Projet : méthode itérative des directions alternées (ADI)

Nathan Toussaint

20 janvier 2024

L'implémentation complète des différentes fonctions et des tests unitaires se trouvent dans les répertoires de mon projet que j'ai nommés Q1 à Q6.

1 Question 1

La fonction `Vec* buildVec(double val, int nx, int ny)` permet de construire un vecteur de réels double précision de taille $n_x \times n_y$. Elle le crée de manière dynamique et retourne un pointeur pt de type `Vec*`. On alloue d'abord pt puis on procède le pointeur membre $pt->val$ pointant sur l'espace mémoire où va être stocké les $n_x \times n_y$ valeurs réelles. On remarquera qu'à chaque appel de *malloc*, on vérifie que l'allocation s'est bien déroulée sinon dans le cas contraire, on arrête le programme. On remplit le tableau avec la même valeur val.

La fonction `void destroyVec(Vec* pt)` désalloue l'espace mémoire, on libère d'abord l'espace pris par le tableau de réels d'adresse mémoire $pt \to val$ puis on désalloue le pointeur pt.

La fonction `Tridiag* buildM(double omega, int nl)` permet de construire de manière dynamique une matrice tridiagonale sous la forme de 3 vecteurs réels de taille nl. Elle retourne un pointeur pt de type `*Tridiag`. On alloue d'abord pt puis on procède à l'allocation des 3 tableaux $pt \to l$, $pt \to d$ et $pt \to u$. Comme pour les vecteurs, on vérifie que l'allocation s'est bien passée. On remplit ensuite le tableau grâce à une boucle pour construire $M(\omega)$. On annule les composantes non pertinentes : la première composante de $pt \to l$ et la dernière de $pt \to u$.

La fonction `void destroyM(Tridiag *pt)` désalloue l'espace mémoire, on libère d'abord l'espace pris par les tableaux de réels d'adresses mémoires $pt \to l$, $pt \to d$ et $pt \to u$ puis on désalloue le pointeur pt.

2 Question 2

Dans une première partie, je décris l'algorithme de la décomposition LU d'une matrice tridiagonale T stockée sous la forme de 3 vecteurs. Puis dans une seconde partie, je parle de son implémentation en C.

2.1 Algorithme de la décomposition LU d'une matrice tri-diagonale T

On considère une matrice tridiagonale T de dimension $N \times N$:

$$T = \begin{pmatrix} d_0 & u_0 & 0 & \cdots & 0 \\ l_1 & d_1 & u_1 & \ddots & \vdots \\ 0 & l_2 & d_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & u_{n-2} \\ 0 & \cdots & 0 & l_{n-1} & d_{n-1} \end{pmatrix} \tag{1}$$

dont les termes diagonaux sont supposés non nuls.

On la stocke en mémoire sous la forme de 3 tableaux de même taille N dont les indices commencent à 0 pour faciliter l'implémenation en C. La diagonale de T est stockée dans le tableau d,

la première sous-diagonale dans l et la première sur-diagonale dans u. On note que les composantes 0 du vecteur l et la composante $n-1$ du vecteur u ne sont pas pertinentes.

La décomposition LU consiste à écrire la matrice tridiagonale T sous la forme d'un produit LU où

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ \beta_1 & 1 & 0 & \ddots & \vdots \\ 0 & \beta_2 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \beta_{n-1} & 1 \end{pmatrix} \quad \text{et} \quad U = \begin{pmatrix} \alpha_0 & u_0 & 0 & \cdots & 0 \\ 0 & \alpha_1 & u_1 & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & 0 & \alpha_{n-2} & u_{n-2} \\ 0 & \cdots & 0 & 0 & \alpha_{n-1} \end{pmatrix} \tag{2}$$

Comme la première sur-diagonale supérieure de U s'identifie à celle de la matrice T, l'algorithme de décomposition LU se simplifie grandement. Une première version s'écrit :

1: **function** LU(T)
2: $\quad \alpha_0 \leftarrow d_0$
3: $\quad$ **for** $k = 1..(n-1)$ **do**
4: $\quad\quad \beta_k \leftarrow l_k / \alpha_{k-1}$
5: $\quad\quad \alpha_k \leftarrow d_k - u_{k-1}\,\beta_k$
6: $\quad$ **end for**
7: **end function**

