— Comme on calcule u sur l'intérieur, on fait la résolution avec $u|_{\text{int}}$ et $f|_{\text{int}}$. On utilise les fonctions `extraire_interieur` pour extraire l'intérieur d'une matrice linéarisée et `inserer_interieur` pour remettre l'intérieur d'une matrice linéarisé dans la matrice initiale.

— On note ces résultats :

N	10	50	100	300	500	700
$\|e\|_\infty$	0.00038444	0.00001661	0.00000417	0.00000046	0.00000017	0.00000009
Temps d'exécution (s)	<0.1	<0.1	<0.1	4.6	35.8	383.5

— Cette méthode est impossible à paralléliser à cause des dépendances.

— Cette méthode entraîne probablement beaucoup de cach-miss.

— A possède $O(N^2)$ colonne. Pour chaque colonne, il y a $O(N)$ lignes à calculer. Pour chaque case, il y a $O(N)$ opérations. Donc la complexité algorithmique est $O(N^4)$.

Implémentation *(version 6 résumée, voir les détails dans `/Probleme-1D/sequentiel-3`)*
Enfin, on implémente une version séquentielle du schéma (2.12) avec la librairie `cholmod` qui utilise une structure de matrice creuse CSC.
Étapes :

— créer une fonction pour définir la structure de matrice creuse en créant des tableaux :

  — `lignes` qui contient les indices des lignes où se trouve une valeur non nulle,

  — `valeurs` qui contient les valeurs aux indices stockés,

  — `offsets` qui contient le nombre de valeurs non nulles d'une colonne,

  (voir les fonctions `construire_matrice_creuse` et `connaitre_bord`).

— créer une fonction qui fait le travail :

```
void resoudre(cholmod_sparse *A, double *f, double *u){

    h_carre = 1.0 / pow(N, 2);
    double *f_int = (double *)malloc(idx_max * sizeof(double));
    double *u_int = (double *)malloc(idx_max * sizeof(double));

    extraire_interieur(f, f_int, nb_pt);
```

```
    extraire_interieur(u, u_int, nb_pt);

    cholmod_dense *f_dense = cholmod_allocate_dense(A -> nrow, 1, A -> nrow,
        CHOLMOD_REAL, &c);
    memcpy(f_dense -> x, f_int, A -> nrow * sizeof(double));

    cholmod_factor *L = cholmod_analyze(A, &c);
    cholmod_factorize(A, L, &c);

    cholmod_dense *u_dense = cholmod_solve(CHOLMOD_A, L, f_dense, &c);

    memcpy(u_int, u_dense -> x, A -> nrow * sizeof(double));

    inserer_interieur(u_int, u, nb_pt);

    cholmod_free_factor(&L, &c);
    cholmod_free_dense(&f_dense, &c);
    cholmod_free_dense(&u_dense, &c);

    free(f_int);
    free(u_int);

}
```

Commentaires

— Le nombre d'éléments non nuls de A est de : $5 \cdot (N-3)^2 + 4 \cdot 4 \cdot (N-3) + 4 \cdot 3 = (5N+1)(N-3) + 12$.

— On note ces résultats :

N	1000	2000	3000	4000	5000
$\|e\|_\infty$	0.00000004	0.00000001	$<$0.00000001	$<$0.00000001	$<$0.00000001
Temps d'exécution (s)	0.7	15.7	39.3	79.3	174.5

2.2.5 Comparaison des performances des méthodes

— La version de base est inutilisable en pratique.

— Les versions itératives, en particulier avec le parallélisme, donnent de bien meilleurs résultats.

— La version avec la bibliothèque `cholmod` donne d'excellents résultats.

