Solveur Elliptique Multi-Grille - Implémentation Python/PyTorch et Applications à la Mécanique des Fluides

Rapport de stage de L3 sous la direction de Louis Thiry dans l'équipe ODYSSEY à l'INRIA Rennes.

GAUVAIN THOMAS, ENS Rennes, France and INRIA - Équipe ODYSSEY, France

Nous présentons dans ce papier une implémentation Python / PyTorch d'un solveur multi-grille pour les équations différentielles elliptiques, essentielles en mécanique des fluides. Ce solveur est ensuite comparé à une implémentation en Fortran, ainsi qu'à un solveur spectral. Nous montrons que grâce à la bibliothèque PyTorch et aux architectures modernes, il est possible de réunir lisibilité et efficacité dans une telle implémentation. Montrant que le bon fonctionnement d'une méthode multi-grille repose sur des détails essentiels, nous menons une étude de ces paramètres. Enfin, nous fournissons un exemple d'application du solveur.

Additional Key Words and Phrases: Multigrid, Python, PyTorch

Contents

1 INTRODUCTION

En mécanique des fluides, et donc en océanographie, les équations de Laplace et Helmholtz jouent des rôles fondamentaux. Par exemple dans le cas de fluides incompressibles, la pression est directement reliée au champ de vitesse par une équation de Poisson.

Ainsi, dans une simulation de fluide, une équation elliptique de ce type sans composante temporelle se doit d'être résolue efficacement à chaque pas de temps. À ce jour, les méthodes les plus efficaces sont des solveurs spectraux utilisant des transformées de Fourier (ou transformées en sinus ou cosinus) discrètes rapides, très optimisées et parallélisables, et possédant déjà des implémentations sur GPU. Cependant, ces méthodes sont principalement limitées à des géométries rectangulaires, et difficilement adaptables à des géométries plus complexes, pourtant indispensables pour réaliser des simulations plus réalistes de l'océan par exemple. De plus, ces méthodes et les

Author's address: Gauvain Thomas, gauvain.thomas@ens-rennes.fr, ENS Rennes, Rennes, France and INRIA - Équipe ODYSSEY, Rennes, France.

interfaces les utilisant sont historiquement souvent écrites en Fortran, et de fait produisent des codes longs et difficilement lisibles ou modulables.

Aussi, ces vingt dernières années ont vu se développer des solveurs itératifs de meilleures complexités, basés sur des méthodes multi-grilles. Ces méthodes se prêtent particulièrement à des architectures CPU/GPU récentes.

L'objectif de ce papier est ainsi de présenter une implémentation d'un solveur Poisson 2D multi-grille en Python efficace grâce la bibliothèque PyTorch permettant du calcul tensoriel optimisé avec accélaration GPU et compilation à la volée (jit), et d'en comparer les performances avec celles de solveurs déjà existant. Un accent particulier sera porté sur la lisibilité et la portabilité du programme.

2 PUBLICATIONS

Les deux références principales utilisées pour effectuer ce travail sont [7] et [3], deux livres relativement récents et très complets présentant en détail les méthodes multi-grilles, à la fois en théorie et en pratique. Ces ouvrages récapitulent l'ensemble des publications sur le sujet de manière organisée, on pourra donc s'y référer pour une bibliographie exhaustive sur le sujet.

La mention de l'utilisation de paramètres de relaxations différents est introduite et étudiée dans [4].

La création de cette implémentation poursuit le travail et la volonté de [5] et [2].

D'autres solveurs existent déjà, tels que PyAMG [1], un solveur multi-grille algébrique en Python, bien plus général qu'un solveur multi-grille géométrique, mais au prix d'une efficacité moindre et sans support GPU.

Oceananigans [6] est à l'inverse un simulateur de fluides ne faisant pas usage d'algorithme multi-grille pour ses solveurs Poisson itératifs, mais utilise plutôt des méthodes de gradient conjugués avec pré-conditionneurs, en général plus lentes.

Ainsi, on remarque que malgré ce que les avantages que les méthodes multi-grilles ont a offrir, elles ne sont pas encore répandues au sein de la communauté.

3 BACKGROUND

3.1 Modélisation du problème (version continue)

3.1.1 Équations.

Soit $\Omega \in \mathbb{R}^d$ un ouvert, $\Gamma = \partial\Omega$ sa frontière. On considère le problème suivant :

$$\begin{cases} L^\Omega u = f^\Omega & (\Omega) \\ L^\Gamma u = f^\Gamma & (\Gamma) \end{cases} \tag{1}$$

avec L^Ω un opérateur linéaire elliptique sur Ω et L^Γ un opérateur représentant les conditions aux limites.