Grâce à la structure tri-diagonale de T, il n'est pas nécessaire d'allouer une matrice supplémentaire pour stocker la décomposition LU. Elle peut remplacer progressivement la matrice T ligne par ligne. L'algorithme optimisé en place mémoire s'écrit finalement :

1: **function** LU(T) ▷ à la place de la matrice tridiagonale T
2: $\quad$ **for** $k = 1..(n-1)$ **do**
3: $\quad\quad l_k \leftarrow l_k / d_{k-1}$
4: $\quad\quad d_k \leftarrow d_k - u_{k-1}\,l_k$
5: $\quad$ **end for**
6: **end function**

On note que la complexité de la décomposition LU pour une matrice tri-diagonale de rang N est $\vartheta(N)$. Elle est donc peu couteuse par rapport à celle en $\vartheta(N^3)$ pour une matrice pleine de rang N.

2.2 Implémentation

Listing 1 – Construction de la décomposition LU

```
#include "main.h"

/* Construction de la decomposition LU a la place de la matrice
Entree : Tridiag *ptM
Sortie : vide
*/
void invM(Tridiag *ptM){
   int nl=ptM->nl;
   // l[0] (non pertinent) , d[0] et u[0] sont conserves
   for (int k=1; k<nl; k++){
       // l[k] = l[k]/d[k-1]
       ptM->l[k] =ptM->l[k]/ptM->d[k-1];
       // d[k] = d[k]-u[k-1]*l[k]
       ptM->d[k] =ptM->d[k]-ptM->u[k-1]*ptM->l[k];
       }
}
```

3 Question 3

On suppose que la matrice T a été décomposée en un produit LU où L est une matrice bi-diagonale inférieure avec des 1 sur la diagonale et U est une matrice bi-diagonale supérieure dont les termes diagonaux sont non nuls.

Résoudre $LUx = b$ s'effectue en deux étapes : i) résolution de $Ly = b$ par la méthode de descente pui ii) résolution de $Ux = y$ par la méthode de montée.

3.1 Résolution de $Ly = b$ par la méthode de descente

Soit le système d'équations linéaires où la matrice associée est bi-diagonale inférieure :

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_1 & 1 & 0 & \ddots & \vdots \\ 0 & l_2 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & l_{n-1} & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_{n-1} \end{pmatrix} \tag{3}$$

La méthode de résolution est celle de la descente. De nouveau, dans le cas de matrice bi-diagonale, l'algorithme se simplifie et s'écrit :

1: **function** DESCENTE(LU, b)
2: $y_0 \leftarrow b_0$
3: **for** $k = 1..(n-1)$ **do**
4: $y_k \leftarrow b_k - l_k\, y_{k-1}$
5: **end for**
6: **end function**

La complexité de cet algorithme est $\vartheta(N)$.

3.2 Résolution de $Ux = y$ par la méthode de montée

Soit le système d'équations linéaires où la matrice associée est bi-diagonale supérieure :

$$U = \begin{pmatrix} d_0 & u_0 & 0 & \cdots & 0 \\ 0 & d_1 & u_1 & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & 0 & d_{n-2} & u_{n-2} \\ 0 & \cdots & 0 & 0 & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} \tag{4}$$

L'algorithme de la méthode de la montée s'écrit dans notre cas :

1: **function** MONTEE(LU, y)
2: $x_{n-1} \leftarrow y_{n-1}/d_{n-1}$
3: **for** $i = (n-2)..1$ **do**
4: $x_k \leftarrow (y_k - u_k\, x_{k+1})/d_k$
5: **end for**
6: **end function**

La complexité de cet algorithme est $\vartheta(N)$.

3.3 implémentation de la méthode de résolution

L'implémentation se trouve dans le répertoire $Q3/0\_solveLU$. On a mis en oeuvre le cas test suivant (voir main.c) :

1. Création dynamique d'une matrice $M(\omega = 1)$ de rang $nl = 6$.
2. Création dynamique de deux vecteurs RHS rempli de 1 et X rempli de 0, de dimension $nl = 6$.