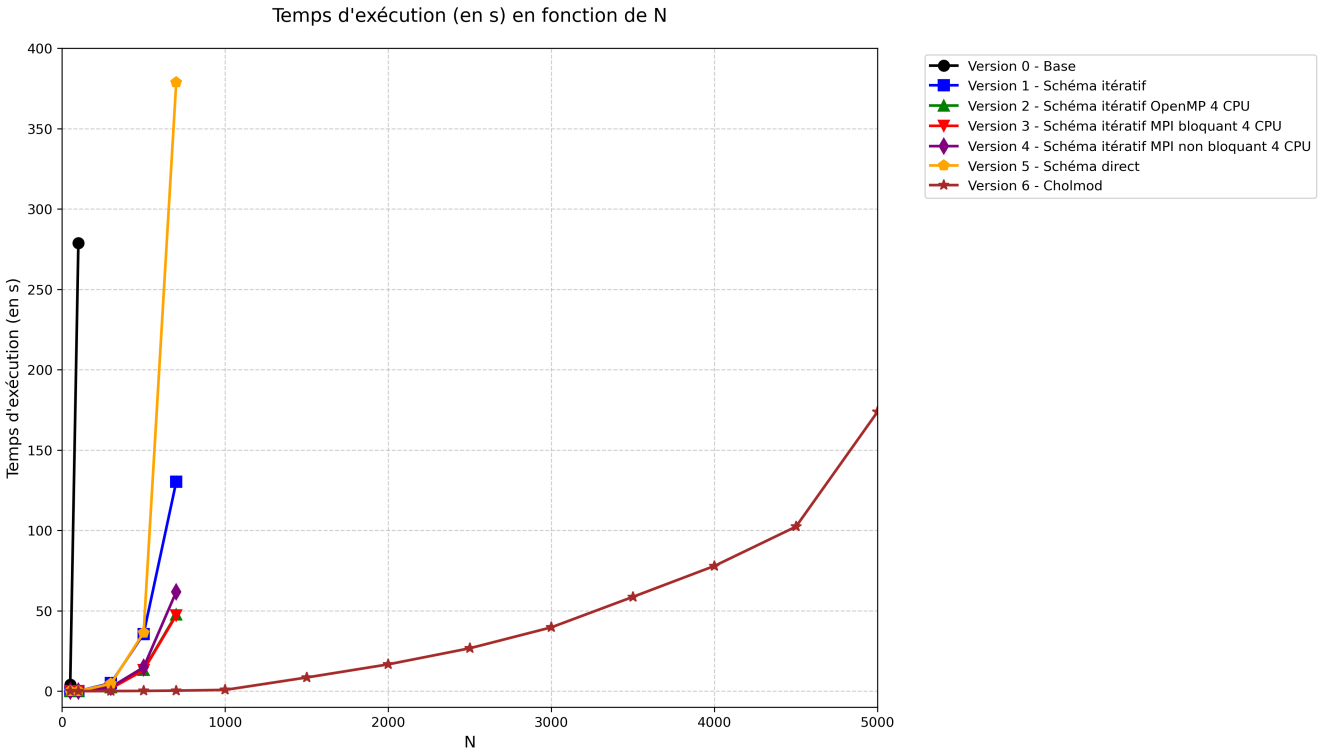


FIGURE 1 – Comparaison des versions

3 Équation des ondes en dimension 1

3.1 Analyse numérique

3.1.1 Présentation du problème

Soient $L, T > 0, D :=]0, L[$. Soit le problème suivant :
Trouver u de classe C^2 telle que :

$$\begin{cases} \dfrac{\partial^2 u}{\partial t^2} - c^2 \dfrac{\partial^2 u}{\partial x^2} = 0 \quad x \in D \ \forall\, t \in]0, T], c > 0 \\ u(x) = 0 \quad \forall\, x \in \partial D \ \forall\, t \in [0, T] \\ u(x, 0) =: u_0(x) \quad \forall\, x \in D \\ \dfrac{\partial u}{\partial t}(x, 0) =: u_1(x) \quad \forall\, x \in D \end{cases}.$$

Un exemple de solution connue est si $c = 1, u_0(x) = \sin(\pi x)$ et $u_1(x) = 0$, alors $u(x, t) = \cos(\pi t)\sin(\pi x)$.

3.1.2 Schéma numérique

Soient $t, h_t \in]0, T[$ tels que $[t, t + h_t] \subset [0, T]$. On utilise la formule de Taylor à l'ordre 3 :

$$\exists\, \theta_+ \in]0, 1[: u(x, t + h_t) = u(x, t) + h_t \frac{\partial u}{\partial t}(x, t) + \frac{1}{2} h_t^2 \frac{\partial^2 u}{\partial t^2}(x, t) + \frac{1}{6} h_t^3 \frac{\partial^3 u}{\partial t^3}(x, t) + \frac{1}{24} h_t^4 \frac{\partial^4 u}{\partial t^4}(x, t + \theta_+ t_h),$$

$$\exists\, \theta_- \in]-1, 0[: u(x, t - h_t) = u(x, t) - h_t \frac{\partial u}{\partial t}(x, t) + \frac{1}{2} h_t^2 \frac{\partial^2 u}{\partial t^2}(x, t) - \frac{1}{6} h_t^3 \frac{\partial^3 u}{\partial t^3}(x, t) + \frac{1}{24} h_t^4 \frac{\partial^4 u}{\partial t^4}(x, t + \theta_- t_h).$$

En additionnant, on obtient :

$$\boxed{u(x,t+h_t)+u(x,t-h_t)=2u(x,t)+h_t^2\frac{\partial^2 u}{\partial t^2}(x,t)+\frac{1}{24}h_t^4\left(\frac{\partial^4 u}{\partial t^4}(x,t+\theta_+h)+\frac{\partial^4 u}{\partial t^4}(x,t+\theta_-h)\right)}. \tag{3.1}$$

D'après le théorème des valeurs intermédiaires, on a :

$$\exists\,\theta\in\,]-1,1[\,:u(x,t+h_t):2\frac{\partial^4 u}{\partial t^4}(x,t+\theta h_t)=\left(\frac{\partial^4 u}{\partial t^4}(x,t+\theta_+h)+\frac{\partial^4 u}{\partial t^4}(x,t+\theta_-h)\right)$$

En injectant dans (3.1), on obtient :

$$\begin{aligned}u(x,t+h_t)+u(x,t-h_t)&=2u(x,t)+h_t^2\frac{\partial^2 u}{\partial t^2}(x,t)+\frac{1}{12}h_t^4\frac{\partial^4 u}{\partial t^4}(x,t+\theta h_t)\\ \Leftrightarrow\ &\boxed{\frac{\partial^2 u}{\partial t^2}(x,t)=\frac{1}{h_t^2}\left(u(x,t+h_t)-2u(x,t)+u(x,t-h_t)\right)+E_{h_t}}.\end{aligned} \tag{3.2}$$

avec

$$E_{h_t}:=-\frac{1}{12}h_t^2\frac{\partial^4 u}{\partial t^4}(x,t+\theta h_t).$$

Avec l'égalité (1.2) (sans l'erreur de troncature) et en injectant dans l'équation sans l'erreur de troncature E_{h_t}, on obtient :

$$\boxed{u(x,t+h)=-u(x,t-h_t)+\left(2-2\frac{c^2h_t^2}{h^2}\right)u(x,t)+\frac{c^2h_t^2}{h^2}\left(u(x-h,t)+u(x+h,t)\right)}. \tag{3.3}$$

On pose $x_i:=ih$ et $t_k:=kh_t$ pour $i\in\{0,\dots,N\}$ et $t\in\{0,\dots,T_n\}$ (on utilise $N+1$ noeuds avec $h=1/N$ et N_t+1 instants avec $h_t=1/N_t$), $u_i^k:\approx u(x_i,t_k)$ et $f_i^k:\approx f(x_i,t_k)$, si $k>0$, alors on obtient le schéma numérique suivant :

$$\boxed{u_i^{k+1}=-u_i^{k-1}+\left(2-2\frac{c^2h_t^2}{h^2}\right)u_i^k+\frac{c^2h_t^2}{h^2}\left(u_{i-1}^k+u_{i+1}^k\right)}. \tag{3.4}$$

Soit $i\in\{1,\dots,N-1\}$. Alors, de plus,

$$\begin{aligned}\frac{\partial u}{\partial t}(x_i,0)\approx\frac{u_i^1-u_i^0}{h_t}=u^1(x_i)\Leftrightarrow u_i^1&=h_tu_1(x_i)+u_0(x_i)\\ \Leftrightarrow\ &\boxed{u_i^1=h_tu_1(x_i)+u_0(x_i)}.\end{aligned} \tag{3.5}$$

Finalement, on obtient le schéma numérique suivant :

$$\boxed{\begin{cases}u_i^1=h_tu_1(x_i)+u_0(x_i)\\ u_i^{k+1}=-u_i^{k-1}+2\left(1-\dfrac{c^2h_t^2}{h^2}\right)u_i^k+\dfrac{c^2h_t^2}{h^2}\left(u_{i-1}^k+u_{i+1}^k\right)\quad\text{si } k>0\end{cases}}. \tag{3.6}$$

Remarques

— Le schéma (3.6) est explicite : il dépend de valeurs connues et ne nécessite pas de résoudre un système linéaire.

