

# Approximation par différences finies 2D

Loubna TALEB

02 Février 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Structure du Code</b>	<b>2</b>
2.1	Définitions des fonctions : . . . . .	3
2.2	Modèles testés: . . . . .	3
<b>3</b>	<b>Résultats :</b>	<b>4</b>
3.1	Jacobi: . . . . .	4
3.2	Gauss-Seidel: . . . . .	6
3.3	Euler-implicite: . . . . .	6
3.4	Crank-Nicolson: . . . . .	7
<b>4</b>	<b>Conclusion:</b>	<b>9</b>

## 1 Introduction

L'objectif du TP est de programmer l'équation différentielle suivante en espace-temps du problème suivant sur  $\omega = [a_x, b_x] \times [a_y, b_y]$  :

$$\begin{cases} \frac{\partial u(x, t)}{\partial t} - v \frac{\partial u(x, t)}{\partial x} = f(x, t) & \text{dans } \Omega \times [0, T] \\ u(x, t) = g(x, t) & \text{sur le bord } \partial\Omega \times (0, T) \\ u(x, 0) = u_0(x) & \text{sur } \Omega \end{cases}$$

telles que  $f$ ,  $g$  et  $u_0$  sont les données du problème. Pour ce faire, nous avons discrétisé l'équation en temps en nous basant sur Euler implicite et Crank-Nicolson. Pour  $n > 1$ , nous avons :

**Euler implicite :**

$$\frac{1}{h_t} u^n(x) - \mu \Delta u^n(x) = f(x) + \frac{1}{h_t} u^{n-1}(x)$$

**Crank-Nicolson :**

$$\frac{1}{h_t}u^n(x) - \frac{\mu}{2}\Delta u^n(x) = f(x) + \frac{1}{h_t}u^{n-1}(x) + \frac{\mu}{2}\Delta u^{n-1}(x)$$

pour  $h_t > 0$  et  $t_n = nh_t$ , où  $u^n(x) = u(x, t_n)$ .

Ensuite, nous avons discrétisé ces équations en espace en nous basant sur la méthode de Jacobi et Gauss-Seidel. En effet, les deux équations précédentes pour Euler implicite et Crank-Nicolson donnent l'équation suivante :

$$\alpha u^n(x) + \mu \Delta u^n(x) = f^n(x) \text{ dans } \Omega$$

$$u^n = g^n \text{ sur le bord } \partial\Omega$$

Donc, en appliquant l'approximation en 5 points, on trouve les deux méthodes itératives sans utiliser explicitement la matrice sous-jacente. Ainsi, pour :

**Méthode de Jacobi :**

$$u_{i,j}^{(k+1)} = \frac{1}{\lambda} \left[ f_{i,j} + \frac{\mu}{h_x^2}(u_{i-1,j}^k + u_{i+1,j}^k) + \frac{\mu}{h_y^2}(u_{i,j-1}^k + u_{i,j+1}^k) \right]$$

où  $\lambda = \alpha + \frac{2\mu}{h_x^2} + \frac{2\mu}{h_y^2}$ .

**Méthode de Gauss-Seidel :**

$$u_{i,j}^{(k+1)} = \frac{1}{\lambda} \left[ f_{i,j} + \frac{\mu}{h_x^2}(u_{i-1,j}^{k+1} + u_{i+1,j}^k) + \frac{\mu}{h_y^2}(u_{i,j-1}^{k+1} + u_{i,j+1}^k) \right]$$

## 2 Structure du Code

Nous avons créé trois fichiers : 'Functions.h' pour les définitions des fonctions et des variables globales, 'Functions.c' pour l'implémentation de ces fonctions, et 'main.c' pour calculer le temps d'exécution de chaque fonction et tester les fonctions implémentées. Nous avons également un 'makefile' pour compiler et exécuter le programme (vous pouvez simplement taper 'make' puis './my-program' pour tester et voir les résultats).

```
~/LoudOutrageousSoftwareengineer$ tree
.
├── Functions.c
├── Functions.h
├── main.c
├── Makefile
├── my_program
└── replit.nix

0 directories, 6 files
```