3. Résolution de $MX = RHS$.
4. Comme la décomposition LU remplace les valeurs de M. Il est nécessaire de le détruire et de le reconstruire.
5. Vérification que $||RHS - MX||_\infty \to 0$.

3.4 Méthode de résolution ADI

On cherche à résoudre en 2D, l'équation de Poisson : $-\Delta u(x,y) = f(x,y)$ sur un domaine carré $\Omega =]0,1[^2$. Elle est discrétisée en différences finies sur une grille est de taille $n_x \times n_y$. On impose des conditions de Dirichlet sur le contour $\partial\Omega$ épousant le contour de la grille : $u(x,y) = u^d(x,y)$ (*notation différente du sujet pour réduire les ambiguités de notations avec les indices*).

3.5 Discrétisation du domaine carré

On repère chaque noeud de la grille par ses coordonnées entières (i,j) comme dans l'exemple ci-dessous d'une grille 5×4 :

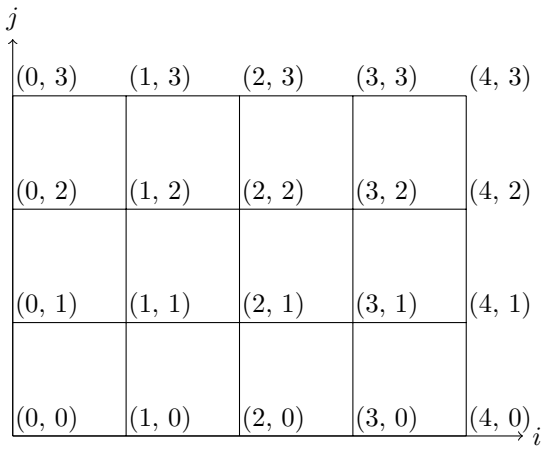

TABLE 1 – coordonnées entières des noeuds de la grille 5×4

Dans l'approximation des différences finies, on définit :

$$\hat{L}_x u\big|_{(i,j)} = -\partial_x^2 u\big|_{(i,j)} = \frac{-u(i-1,j) + 2u(i,j) - u(i+1,j)}{h_x^2} + \vartheta(h_x^2) \tag{5}$$

et

$$\hat{L}_y u\big|_{(i,j)} = -\partial_y^2 u\big|_{(i,j)} = \frac{-u(i,j-1) + 2u(i,j) - u(i,j+1)}{h_y^2} + \vartheta(h_y^2) \tag{6}$$

où $h_x = \frac{1}{n_x - 1}$ et $h_y = \frac{1}{n_y - 1}$ sont les pas de la grille.
Dans la suite, on introduit les notations $m_x = h_x^{-1} = n_x - 1$ et $m_y = h_y^{-1} = n_y - 1$.
Notons que la fonction $f(x,y)$ est évaluée aux noeuds intérieurs (i,j) de la grille.

3.6 Les pas fractionnaires de la méthode

La méthode de résolution ADI est une méthode itérative composée de deux pas fractionnaires. Elle se caractérise par une approche qui alterne entre les directions x et y, en résolvant chaque sous-problème de manière implicite :

$$\begin{cases} (L_x + \omega_k I)u^* &= -(L_y - \omega_k I)u^k + b \\ (L_y + \omega_k I)u^{k+1} &= -(L_x - \omega_k I)u^* + b \end{cases} \tag{7}$$

Cette alternance permet de transformer la résolution de l'équation bidimensionnelle en deux problèmes unidimensionnels plus simples, qui peuvent être résolus efficacement. En pratique, la résolution s'arrête lorsque $\|(L_x + L_y)\, u^k - b\|_\infty < \epsilon$ avec $\epsilon \approx 10^{-6}$ par exemple, voire plus petit.