— Pour $k>0$, le schéma (3.6) dépend de valeurs en $k-1$ et en $k-2$.

3.1.3 Existence et unicité de la solution approchée

Le schéma (3.6) est explicite et dépend de termes connus et définis donc il existe une unique solution approchée.

3.1.4 Consistance du schéma

Proposition Le schéma (3.6) est consistant en espace et en temps : $\lim\limits_{h\to0}|E_h|=0$ et $\lim\limits_{h_t\to0}|E_{h_t}|=0$.

Remarque Le schéma (3.6) est d'ordre 2 pour x et pour y et d'ordre 2 pour t.

3.1.5 Stabilité et convergence du schéma

Proposition *(admise)* Le schéma (3.6) est convergent si $c\dfrac{h_t}{h} \leq 1$.

Remarques

— On vérifiera cette propriété à la fin de la sous-section 3.2 avec un exemple.

— Cette proposition s'appelle la condition de Courant-Friedrich-Levy (CFL).

3.2 Implémentation

Pour la suite, la fonction à approcher aura pour conditions initiales $u_0(x) = \sin(\pi x)$ et $u_1(x) = 0$.

A la différence des problèmes stationnaires, il y a un tableau pour u à chaque pas de temps. Pour calculer l'erreur, on met en place la stratégie suivante :

— On créera pour chaque version 2 fonctions de résolutions : une qui calcule uniquement la solution approchée (pour mesurer le temps et/ou faire des entrées/sorties pour la visualisation) et une qui calcule la solution approchée en même temps que la solution exacte à chaque itération pour avoir l'erreur.

— On ne calcule pas la solution exacte et la solution approchée séparément pour ne pas devoir stocker u à chaque pas de temps en mémoire.

— On définiera l'erreur `erreur_infty` comme : $\|e\|_\infty^\infty := \max\limits_{1\leq k\leq N_t} \|e_k\|_\infty$ avec $\|e_k\|_\infty = \|e\|_\infty$ à l'itération k. Une macro `EXACTE` sera activée ou non pour basculer entre le calcul pratique ou le calcul de l'erreur.

— u ne sera pas stocké en mémoire pour tout k mais seulement u_{T_k}. Pour stocker la simulation intégrale, une macro `ECRITURE` sera activée ou non pour écrire chaque u_k dans un fichier (en mode ajout) de manière contigue. Le fichier pourra être vu comme un tableau 3D ou la dimension 0 sera pour t, et les dimensions 1 et 2 seront pour x et y.

— On mesurera donc séparément ces informations : $\|e\|_\infty^\infty$, le temps d'exécution sans écriture dans un fichier et le temps d'exécution avec écriture dans un fichier.

Implémentation *(version 1 résumée, voir les détails dans `/Probleme-Ondes/sequentiel-1`)*
Pour commencer, on implémente une version séquentielle du schéma (3.6).
Étapes :

— créer des fonctions qui font le travail :

— calculer la solution approchée `u` :

Fonction principale :

```
void calculer_u(double *u){

    const_1 = pow(c, 2) * pow(h_t, 2) / pow(h, 2);
    double *u_anc_0; double *u_anc_1; double *permut;
    init_u_0(u_0, &u_anc_1); init_u_1(u_1, u_anc_1, &u_anc_0);
    for (int i = 0 ; i < nb_pt ; i ++){
        u[i] = 0.0;
    }

    for (int k = 1 ; k <= N_t ; k ++){

        # ifdef ECRITURE
        ecrire_double_iteration(u_anc_0);
        # endif

        for (int i = 1 ; i < nb_pt - 1 ; i ++){
            u[i] = schema(u_anc_0, u_anc_1, i, k);
        }

        permut = u_anc_1; u_anc_1 = u_anc_0; u_anc_0 = u; u = permut;

    }
```

```
    # ifdef ECRITURE
    ecrire_double_iteration(u);
    # endif

    terminaison(&permut, &u, &u_anc_0, &u_anc_1);

}
```

Fonction qui applique le schéma à un point :

```
static inline __attribute__((always_inline))
double schema(double *u_anc_0, double *u_anc_1, int i, int k){

    // const_1 = pow(c, 2) * pow(h_t, 2) / pow(h, 2)
    double res = -u_anc_1[i] + 2 * (1 - const_1) * u_anc_0[i] + const_1 * (
        u_anc_0[i - 1] + u_anc_0[i + 1]);

    return res;

}
```

Fonction pour initialiser u_1 :

```
void init_u_1(double (*fonction)(double), double *u_anc_1, double **u_anc_0){

    *u_anc_0 = (double *)malloc(nb_pt * sizeof(double));

    (*u_anc_0)[0] = 0.0;
    (*u_anc_0)[nb_pt - 1] = 0.0;

    for (int i = 1; i < nb_pt - 1; i++) {
        (*u_anc_0)[i] = u_anc_1[i] + h_t * fonction(i * h);
    }
}
```

Fonction pour terminer :

```
void terminaison(double **permut, double **u, double **u_anc_0, double **
    u_anc_1){

    int nb_permut = 0;

    if (N_t % 3 == 1){
        nb_permut = 2;
    }
    else if (N_t % 3 == 2){
        nb_permut = 1;
    }

    for (int i = 0 ; i < nb_permut ; i ++){
        *permut = *u_anc_1; *u_anc_1 = *u_anc_0; *u_anc_0 = *u; *u = *permut;
    }

    free(*u_anc_0); free(*u_anc_1);

}
```

Commentaires

— La fonction pour le calcul de la solution exacte est `calculer_u_u_exact`.