On s'intéresse en particulier aux cas suivants :

- Équation de Poisson ($L^\Omega = \Delta$) :
  $\begin{cases} \Delta u = f^\Omega & (\Omega) \\ L^\Gamma u = f^\Gamma & (\Gamma) \end{cases}$
- Équation de Helmholtz ($L^\Omega = \Delta - \lambda I$):
  $\begin{cases} \Delta u - \lambda u = f^\Omega & (\Omega) \\ L^\Gamma u = f^\Gamma & (\Gamma) \end{cases}$

Avec $\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$ l'opérateur Laplacien.

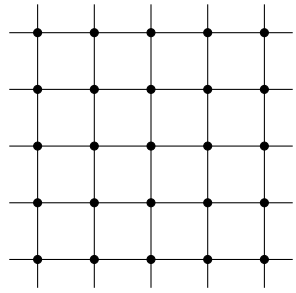


Fig. 1. Exemple d'une grille G_h

3.1.2 *Conditions limites.*

On se limite ici a deux types de conditions aux limites:

- Conditions de Dirichlet homogènes :
  $u(\boldsymbol{x}) = 0, \quad \boldsymbol{x} \in \Gamma$
- Conditions de Neumann homogènes :
  $\frac{\partial u}{\partial n}(\boldsymbol{x}) = 0, \quad \boldsymbol{x} \in \Gamma$
  Avec cette condition uniquement, il n'y a pas unicité de la solution. Pour y remédier, on impose en général une moyenne nulle à la solution.

3.2 Discrétisation du problème

On se place dans un cas en deux dimensions ($d = 2$).

Étant donné un paramètre de grille $\boldsymbol{h} = (h_x, h_y)$ représentant les pas de la grille uniforme, on note la grille Ω_h.

Formellement, $\Omega_h = \Omega \cap G_h$ avec $G_h = h_x\mathbb{Z} + h_y\mathbb{Z}$, avec $h_x = h_y$ dans le cas d'une grille carrée.

De même, on considère les opérateurs discrétisés L_h^Ω et L_h^Γ, définis respectivement sur Ω_h et Γ_h.

3.2.1 *Équation de Poisson discrétisée.*

Il nous faut discrétiser les équations continues sur notre grille.

On discrétise l'opérateur Laplacien ainsi (Laplacien 2D 5 points) :

Pour $(x, y) \in \Omega_h \setminus \Gamma_h$ (grille intérieure)

$$\begin{aligned}
\Delta_h u_h(x,y) &= \frac{u_h(x+h_x,y) + u_h(x-h_x,y) - 2u_h(x,y)}{h_x^2} + \frac{u_h(x,y+h_y) + u_h(x,y-h_y) - 2u_h(x,y)}{h_y^2} \\
&= \frac{1}{h^2}\left[u_h(x+h,y) + u_h(x-h,y) + u_h(x,y+h) + u_h(x,y-h) - 4u_h(x,y)\right] \quad \text{pour une grille carrée} \\
&\coloneqq \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}_h * u_h
\end{aligned}$$

Les conditions aux limites sont souvent difficiles à traiter, et seront étudiées plus en détail dans la suite. Dans les cas présentés ici, il est assez aisé d'éliminer ces conditions. Par exemple, pour des conditions de Dirichlet homogènes, il suffirait d'imposer les valeurs aux bords (ou de manières équivalente modifier de manière appropriée le système d'équations aux bords).

3.2.2 *Équation de Helmholtz discrétisée.*

On a de manière analogue :
Pour $(x, y) \in \Omega_h \setminus \Gamma_h$ (grille intérieure)

$$\begin{aligned} L_h u_h(x,y) &= \frac{u_h(x+h_x,y) + u_h(x-h_x,y) - 2u_h(x,y)}{h_x^2} \\ &+ \frac{u_h(x,y+h_y) + u_h(x,y-h_y) - 2u_h(x,y)}{h_y^2} - \lambda u_h(x,y) \\ &= \frac{1}{h^2}[u_h(x+h,y) + u_h(x-h,y) + u_h(x,y+h) + u_h(x,y-h) - 4u_h(x,y)] \\ &- \lambda u_h(x,y) \end{aligned}$$

pour une grille carrée

$$\coloneqq \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -(4+h^2\lambda) & 1 \\ 0 & 1 & 0 \end{bmatrix}_h * u_h$$