3.7 Application à la grille 5×4

On propose de détailler la méthode ADI dans le cas d'une grille de taille $n_x \times n_y = 5 \times 4$. Le premier pas fractionnaire de la méthode ADI consiste à résoudre, pour chacune des lignes $j \in]0, n_y - 1[= \{1, 2\}$. Pour $i \in [1, 3]$, il s'écrit :

$$-m_x^2 u^*_{i-1,j} + (2m_x^2 + \omega)u^*_{i,j} - m_x^2 u^*_{i+1,j} = m_y^2\, u^k_{1,j-1} - (2m_y^2 - \omega)\, u^k_{1,j} + m_y^2\, u^k_{1,j+1} + b_{i,j} \tag{8}$$

Viennent ensuite s'ajouter les conditions de dirichlet en $i = 0$ et $i = n_x - 1$. De manière matricielle, le système d'équations à résoudre s'écrit :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -m_x^2 & 2m_x^2 + \omega & -m_x^2 & 0 & 0 \\ 0 & -m_x^2 & 2m_x^2 + \omega & -m_x^2 & 0 \\ 0 & 0 & -m_x^2 & 2m_x^2 + \omega & -m_x^2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u^*_{0,j} \\ u^*_{1,j} \\ u^*_{2,j} \\ u^*_{3,j} \\ u^*_{4,j} \end{pmatrix} = \begin{pmatrix} 0 \\ m_y^2\, u^k_{1,j-1} - (2m_y^2 - \omega)\, u^k_{1,j} + m_y^2\, u^k_{1,j+1} \\ m_y^2\, u^k_{2,j-1} - (2m_y^2 - \omega)\, u^k_{2,j} + m_y^2\, u^k_{2,j+1} \\ m_y^2\, u^k_{3,j-1} - (2m_y^2 - \omega)\, u^k_{3,j} + m_y^2\, u^k_{3,j+1} \\ 0 \end{pmatrix} + \begin{pmatrix} u^d_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \\ u^d_{4,j} \end{pmatrix} \tag{9}$$

où $u^d_{i,j}$ est la valeur de dirichlet au noeud de coordonnées (i, j). Cette première étape revient à résoudre $n_y - 2$ petits systèmes linéaires dont la matrice est tri-diagonale de rang n_x.

De manière similaire, le second pas fractionnaire revient à résoudre, pour chacune des colonnes $i \in]0, n_x - 1[= \{1, 2, 3\}$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -m_y^2 & 2m_y^2 + \omega & -m_y^2 & 0 \\ 0 & -m_y^2 & 2m_y^2 + \omega & -m_y^2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u^{k+1}_{i,0} \\ u^{k+1}_{i,1} \\ u^{k+1}_{i,2} \\ u^{k+1}_{i,3} \end{pmatrix} = \begin{pmatrix} 0 \\ m_x^2\, u^*_{i-1,1} - (2m_x^2 - \omega)\, u^*_{i,1} + m_x^2\, u^*_{i+1,1} \\ m_y^2\, u^*_{i-1,2} - (2m_x^2 - \omega)\, u^*_{i,2} + m_y^2\, u^*_{i+1,2} \\ 0 \end{pmatrix} + \begin{pmatrix} u^d_{i,0} \\ b_{i,1} \\ b_{i,2} \\ u^d_{i,4} \end{pmatrix} \tag{10}$$

De même, cette seconde étape revient à résoudre $n_x - 2$ petits systèmes linéaires dont la matrice est tri-diagonale de rang n_y.

Dans l'implémentation, on ne traitera que le cas où les valeurs de u au bord sont strictement nulles.

3.8 Stockage des valeurs aux noeuds sous la forme d'un vecteur

Du point de vue de l'implémentation, on stockera les approximations de la solution dans un tableau unidimensionnel de taille $n_x \times n_y$ en utilisant la relation $n(i, j) = i + j\, n_x$ qui numérote de manière unique chaque noeud (i, j).

3.9 Implémentation de computeRHS

L'implémentation se trouve dans le répertoire `Q3/1_computeRHS_solve`. Je mets en oeuvre la présentation de la méthode ADI faite dans une précédente section.

La fonction `void computeRHSX(int j, double omega, Vec *ptX, Vec *ptB, Vec *ptRHS)` construit le second membre pour la ligne j, argument supplémentaire que j'ai ajouté par rapport à l'énoncé, pour le premier pas fractionnaire.