— Contrairement aux problèmes stationnaires, on n'initialise pas `f` dans `main`. Comme il dépend du temps, on

le calcule dans la boucle la plus interne sans le stocker dans un tableau.

— On note ces résultats de $\|e\|_\infty^\infty$ en fonction de h et de h_t (avec $L = 1, T = 1$ et $c = 1$) (disponibles aussi dans `Textes/resultats_erreurs.txt`) :

$\downarrow h \quad h_t \rightarrow$	$1/100$	$1/200$	$1/500$	$1/1000$
$1/100$	0.01570926	0.00780545	0.00307972	0.00150733
$1/200$	∞_f	0.00785414	0.00312801	0.00155531
$1/500$	∞_f	∞_f	0.00314160	0.00156886
$1/1000$	∞_f	∞_f	∞_f	0.00157080

où ∞_f est ou bien l'infini des flottants double précision (`inf`), ou bien une valeur très élevée. On vérifie la proposition énoncée en sous-sous-section 3.1.5, les valeurs ∞_f sont bien atteintes lorsque $c\frac{h_t}{h} > 1$ donc la proposition de CFL est vérifiée pour cet exemple.

— Lorsque h/h_t est fixé, le schéma semble bien converger.

— On ne s'intéresse pas ici aux temps d'exécutions.

— La complexité algorithmique est $O(N \cdot N_t)$.

4 Équation de la chaleur en dimension 2

4.1 Analyse numérique

4.1.1 Présentation du problème

Soient $L, T > 0, f :]0, L[\times]0, L[\times]0, T] \to \mathbb{R}$ continue et bornée, $D :=]0, L[\times]0, L[$. Soit le problème suivant : Trouver u de classe C^2 telle que :

$$\begin{cases} \dfrac{\partial u}{\partial t} - a\Delta u = f(x,y,t) \quad \forall\ (x,y) \in D\ \forall\ t \in]0,T]\,, a > 0 \\ u\,(x,y,t) = 0 \quad \forall\ (x,y) \in \partial D\ \forall\ t \in [0,T] \\ u\,(x,y,0) =: u_0(x,y) \quad \forall\ (x,y) \in D \end{cases}.$$

Un exemple de solution connue est si $f\,(x,y,t) = \left(-\lambda + 2a\pi^2\right)\sin(\pi x)\sin(\pi y)\,e^{-\lambda t}$ et $u_0\,(x,y) = \sin(\pi x)\sin(\pi y)$, alors $u\,(x,y,t) = \sin(\pi x)\sin(\pi y)\,e^{-\lambda t}$.

4.1.2 Schémas numériques

Méthode explicite

Soient $t, h_t \in]0,T[$ tels que $[t, t+h_t] \subset [0,T]$. On utilise la formule de Taylor à l'ordre 2 (en avant) :

$$\exists\ \theta_t \in]0,1[: u\,(x,y,t+h_t) = u\,(x,y,t) + h_t\frac{\partial u}{\partial t}\,(x,y,t) + \frac{1}{2}h_t^2\frac{\partial^2 u}{\partial t^2}\,(x,y,\theta_t h_t)\,.$$

donc $\exists\ \theta_t \in]0,1[$:

$$\boxed{\frac{\partial u}{\partial t}\,(x,y,t) = \frac{1}{h_t}\left(-u\,(x,y,t) + u\,(x,y,t+h_t)\right) + E_{h_t}} \tag{4.1}$$

avec :

$$E_{h_t} := -\frac{1}{2}h_t\frac{\partial^2 u}{\partial t^2}\,(x,y,t+\theta_t h_t)\,.$$

Avec l'égalité (2.4) (sans l'erreur de troncature avec) $h := h_x = h_y$ et en injectant dans l'équation sans l'erreur de troncature E_{h_t}, on obtient :

$$\frac{1}{h_t}\left(-u\,(x,y,t) + u\,(x,y,t+h_t)\right) - \frac{a}{h^2}\left(u\,(x-h,y,t) + u\,(x,y-h,t) - 4u\,(x,y,t) + u\,(x+h,y,t) + u\,(x,y+h,t)\right) = f(x,y,t)$$

$$\Leftrightarrow \boxed{\begin{array}{c} u(x,y,t+h_t) \\ = \alpha u(x,y,t) + \beta\left(u(x-h,y,t) + u(x,y-h,t) + u(x+h,y,t) + u(x,y+h,t)\right) + h_t f(x,y,t) \end{array}} \tag{4.2}$$

avec :

$$\alpha := 1 - \frac{4ah_t}{h^2} \quad \text{et} \quad \beta := \frac{ah_t}{h^2}.$$

On pose $x_i := ih, y_j := jh$ et $t_k := kh_t$ pour $i,j \in \{0,\dots,N\}$ et $k \in \{0,\dots,T\}$ (on utilise $N+1$ noeuds avec $h = L/N$ et $T+1$ instants avec $h_t = T/N_t$), $u_{i,j}^k :\approx u(x_i,y_j,t_k)$ et $f_{i,j}^k := f(x_i,y_j,t_k)$, on obtient le schéma numérique suivant :

$$\boxed{u_{i,j}^{k+1} = \alpha u_{i,j}^k + \beta\left(u_{i-1,j}^k + u_{i,j-1}^k + u_{i+1,j}^k + u_{i,j+1}^k\right) + h_t f_{i,j}^k}. \tag{4.3}$$