3.3 Méthodes de relaxation

3.3.1 *Méthode de Jacobi.*

Présentons ici la méthode de Jacobi pour une équation de Poisson en 1D avec une subdivision uniforme h. Le cas général s'en déduit sans difficulté.
Pour un point $x \in \Omega_h \setminus \Gamma_h$ à l'intérieur, on a :

$$\Delta_h u = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} = f(x)$$

En résolvant indépendamment sur chaque point en $u(x)$, on obtient :

$$u(x) = \frac{u(x+h) + u(x-h) - h^2 f(x)}{2} \coloneqq z(x)$$

Une itération de Jacobi consiste alors à calculer la nouvelle approximation ainsi :

$$u^{m+1} = z^{m+1}$$

L'itération de cette méthode converge bien vers la solution du système, mais avec une complexité quadratique en le nombre d'inconnues.
On introduit le paramètre de relaxation ω, et la méthode de relaxation de Jacobi consiste à effectuer une combinaison linéaire entre z^m et l'approximation précédente, permettant ainsi, si le paramètre est bien choisi, d'accélerer le lissage des erreurs de plus hautes fréquences.

$$u^{m+1} = u^m + \omega(z^{m+1} - u^m) = (1-\omega)u^m + \omega z^{m+1}$$

La convergence de cette méthode est optimale lorsque $\omega = 1$, dans ce cas toujours insatisfaisante (quadratique en le nombre d'inconnues). Toutefois, ce n'est pas la convergence asymptotique qui sera essentielle par la suite, mais plutôt ses propriétés de lissage de l'erreur, c'est à dire diminuer la capacité à diminuer erreurs de hautes fréquences. Cette méthode, malgré sa convergence lente est en revanche entièrement parallélisable.

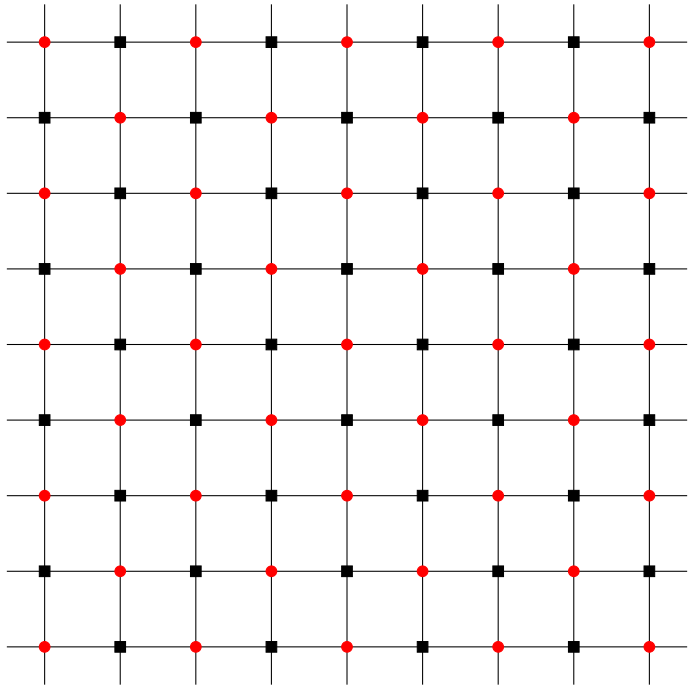


Fig. 2. Découpage rouge/noir pour la méthode de Gauss-Seidel

3.3.2 Méthode de Gauss-Seidel rouge-noir (GS-RB).

La méthode de Gauss-Seidel consiste à effectuer successivement une relaxation sur un sous-ensemble de la grille, puis sur l'autre sous-ensemble en se servant de la nouvelle approximation calculée pour le premier ensemble.

Pour la méthode de Gauss-Seidel présentée ici, un découpage rouge-noir est utilisé ici (voir 2).

En pratique, cette méthode est nettement plus efficace que la précédente : la convergence est plus rapide, tout en étant parallélisable au niveau de chaque moitié de grille. C'est cette méthode que nous utiliserons dans la suite.