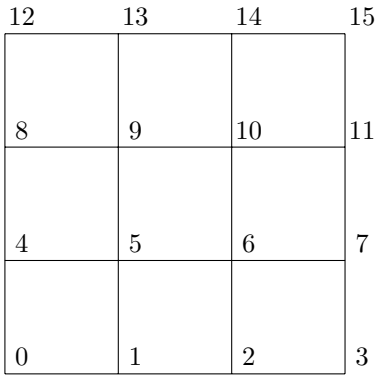

TABLE 2 – Numérotation unique des noeuds de la grille 4×4

La fonction `void computeRHSY(int i, double omega, Vec *ptX, Vec *ptB, Vec *ptRHS)` fait de même pour la colonne i.

La programme principal `main.c` montre l'enchainement des 2 pas fractionnaires. Pour chacune des lignes (resp. chacune des colonnes), on a

1. Contruction de la matrice M et du second membre.
2. Résolution du système tridiagonale associé.
3. Mise à jour de la solution intermédiaire.
4. Libération de l'espace mémoire.

Listing 2 – Appel des fonctions computeRHSX et computeRHSY

```
    int iter=0;
    for(;;){
    // premier pas fractionnaire selon Ox
        for (int j=1; j<Ny-1; j++){
            Tridiag *ptM     = buildM(omega, Nx);
            Vec *ptRHS = buildVec(0.0, Nx, 1);

/*          compute RHSX */
            computeRHSX(j, omega, u, ptB, ptRHS);

            Vec *ptX = buildVec(0.0, Nx, 1);
            solve(ptM, ptRHS, ptX);

            for (int i=0; i<Nx; i++){
                int n = idx(i,j);
                u_star->val[n] = ptX->val[i];
                }//endfor i
            destroyVec(ptX);
            destroyVec(ptRHS);
            destroyM(ptM);
            }//endfor j

    // second pas fractionnaire selon Oy
        for (int i=1; i<Nx-1; i++){
            Tridiag *ptM = buildM(omega, Ny);
            Vec *ptRHS = buildVec(0.0, 1, Ny);

/*          compute RHSY */
```

```
            computeRHSY(i, omega, u_star, ptB, ptRHS);

            Vec *ptX = buildVec(0.0, 1, Ny);
            solve(ptM, ptRHS, ptX);

            for (int j=0; j<Ny; j++){
                int n = idx(i,j);
                u->val[n] = ptX->val[j];
                }//endfor j
            destroyVec(ptX);
            destroyVec(ptRHS);
            destroyM(ptM);
            }//endfor i

        iter++;
        printf("iter %d  residu %g \n", iter, norminf(u, ptB));
        if (norminf(u, ptB)<tolerance) break;
    } //endfor ever
```

4 Convergence

On note $\bar{u}$ la solution de $(L_x + L_y)\bar{u} = b$. On définit les erreurs $\epsilon^k = u^k - \bar{u}$ et $\epsilon^* = u^* - \bar{u}$. En soustrayant $(L_x + L_y)\bar{u} = b$ des équations (7), on obtient :

$$\begin{cases} (L_x + \omega_k I)\epsilon^* &= -(L_y - \omega_k I)\epsilon^k \\ (L_y + \omega_k I)\epsilon^{k+1} &= -(L_x - \omega_k I)\epsilon^* \end{cases} \tag{11}$$

Après élimination de e^*, on obtient :

$$\epsilon^{k+1} = (L_y + \omega I)^{-1}(L_x - \omega I)(L_x + \omega I)^{-1}(L_y - \omega I)\,\epsilon^k = T(\omega)\epsilon^k \tag{12}$$

où

$$T(\omega) = (L_y + \omega I)^{-1}(L_x - \omega I)(L_x + \omega I)^{-1}(L_y - \omega I) \tag{13}$$

A titre d'exemple, prenons $n_x = n_y = 4$, les matrices L_x et L_y en tenant compte des conditions de dirichlet sur les bords sont de la forme :

$L_x =$

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10	11	12	13	14	15
0	1															
1		1														
2			1													
3				1												
4					1											
5					$-m_x^2$	$2m_x^2$	$-m_x^2$									
6						$-m_x^2$	$2m_x^2$	$-m_x^2$								
7								1								
8									1							
9									$-m_x^2$	$2m_x^2$	$-m_x^2$					
10										$-m_x^2$	$2m_x^2$	$-m_x^2$				
11												1				
12													1			
13														1		
14															1	
15																1