Méthode implicite

Soient $t, h_t \in]0,T[$ tels que $[t, t+h_t] \subset [0,T]$. On utilise la formule de Taylor à l'ordre 2 (en arrière) :

$$\exists\, \theta_t \in]-1,0[: u(x,y,t) = u(x,y,t+h_t) - h_t \frac{\partial u}{\partial t}(x,y,t+h_t) + \frac{1}{2}h_t^2 \frac{\partial^2 u}{\partial t^2}(x,y,t+(\theta_t+1)h_t).$$

donc $\exists\, \theta_t \in]-1,0[$:

$$\boxed{\frac{\partial u}{\partial t}(x,y,t+h_t) = \frac{1}{h_t}\left(u(x,y,t+h_t) - u(x,y,t)\right) + E_{h_t}} \tag{4.4}$$

avec :

$$E_{h_t} := \frac{1}{2}h_t \frac{\partial^2 u}{\partial t^2}(x,y,t+(\theta_t+1)h_t).$$

Avec l'égalité (2.4) (sans l'erreur de troncature) avec $h := h_x = h_y$ et en injectant dans l'équation sans l'erreur de troncature E_{h_t}, on obtient :

$$\boxed{\begin{array}{c} \alpha u(x,y,t+h_t) + \beta\left(u(x-h,y,t+h_t) + u(x,y-h,t+h_t) + u(x+h,y,t+h_t) + u(x,y+h,t+h_t)\right) \\ = u(x,y,t) + h_t f(x,y,t+h_t) \end{array}} \tag{4.5}$$

avec :

$$\alpha := 1 + \frac{4ah_t}{h^2} \quad \text{et} \quad \beta := -\frac{ah_t}{h^2}.$$

On pose $x_i := ih, y_j := jh$ et $t_k := kh_t$ pour $i,j \in \{0,\dots,N\}$ et $t \in \{0,\dots,T_n\}$ (on utilise $N+1$ noeuds avec $h = 1/N$ et N_t+1 instants avec $h_t = 1/N_t$), $u_{i,j}^k :\approx u(x_i,y_j,t_k)$ et $f_{i,j}^k :\approx f(x_i,y_j,t_k)$, on obtient le schéma numérique suivant :

$$\boxed{\alpha u_{i,j}^{k+1} + \beta\left(u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right) = u_{i,j}^k + h_t f_{i,j}^{k+1}}. \tag{4.6}$$

Soient $u^k := \begin{pmatrix} u_1^k & \cdots & u_{N-1}^k \end{pmatrix}^T$ (avec $u_j^k := \begin{pmatrix} u_{1,j}^k & \cdots & u_{N-1,j}^k \end{pmatrix}^T$) le vecteur de la solution approchée au temps t^k linéarisé ($u_{0,\cdot}^k = 0, u_{\cdot,0}^k = 0, u_{N,\cdot}^k = 0$, $u_{\cdot,N}^k = 0$ et $u_{\cdot,\cdot}^0$ sont connus) et $f^k := \begin{pmatrix} f_1^k & \cdots & f_{N-1}^k \end{pmatrix}^T$ (avec $f_j^k := \begin{pmatrix} f_{1,j}^k & \cdots & f_{N-1,j}^k \end{pmatrix}^T$) le vecteur du second membre exact au temps t^k linéarisé. Alors la forme matricielle du schéma numérique est la suivante :

$$\boxed{Au^{k+1} = b^k} \tag{4.7}$$

$$\Leftrightarrow \underbrace{\begin{pmatrix} \boxed{M} & \boxed{N} & \cdot & \cdot \\ \boxed{N} & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \boxed{N} \\ \cdot & \cdot & \boxed{N} & \boxed{M} \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} u_1^{k+1} \\ \vdots \\ \vdots \\ u_{N-1}^{k+1} \end{pmatrix}}_{=u^{k+1}} = \underbrace{\begin{pmatrix} b_1^{k+1} \\ \vdots \\ \vdots \\ b_{N-1}^{k+1} \end{pmatrix}}_{=b^{k+1}}$$

avec

$$M := \begin{pmatrix} \alpha & \beta & \cdot & \cdot \\ \beta & \ddots & \ddots & \cdot \\ \cdot & \ddots & \ddots & \beta \\ \cdot & \cdot & \beta & \alpha \end{pmatrix}, \qquad N := \beta I \quad \text{et} \quad b^k := u^k + h_t f^{k+1}.$$

4.1.3 Existence et unicité des solutions approchées

Méthode explicite
Le schéma (4.3) est explicite et dépend de termes connus et définis donc il existe une unique solution approchée.

Méthode implicite

Proposition A est définie-positive et $Au = f$ admet une unique solution.

Démonstration Montrons que A est à diagonale strictement dominante. Soit $i \in \{1, \dots, N-1\}$, alors

$$\sum_{i=1, j\neq i}^{N-1} |a_{i,j}| \in \{2|\beta|, 3|\beta|, 4|\beta|\} \quad \text{et} \quad \max = \{2|\beta|, 3|\beta|, 4|\beta|\} = 4|\beta| = \frac{4ah_t}{h^2}$$

et

$$|a_{i,i}| = |\alpha| = 1 + \frac{4ah_t}{h^2}$$

donc

$$|a_{i,i}| > \sum_{i=1, j\neq i}^{N-1} |a_{i,j}|$$

donc A est à diagonale strictement dominante. A est symétrique donc A est définie-positive donc A est inversible et $Au = f$ admet une unique solution.

4.1.4 Consistance des schémas

Proposition Les schémas (4.3) et (4.7) sont consistants en espace et en temps : $\lim_{h\to 0} |E_h| = 0$ et $\lim_{h_t\to 0} |E_{h_t}| = 0$.

Remarque Les schémas (4.3) et (4.7) sont d'ordre 2 pour x et pour y et d'ordre 1 pour t.

4.1.5 Stabilité et convergence des schémas

Méthode explicite

Proposition *(admise)* Le schéma (4.3) est convergent $\Leftrightarrow \beta \leq \frac{1}{4}$.

Remarques
- On vérifiera cette propriété à la fin de la sous-sous-section 4.2.1 avec un exemple.
- Cette proposition s'appelle la condition de Courant-Friedrich-Levy (CFL).

Méthode implicite

Proposition *(admise)* Soit $(\lambda_i)_{1\leq i\leq N-1}$ l'ensemble des valeurs propres de A avec $\lambda_1 < ... < \lambda_{N-1}$. Alors, $\lambda_1 \geq 1$.