Figure 1: les Fichiers utilisée

## 2.1 Définitions des fonctions :

1. `index(int i, int j, int m)` : J'ai choisi de calculer  $i \times (m + 2) + j$  dans une fonction pour une meilleure lisibilité du calcul.
2. `square(double x)` : Pour calculer le carré d'un double.
3. `sol-exact(int i, int j, double hx, double hy)` : Pour calculer la solution exacte du premier problème, qui est :  $u_e(x, y) = xy \sin(2\pi x) \cos(\pi y/2)$
4. `Jacobi(int m, int n)` : Pour implémenter la méthode itérative de Jacobi décrite ci-dessus.
5. `Gauss(int m, int n)` : Pour implémenter la méthode itérative de Gauss-Seidel décrite ci-dessus.
6. `Euler-implicite(int m, int n, int choix)` : Pour implémenter la méthode d'Euler implicite en se basant sur la discrétisation en espace de Jacobi. Il prend en paramètre un entier `choix` qui vaut 1 si l'utilisateur a choisi la méthode de Jacobi ou 2 s'il a choisi la méthode de Gauss-Seidel.
7. `Crank-Nicolson(int m, int n, int choix)` : Pour implémenter la méthode de Crank-Nicolson en se basant également sur la discrétisation en espace de Jacobi. Il prend en paramètre un entier `choix` pour choisir soit la méthode de Jacobi, soit la méthode de Gauss-Seidel.

## 2.2 Modèles testés:

Pour tester nos codes, nous avons testé deux modèles :

1. Celui-ci est écrit et testé dans la fonction du Jacobi et Gauss-Seidel :

$$\begin{cases} -\frac{\partial^2 u(x)}{\partial x^2} - \frac{\partial^2 u(x)}{\partial y^2} + u(x, t) = f(x) & \text{dans } \Omega = (0, 1)^2 \\ u(x) = 0 & \text{sur le bord } \partial\Omega \end{cases}$$

sachant que  $f(x) = \frac{17\pi}{4}2xy \sin(2\pi x) \cos(\frac{\pi y}{2}) + \pi x \sin(2\pi x) \sin(\frac{\pi y}{2}) - 4\pi y \cos(2\pi x) \cos(\frac{\pi y}{2}) + xy \sin(2\pi x) \cos(\frac{\pi y}{2})$

2. Celui-ci est écrit et testé dans la fonction Euler-implicite et Crank-Nicolson :

$$\left\{ \begin{array}{ll} \frac{\partial u(x,t)}{\partial t} - \Delta u(x,t) = (a^2 + b^2)w_0(x,t) & \text{dans } (0,1)^2 \times [0,T] \\ u(x,t) = w_0(x,y) & \text{sur le bord } \partial\Omega \times (0,T) \\ u(x,0) = w_0(x) & \text{sur } (0,1)^2 \end{array} \right.$$

sachant que  $w_0(x,y) = \sin(ax) \cos(by)$  et  $a = b = \pi$ .

### 3 Résultats :

#### 3.1 Jacobi:

On a testé la méthode de Jacobi pour plusieurs valeurs de  $n$  et  $m$  (mais puisque nous sommes dans une grille carrée donc  $n = m$ ). Comme nous avons vu en cours, on prend  $h = \frac{1}{2^n} = \frac{1}{m+1}$  donc nous avons testé :  $m = 3, 7, 15, 31, 63, 127$  et nous avons bien trouvé que le nombre d'itérations se redouble aussi si on double  $h$ , et nous avons aussi trouvé que quand  $h \rightarrow 0$ ,  $\text{err} \rightarrow 0$ , comme on le montre cette exécution.

```
~/LoudOutrageousSoftwareengineer$ ./my_program
*****n=2*****
Time taken for Jacobi(3, 3): 0.000042 seconds
Nombre d'itérations: 40
Erreur: 0.773533
*****n=3*****
Time taken for Jacobi(7, 7): 0.000284 seconds
Nombre d'itérations: 80
Erreur: 0.271931
*****n=4*****
Time taken for Jacobi(15, 15): 0.002315 seconds
Nombre d'itérations: 160
Erreur: 0.071078
*****n=5*****
Time taken for Jacobi(31, 31): 0.019675 seconds
Nombre d'itérations: 320
Erreur: 0.016056
*****n=6*****
Time taken for Jacobi(63, 63): 0.144393 seconds
Nombre d'itérations: 640
Erreur: 0.002902
*****n=7*****
Time taken for Jacobi(127, 127): 1.296068 seconds
Nombre d'itérations: 1280
Erreur: 0.000328
```