$L_y =$

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10	11	12	13	14	15
0	1															
1		1														
2			1													
3				1												
4					1											
5		$-m_y^2$				$2m_y^2$				$-m_y^2$						
6			$-m_y^2$				$2m_y^2$				$-m_y^2$					
7								1								
8									1							
9						$-m_y^2$				$2m_y^2$				$-m_y^2$		
10							$-m_y^2$				$2m_y^2$				$-m_y^2$	
11												1				
12													1			
13														1		
14															1	
15																1

On remarque que L_x commute avec L_y donc elles possèdent la même base de vecteurs propres. Leurs valeurs propres **ne sont identiques pas** dans le cas générale sauf dans le cas où $n_x = n_y$.

On en déduit que $[(L_y + \omega I)^{-1}, (L_x - \omega I)] = 0$ ainsi que $[(L_x + \omega I)^{-1}, (L_y - \omega I)] = 0$. On peut donc arranger les différents produits matriciels dans l'expression 13 et on obtient la nouvelle expression suivante :

$$T(\omega) = (L_x + \omega I)^{-1}(L_x - \omega I)(L_y + \omega I)^{-1}(L_y - \omega I) \tag{14}$$

Les valeurs propres de $T(\omega)$ valent $\frac{\lambda_n - \omega}{\lambda_n + \omega}\frac{\gamma_n - \omega}{\gamma_n + \omega}$ et son rayon spectral vaut $\max_{\lambda_n, \gamma_n} \left|\frac{\lambda_n - \omega}{\lambda_n + \omega} \times \frac{\gamma_n - \omega}{\gamma_n + \omega}\right|$. Pour $\omega > 0$, la fonction $\varphi(\lambda) = \frac{\lambda - \omega}{\lambda + \omega}$ est croissante (voir Fig. 1).

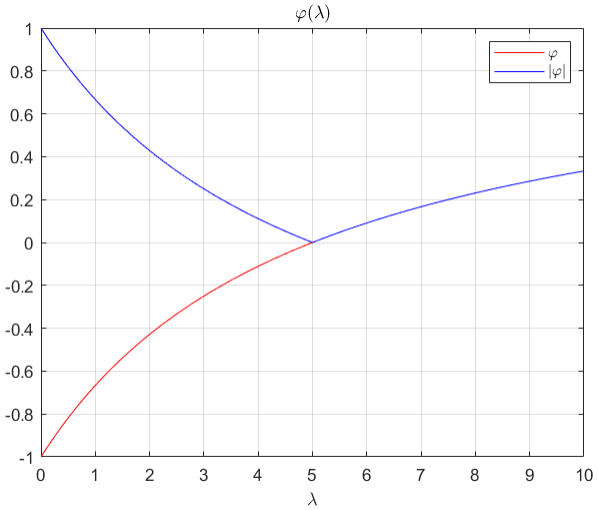

FIGURE 1 – Evolution de φ et $|\varphi|$ avec λ

D'après Lascaux et Théodore (Tome 2, page 386), le choix optimal de ω est celui qui minimise $\Psi(\omega) = \max\{\frac{\lambda_1 - \omega}{\lambda_1 + \omega}, \frac{\lambda_M - \omega}{\lambda_M + \omega}\}$ où $\lambda = 1$ et $\lambda_M \approx 4m^2$.
On en déduit que $\omega_{opt} = \sqrt{\lambda_1 \lambda_M} \approx 2m$ (voir Fig. 2).

Dans le cas général où $n_x \neq n_y$, on a deux paramètres optimaux qui valent approximativement $\omega_x = 2m_x$ et $\omega_y = 2m_y$.