Proposition Si $f \equiv 0$, alors le schéma (4.7) est stable : $\|u^{k+1}\| \leq \|u^0\|$.

Démonstration Soit $\langle u, v\rangle := \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} u_{i,j}v_{i,j}$ une application produit scalaire. Alors :

$$\alpha u_{i,j}^{k+1} + \beta\left(u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right) = u_{i,j}^{k} + h_t f_{i,j}^{k+1}$$

$$\Leftrightarrow \left(\alpha u_{i,j}^{k+1} + \beta\left(u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right)\right) u_{i,j}^{k+1} = \left(u_{i,j}^{k} + h_t f_{i,j}^{k+1}\right) u_{i,j}^{k+1}$$

$$\Leftrightarrow \sum_{i=1}^{N-1}\sum_{j=1}^{N-1}\left(\alpha u_{i,j}^{k+1} + \beta\left(u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right)\right) u_{i,j}^{k+1} = \sum_{i=1}^{N-1}\sum_{j=1}^{N-1}\left(u_{i,j}^{k} + h_t f_{i,j}^{k+1}\right) u_{i,j}^{k+1}$$

et A est définie-positive donc diagonalisable donc

$$\sum_{i=1}^{N-1}\sum_{j=1}^{N-1}\left(\alpha u_{i,j}^{k+1} + \beta\left(u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right)\right) u_{i,j}^{k+1} = \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} Au^{k+1}u^{k+1} = \langle Au^{k+1}, u^{k+1}\rangle$$

$$\geq \underbrace{\lambda_1}_{\geq 0}\|u^{k+1}\|^2$$

et d'après l'inégalité de Cauchy-Schwarz :

$$\sum_{i=1}^{N-1}\sum_{j=1}^{N-1} u_{i,j}^{k}u_{i,j}^{k+1} \leq \|u^k\|\|u^{k+1}\| \quad \text{et} \quad \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} h_t f_{i,j}^{k+1}u_{i,j}^{k+1} \leq h_t\|f^{k+1}\|\|u^{k+1}\|$$

donc

$$\lambda_1\|u^{k+1}\|^2 \leq \|u^{k+1}\|\left(\|u^k\| + h_t\|f^{k+1}\|\right) \Leftrightarrow \|u^{k+1}\|^2 \leq \frac{1}{\lambda_1}\|u^{k+1}\|\left(\|u^k\| + h_t\|f^{k+1}\|\right)$$

$$\Leftrightarrow \|u^{k+1}\| \leq \frac{1}{\lambda_1}\left(\|u^k\| + h_t\|f^{k+1}\|\right) \leq \|u^k\| + h_t\|f^{k+1}\|.$$

Par récurrence, on obtient :

$$\|u^{k+1}\| \leq \|u^0\| + \sum_{i=1}^{k+1}\|f^i\|.$$

Si $f \equiv 0$, alors $\|u^{k+1}\| \leq \|u^0\|$ donc le schéma est stable.

4.2 Implémentation

Pour la suite, la fonction à approcher aura pour terme source $f(x, y, t) = \left(-\lambda + 2a\pi^2\right)\sin(\pi x)\sin(\pi y)\, e^{-\lambda t}$.

4.2.1 Version avec schéma explicite en séquentiel

Implémentation *(version 1 résumée, voir les détails dans `/Probleme-Chaleur/sequentiel-1`)*
Pour commencer, on implémente une version séquentielle du schéma (4.3).
Étapes :

- créer des fonctions qui font le travail :
  - calculer la solution approchée `u` :

    Fonction principale :

```
void calculer_u(double *u){

    double *u_anc; double *permut;
    init_u_zero(u_zero, &u_anc);
    for (int i = 0 ; i < nb_pt * nb_pt ; i ++){
        u[i] = 0.0;
    }

    for (int k = 1 ; k <= N_t ; k ++){
```

```
        # ifdef ECRITURE
        ecrire_double_iteration(u_anc);
        # endif

        for (int j = 1 ; j < nb_pt - 1 ; j ++){
            for (int i = 1 ; i < nb_pt - 1 ; i ++){
                double f = f_source(i * h, j * h, k * h_t);
                u[IDX(i, j)] = schema(f, u_anc, i, j, k);
            }
        }

        permut = u; u = u_anc; u_anc = permut;

    }

    # ifdef ECRITURE
    ecrire_double_iteration(u);
    # endif

    terminaison(&permut, &u, &u_anc);

}
```

Fonction qui applique le schéma à un point :

```
static inline __attribute__((always_inline))
double schema(double *f, double *u_anc, int i, int j, int k){

    double res = alpha * u_anc[IDX(i, j)]
    + beta *
    (u_anc[IDX(i - 1, j)] + u_anc[IDX(i, j - 1)] + u_anc[IDX(i + 1, j)] +
        u_anc[IDX(i, j + 1)])
    + h_t * f[IDX(i, j)];

    return res;

}
```

Commentaires

— La fonction pour le calcul de la solution exacte est `calculer_u_u_exact`.

— Contrairement aux problèmes stationnaires, on n'initialise pas `f` dans `main`. Comme il dépend du temps, on le calcule dans la boucle la plus interne sans le stocker dans un tableau.

— On note ces résultats de $\|e\|_\infty^\infty$ en fonction de h et de h_t (avec $L = 1, T = 1, a = 1$ et $\lambda = 2a\pi^2$) (disponibles aussi dans `Textes/resultats_erreurs.txt`) :

$\downarrow h \quad h_t \rightarrow$	$1/10000$	$1/20000$	$1/50000$	$1/100000$
$1/10$	0.00266777	0.00284805	0.00295614	0.00299216
$1/20$	0.00039394	0.00057533	0.00068410	0.00072034
$1/50$	0.00024223	0.00006052	0.00004843	0.00008473
$1/100$	∞_f	∞_f	0.00004237	0.00000605

On vérifie la proposition énoncée en sous-sous-section 4.1.5, les valeurs ∞_f sont bien atteintes lorsque $\beta > 1/4$ donc la proposition de CFL est vérifiée pour cet exemple.