Figure 2: Exécution du Jacobi

### 3.2 Gauss-Seidel:

De même, nous avons testé la méthode itérative de Gauss-Seidel pour plusieurs valeurs de  $n$ , et nous avons trouvé aussi que le nombre d'itérations se redouble si  $h$  se redouble. De plus, l'erreur relative entre la solution exacte et la solution provenant de la méthode itérative tend vers zéro quand  $h$  tend vers zéro. De plus, nous avons trouvé que le temps d'exécution augmente si on augmente la taille de la grille. Voici les résultats après l'exécution :

```
~/LoudOutrageousSoftwareengineer$ ./my_program
*****n=3*****
Time taken for Gauss-Seidel(7, 7): 0.000303 seconds
Nombre d'itérations: 80
Erreur: 0.272696
*****n=4*****
Time taken for Gauss-Seidel(15, 15): 0.003011 seconds
Nombre d'itérations: 160
Erreur: 0.072635
*****n=5*****
Time taken for Gauss-Seidel(31, 31): 0.018344 seconds
Nombre d'itérations: 320
Erreur: 0.017356
*****n=6*****
Time taken for Gauss-Seidel(63, 63): 0.172072 seconds
Nombre d'itérations: 640
Erreur: 0.003552
*****n=7*****
Time taken for Gauss-Seidel(127, 127): 1.388708 seconds
Nombre d'itérations: 1280
Erreur: 0.000530
~/LoudOutrageousSoftwareengineer$
```

Figure 3: Exécution du Gauss Seidel

### 3.3 Euler-implicite:

En utilisant le schéma d'Euler implicite décrit ci-dessus, nous avons implémenté cette méthode en donnant le choix à l'utilisateur entre la méthode de discrétisation en espace de Jacobi ou bien de Gauss-Seidel. Pour ce faire, nous avons créé deux conditions sur le choix de l'utilisateur, dont le contenu est la boucle d'itération en espace de chacune des deux méthodes itératives, le tout dans la boucle d'itération sur le temps pour la méthode implicite.

Ensuite, nous avons testé la méthode pour plusieurs valeurs de  $n$  et  $m$  (sachant que  $n = m$ ), et nous avons également calculé le temps d'exécution

pour la méthode.

Voici les résultats de l'exécution:

```
~/LoudOutrageousSoftwareengineer$ ./my_program
*****n=3+Jacobi*****
Time taken for Euler_implicite(3, 3) and used Jacobi 0.00059 seconds
iter=422 err= 9.96043009e-09
*****n=3+Gauss_Seidel*****
Time taken for Euler_implicite(3, 3) and used Gauss-Seidel 0.000116 seconds
iter=410 err= 9.83149708e-09
*****n=15+Jacobi*****
Time taken for Euler_implicite(15, 15) and used Jacobi 0.002195 seconds
iter=539 err= 9.96106138e-09
*****n=15+Gauss_Seidel*****
Time taken for Euler_implicite(15, 15) and used Gauss-Seidel 0.002826 seconds
iter=392 err= 9.94244019e-09
*****n=31+Jacobi*****
Time taken for Euler_implicite(31, 31) and used Jacobi 0.017756 seconds
iter=972 err= 9.99972804e-09
*****n=31+Gauss_Seidel*****
Time taken for Euler_implicite(31, 31) and used Gauss-Seidel 0.027347 seconds
iter=743 err= 9.94176610e-09
*****n=63+Jacobi*****
Time taken for Euler_implicite(63, 63) and used Jacobi 0.209734 seconds
iter=2668 err= 9.97832049e-09
*****n=63+Gauss_Seidel*****
Time taken for Euler_implicite(63, 63) and used Gauss-Seidel 0.354122 seconds
iter=1811 err= 9.99968134e-09
*****n=127+Jacobi*****
Time taken for Euler_implicite(127, 127) and used Jacobi 3.399822 seconds
iter=9514 err= 9.99409090e-09
*****n=127+Gauss_Seidel*****
Time taken for Euler_implicite(127, 127) and used Gauss-Seidel 3.951675 seconds
iter=5310 err= 9.99365142e-09
~/LoudOutrageousSoftwareengineer$
```