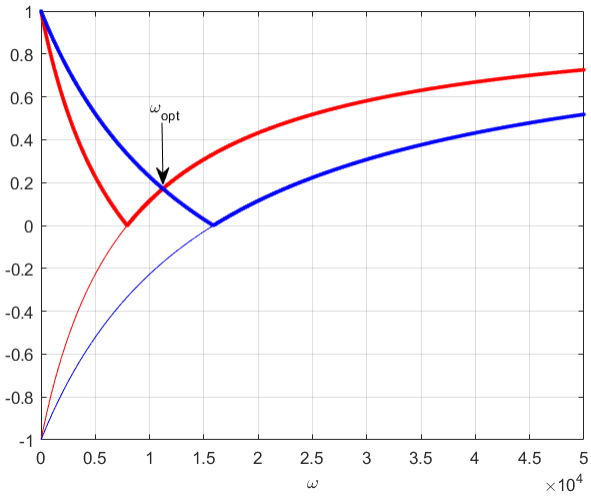

FIGURE 2 – Evolution de φ et $|\varphi|$ avec ω

5 Résultats numériques

5.1 Convergence

Le système est discrétisé avec une grille $n_x = 64 \times n_y = 64$. Le paramètre de relaxation ω a été fixé arbitrairement à 256 et le seuil de tolérance à $\epsilon = 10^{-8}$.

La figure 3 représente l'évolution du résidu défini par $||b - Au||_\infty$ en fonction du nombre d'itérations en échelle semi-logarithmique base 10. Le résidu décroit de manière exponentielle avec deux régimes différents.

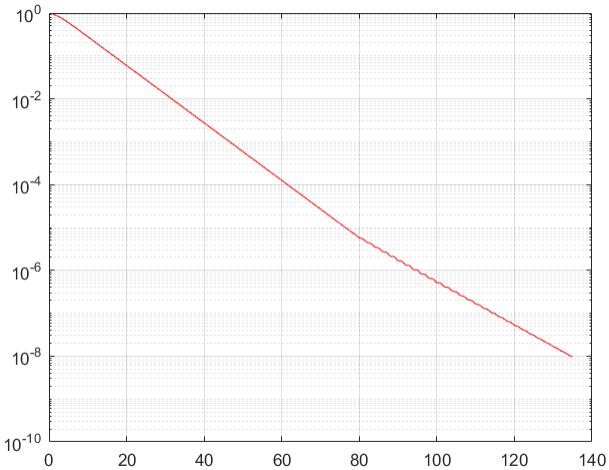

FIGURE 3 – Evolution du résidu avec le nombre d'itérations

5.2 Coefficients optimaux

5.3 Deux cas tests

On n'a traité que des problèmes où la condition de dirichlet aux bords est $u^d = 0$.

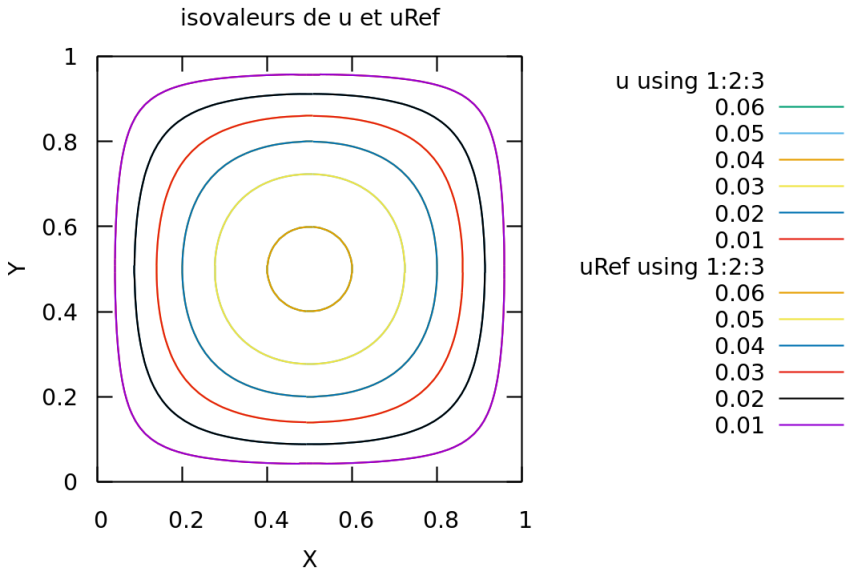

FIGURE 4 – Isovaleurs de $u(x, y)$

Le répertoire `Q6/0_ADI2D` contient le premier cas test où $f(x, y) = 2x(1 - x) + 2y(1 - y)$ s'annulant sur $\partial\Omega$. La solution analytique vérifiant $\Delta u + f = 0$ est $u_{ref}(x, y) = x(1 - x)y(1 - y)$. La figure 3 représente les isovaleurs de la solution numérique $u(x, y)$ et de la solution de référence $u_{ref}(x, y)$. L'accord est excellent.