— Lorsque h/h_t est fixé, le schéma semble bien converger.

— On note ces résultats du temps d'exécution (en s) pour $N = 200$ et $N_t = 160000$ en fonction de l'activation ou non de l'écriture dans un fichier (disponibles aussi dans `Textes/resultats_temps.txt`) :

Mode	Sans écriture	Avec écriture
Temps d'exécution (en s)	52.1	57.8

— La condition sur β est très contraignante : si l'on souhaite diviser par 2 le pas spatial, alors il faut diviser par 4 le pas temporel. Et la constante 1/4 implique que $h_t \leq \frac{1}{4a}h^2$, forçant des pas temporel très petits comparés aux pas spatiaux.

— La complexité algorithmique est $O(N^2 \cdot N_t)$.

4.2.2 Version avec schéma explicite en parallèle avec OpenMP et MPI

Implémentation *(version 2 résumée, voir les détails dans `/Probleme-Chaleur/parallele-1`)*
Ensuite, on implémente une version parallèle avec OpenMP du schéma (4.3).
Étapes :

— créer des fonctions qui font le travail :

— calculer la solution approchée `u` :

Fonction principale :

```
void calculer_u(double *u){

    double *u_anc; double *permut;
    init_u_zero(u_zero, &u_anc);
    for (int i = 0 ; i < nb_pt * nb_pt ; i ++){
        u[i] = 0.0;
    }

    # pragma omp parallel firstprivate(u, u_anc, permut)
    {

        for (int k = 1 ; k <= N_t ; k ++){

            # ifdef ECRITURE
            # pragma omp single
            ecrire_double_iteration(u_anc);
            # endif

            # pragma omp for schedule(static)
            for (int j = 1 ; j < nb_pt - 1 ; j ++){
                for (int i = 1 ; i < nb_pt - 1 ; i ++){
                    double f = f_source(i * h, j * h, k * h_t);
                    u[IDX(i, j)] = schema(f, u_anc, i, j, k);
                }
            }

            permut = u; u = u_anc; u_anc = permut;

        }

        # ifdef ECRITURE
        ecrire_double_iteration(u);
        # endif

        terminaison(&permut, &u, &u_anc);

    }

}
```

Fonction pour terminer :

```
void terminaison(double **permut, double **u, double **u_anc){

    if (N_t % 2 != 0){
        *permut = *u; *u = *u_anc; *u_anc = *permut;
    }

    # pragma omp single
    free(*u_anc);

}
```

Commentaires

— A la différence des implémentations OpenMP des problèmes stationnaires, on définit la zone parallèle (de fork) à l'extérieur des boucles. Les tableaux `u` et `u_anc` sont toujours sur le tas mais chaque thread possède une copie privée des pointeurs. Un seul thread effectue la libération de `u_anc`.

— L'écriture dans un fichier se fait en séquentiel.

— On note ces résultats du temps d'exécution (en s) pour $N = 200$ et $N_t = 160000$ en fonction du nombre de threads de l'activation ou non de l'écriture dans un fichier (disponibles aussi dans `Textes/resultats_temps.txt`) :

↓ Nombre de threads Mode →	Sans écriture	Avec écriture
1	52.0	58.2
2	30.1 1.7	36.4 1.6
4	20.9 2.5	24.3 2.4
6	25.0 2.1	28.5 2.0
8	21.7 2.4	25.6 2.3

Implémentation *(version 3 résumée, voir les détails dans `/Probleme-Chaleur/parallele-2`)*
Ensuite, on implémente une version parallèle avec MPI du schéma (4.3).
Fonction pour écrire u^k dans un fichier en parallèle (qui utilise un type dérivé `vue_fichier`) :

```
static inline __attribute__((always_inline, unused))
void ecrire_double_iteration(double *u, int k){

    uint64_t offset = (uint64_t)k * (uint64_t)nb_pt * (uint64_t)nb_pt * (uint64_t)
        sizeof(double);
    int taille[2] = {nb_pt, nb_pt};
    int sous_taille[2]  = {nb_pt_div_j, nb_pt_div_i};
    int debut[2] = {j_debut, i_debut};

    MPI_Datatype vue_fichier;
    MPI_Type_create_subarray(2, taille, sous_taille, debut, MPI_ORDER_C, MPI_DOUBLE, &
        vue_fichier);
    MPI_Type_commit(&vue_fichier);

    MPI_File_set_view(descripteur, offset, MPI_DOUBLE, vue_fichier, "native",
        MPI_INFO_NULL);

    MPI_File_write_all(descripteur, u, 1, bloc_send, MPI_STATUS_IGNORE);

    MPI_Type_free(&vue_fichier);

}
```

Commentaires

— Cette implémentation reprend exactement le même méthode que pour `/Probleme-2D/parallele-2` en adaptant le schéma et le calcul de f (voir la fonction `calculer_u`).

— Pour le calcul de la solution exacte (en séquentiel), on réutilise la fonction `regrouper_u` (voir la fonction `calculer_u_u_exact`.

— On note ces résultats du temps d'exécution (en s) pour $N = 200$ et $N_t = 160000$ en fonction du nombre de processus de l'activation ou non de l'écriture dans un fichier (disponibles aussi dans `Textes/resultats_temps.txt`) :

↓ Nombre de processus Mode →	Sans écriture	Avec écriture
1	53.3	98.3
2	33.7 1.6	63.7 1.5
4	25.9 2.1	74.2 1.3
6	35.4 1.5	88.5 1.1
8	38.4 1.4	132.0 0.7