Figure 4: Exécution du Euler implicite

### 3.4 Crank-Nicolson:

Par le même principe, nous avons implémenté la méthode de Crank-Nicolson en donnant le choix aux utilisateurs entre la méthode de Jacobi et Gauss-Seidel, en se basant sur le principe de la méthode de Crank-Nicolson qui combine les deux méthodes d'Euler implicite et explicite (en divisant les deux termes par 2). C'est pourquoi j'ai multiplié le deuxième terme par 0.5.

Ensuite, nous avons testé la méthode pour plusieurs valeurs de  $n$  et  $m$  (sachant que  $n = m$ ), et nous avons également calculé le temps d'exécution pour la méthode.

```

~/LoudOutrageousSoftwareengineer$ ./my_program
*****n=3+Jacobi*****
Time taken for Crank-Nickolson(3, 3) and used Jacobi 0.000248 seconds
iter=1261 err= 9.98255203e-09
*****n=3+Gauss_Seidel*****
Time taken for Crank-Nickolson(3, 3) and used Gauss-Seidel 0.000404 seconds
iter=1254 err= 9.94796357e-09
*****n=15+Jacobi*****
Time taken for Crank-Nickolson(15, 15) and used Jacobi 0.003726 seconds
iter=1042 err= 9.91457724e-09
*****n=15+Gauss_Seidel*****
Time taken for Crank-Nickolson(15, 15) and used Gauss-Seidel 0.005596 seconds
iter=870 err= 9.94829500e-09
*****n=31+Jacobi*****
Time taken for Crank-Nickolson(31, 31) and used Jacobi 0.028101 seconds
iter=1985 err= 9.96731434e-09
*****n=31+Gauss_Seidel*****
Time taken for Crank-Nickolson(31, 31) and used Gauss-Seidel 0.041044 seconds
iter=1308 err= 9.94419132e-09
*****n=63+Jacobi*****
Time taken for Crank-Nickolson(63, 63) and used Jacobi 0.365111 seconds
iter=5178 err= 9.99161384e-09
*****n=63+Gauss_Seidel*****
Time taken for Crank-Nickolson(63, 63) and used Gauss-Seidel 0.446739 seconds
iter=3041 err= 9.99165457e-09
*****n=127+Jacobi*****
Time taken for Crank-Nickolson(127, 127) and used Jacobi 4.403099 seconds
iter=14192 err= 9.99947189e-09
*****n=127+Gauss_Seidel*****
Time taken for Crank-Nickolson(127, 127) and used Gauss-Seidel 4.954277 seconds
iter=8359 err= 9.99491660e-09
~/LoudOutrageousSoftwareengineer$

```

Figure 5: Exécution du Crack Nicolson



## 4 Conclusion:

En définitive, les 4 fonctions de discrétisation en espace-temps fonctionnent bien. Toutefois, un problème de durée d'exécution est survenu lorsque j'ai augmenté la taille de la grille, en raison de l'utilisation de la plateforme en ligne Replit. Replit offre un environnement de développement intégré (IDE) pour programmer dans différents langages de programmation, mais sa RAM et son CPU sont limités. C'est pourquoi j'ai testé jusqu'à 127 (en doublant la valeur de "iter" lorsque "h" est doublé). De plus, j'ai vidé la mémoire à la fin de chaque fonction pour éviter les problèmes de fuite de mémoire. Comme nous n'utilisons pas les tableaux de solution pour tracer "u" au cours du temps et en espace, cela n'a pas affecté le résultat final.