Le répertoire `Q6/1_ADI2D` contient le second cas test où $f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y)$ qui s'annule sur $\partial\Omega$. La solution analytique est $u_{ref}(x, y) = \sin(\pi x) \sin(\pi y)$. L'accord est tout aussi parfait.

Listing 3 – Script gnuplot pour visualiser les isovaleurs

```
# Nom du fichier de donnees
u = "out"
uRef = "ref"

# Parametres du trace
set contour base
set view map
unset surface

# Rapport d'aspect egal
set size ratio -1

# Nombre d'isovaleurs souhaitees
set cntrparam levels 11

# Etiquettes et titre
set xlabel "X"
set ylabel "Y"
set title "isovaleurs de u et uRef"

# Placer les legendes en dehors de la figure, a droite
set key outside right
```

```
# Trace des isovaleurs
splot u using 1:2:3 with lines
replot uRef using 1:2:3 with lines

# Affichage du graphique
pause mouse keypress "Appuyez sur une touche pour quitter"
```

5.4 Intérêt de la méthode ADI

La complexité de la méthode de résolution d'un système tri-diagonale de rang N en utilisant la méthode de décomposition LU est $\vartheta(N)$ et est peu couteuse par rapport à la méthode de résolution de Gauss qui a une complexité $\vartheta(N^3)$ pour une matrice dense de rang N.

Calculons maintenant la complexité des deux pas fractionnaires de la méthode ADI pour une grille de taille $n_x \times n_y$.

La première étape revient à résoudre $n_y - 2$ petits systèmes linéaires dont la matrice est tri-diagonale de rang n_x. Sa complexité est donc en $\vartheta(n_y \times n_x)$ en ne gardant que l'ordre le plus élevé.

La seconde étape revient à résoudre $n_x - 2$ petits systèmes linéaires dont la matrice est tri-diagonale de rang n_y. Sa complexité est donc en $\vartheta(n_x \times n_y)$.

La complexité de la méthode ADI est donc $\vartheta(n_x \times n_y)$.

5.5 Détermination numérique des paramètres optimaux

J'ai choisi de prendre $n_x = n_y = 64$ et une tolérance $= 10^{-6}$. J'ai repris la fonction polynomiale $f(x, y)$ du premier cas test.

L'évolution du nombre d'itérations en fonction de $\omega \in [64, 464]$ obtenue avec le code `ADID_param_optim` est représenté sur la figure 5.

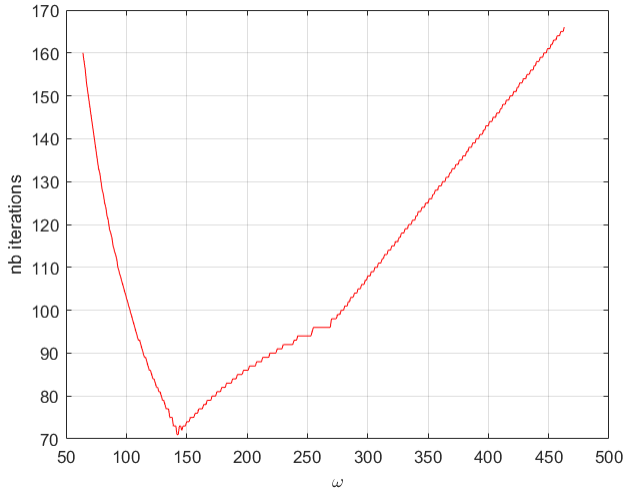

FIGURE 5 – nombre d'itérations en fonction de ω

On obtient une valeur numérique de $\omega_{opt} = 142$ supérieure de 10% à celle estimée théoriquement.