4.2.3 Version avec schéma implicite en séquentiel

Implémentation *(version 4 résumée, voir les détails dans `/Probleme-Chaleur/sequentiel-2`)*
Enfin, on implémente une version séquentielle du schéma (4.7) avec la librairie `cholmod`.
Fonction principale :

```
void resoudre(cholmod_sparse *A, double *u){

    double *b_int = (double *)malloc(idx_max * sizeof(double));
    double *u_int = (double *)malloc(idx_max * sizeof(double));

    init_u_zero(u_zero, u);

    cholmod_dense *b_dense = cholmod_allocate_dense(A -> nrow, 1, A -> nrow,
        CHOLMOD_REAL, &c);
    cholmod_dense *u_dense;
    cholmod_factor *L = cholmod_analyze(A, &c);
    cholmod_factorize(A, L, &c);

    for (int k = 1 ; k <= N_t ; k ++){

        # ifdef ECRITURE
        ecrire_double_iteration(u);
        # endif

        extraire_interieur(u, u_int, nb_pt);
        calculer_b(k + 1, u_int, b_int);

        memcpy(b_dense -> x, b_int, A -> nrow * sizeof(double));

        u_dense = cholmod_solve(CHOLMOD_A, L, b_dense, &c);

        memcpy(u_int, u_dense -> x, A -> nrow * sizeof(double));

        inserer_interieur(u_int, u, nb_pt);

    }

    # ifdef ECRITURE
    ecrire_double_iteration(u);
    # endif
```

```
    cholmod_free_factor(&L, &c);
    cholmod_free_dense(&b_dense, &c);
    cholmod_free_dense(&u_dense, &c);
    free(b_int);
    free(u_int);


}
```

Fonction pour calculer b^k :

```
static inline __attribute__((always_inline))
void calculer_b(double t, double *u, double *b){

    for (int i = 0 ; i < idx_max ; i ++){
        int x = i % (N - 1);
        int y = i / (N - 1);
        b[i] = u[i] + h_t * f_source(x, y, t + 1);
    }

}
```

Commentaires

— On note ces résultats de $\|e\|_\infty^\infty$ en fonction de h et de h_t (avec $L = 1, T = 1, a = 1$ et $\lambda = 2a\pi^2$) (disponibles aussi dans `Textes/resultats_erreurs.txt`) :

$\downarrow h \quad h_t \rightarrow$	1/10000	1/20000	1/50000	1/100000
1/10	0.00338798	0.00320815	0.00310018	0.00306418
1/20	0.00111862	0.00093767	0.00082903	0.00079281
1/50	0.00048370	0.00030244	0.00019361	0.00015732
1/100	0.00039301	0.00021171	0.00010286	0.00006656

— On note ces résultats du temps d'exécution (en s) en fonction de N (avec $N_t = N$) et de l'activation ou non de l'écriture dans un fichier (disponibles aussi dans `Textes/resultats_temps.txt`) :

$\downarrow N(= N_t) \quad$ Mode $\rightarrow$	Sans écriture	Avec écriture
200	0.3	0.4
400	3.3	3.5
600	12.6	12.3
800	32.0	29.2
1000	60.4	58.4
1200	97.7	99.0
1400	159.1	162.8

— Une exécution avec des pas de temps très petit comme dans le schéma (4.3) donne des temps trop élevés. Heureusement, la stabilité du schéma (4.7) est inconditionnelle, ce qui permet d'exécuter sur des pas de temps plus grands.

— Il y a la possibilité de paralléliser certaines parties du code (hors calcul principal).

4.2.4 Comparaison des performances des méthodes

— La version avec le schéma explicite, en particulier avec le parallélisme, donnent de bons résultats mais sont contraignants sur le taille des pas. Le parallélisme est moins efficace que pour le `Probleme-2D`.

— La version avec le schéma implicite est plus lente mais moins contraignante sur la taille des pas.

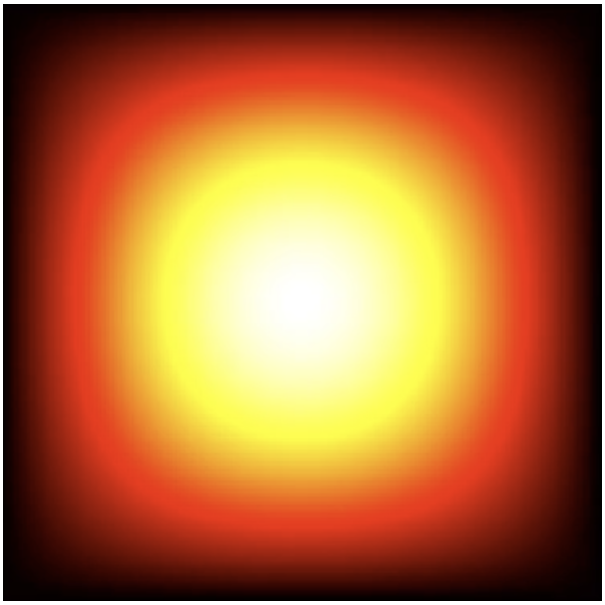
— La version avec la bibliothèque `cholmod` donne d'excellents résultats.

— L'écriture dans un fichier n'augmente pas énormément le temps sauf pour la version MPI.

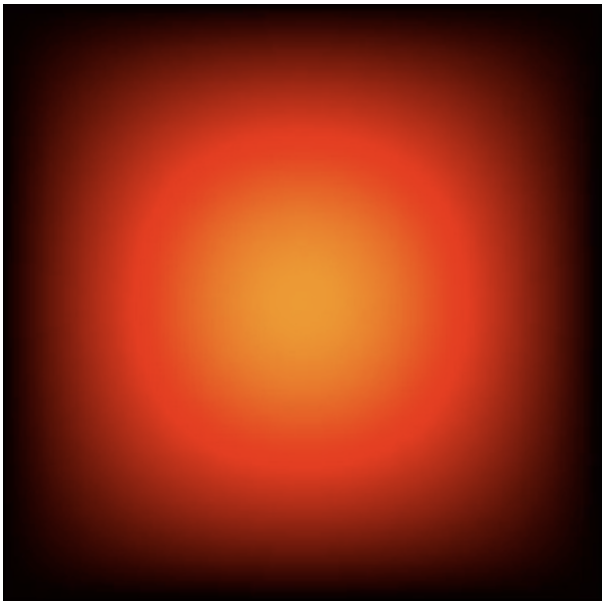
4.3 Visualisation

Enfin, on peut visualiser la diffusion avec Python et matplotlib.

Illustration Diffusion de la chaleur



État initial



État après diffusion