3.4 Multi-Grille

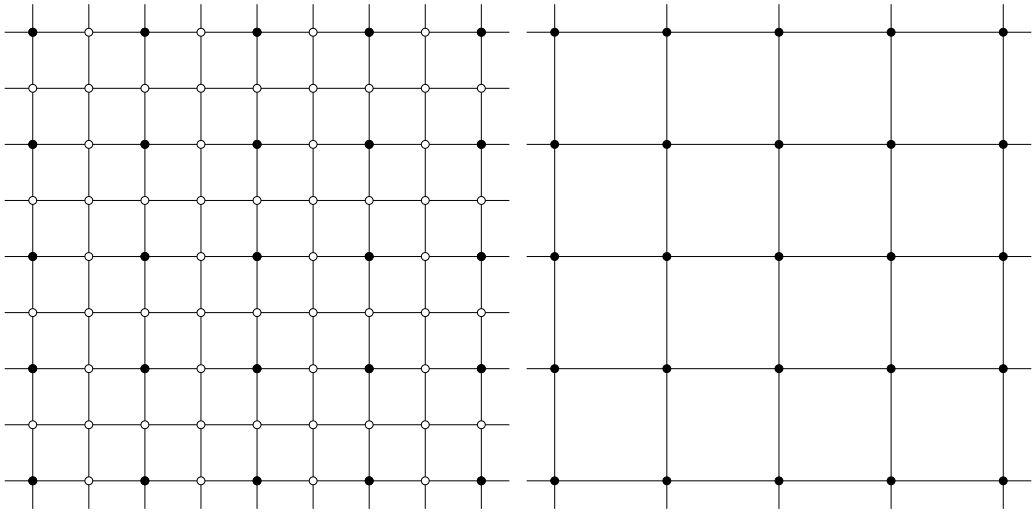
3.4.1 Méthodes itératives.

On souhaite résoudre le système linéaire $Lu = f$

On pose les définitions suivantes, pour une itération m :

- Solution approchée : u^m
- Erreur : $e^m = u - u^m$
  Comme l'erreur est en pratique inaccessible, on calcule le résiduel qui représente à quel point l'approximation n'est pas solution du système. Il n'est cependant pas nécessairement vrai que e^m est faible en norme si r^m l'est.
- Résidu : $r^m = f - Lu^m$

On peut alors vérifier que l'erreur et le résidu vérifient la même équation reliant u et f. On obtient ainsi l'équation du résidu :



(a) Ω_h (b) Ω_{2h}

Fig. 3. Une grille fine et une grille grossière

$$Le^m = r^m \tag{2}$$

Ainsi, si l'on résout l'équation en e^m (en pratique de manière approchée), on peut calculer une nouvelle approximation u^{m+1} :

$$u^{m+1} = u^m + e^m \tag{3}$$

C'est sur ce principe que reposent les méthodes itératives que nous étudierons ici.

3.4.2 Principe du multi-grille.

Revenons à l'équation du résidu 2. On cherche à calculer de manière efficace une approximation de l'erreur e^m. Pour cela, une idée est de calculer une approximation sur une grille plus grossière à l'aide d'un approximation L_H de l'opérateur L_h, avec par exemple $H = 2h$.

Or ce processus n'est pas convergent en soi, mais il suffit de relaxer la solution avant et/ou après le calcul de l'erreur pour qu'il le devienne.

On obtient alors un algorithme multi-grille à deux niveaux :

Two-Grid Cycle :

- Presmoothing
- Calcul du résiduel $r_h^m = f_h - L_h u_h^m$
- Restriction du résiduel r_H sur la nouvelle grille Ω_H
- Résolution sur Ω_H : $L_H e_H^m = r_H^m$
- Interpolation de la correction (approximée) sur Ω_h : e_h
- Calcul de la nouvelle approximation : $u_h^{m+1} = u_h^m + e_h$
- Postsmoothing

Naturellement, on peut récursivement employer la même technique pour résoudre sur Ω_H.

V-cycle

- Presmoothing

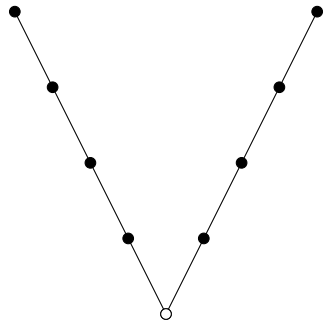


Fig. 4. Structure d'un V-cycle | • : relaxation, ○ : solution exacte, \: restriction, / : interpolation

- Calcul du résiduel $r_h^m = f_h - L_h u_h^m$
- Restriction du résiduel r_H sur la nouvelle grille Ω_H
- Résolution sur Ω_H : $L_H e_H^m = r_H^m$ récursivement (avec 0 en approximation initiale)
- Interpolation de la correction (approximée) sur Ω_h : e_h
- Calcul de la nouvelle approximation : $u_h^{m+1} = u_h^m + e_h$
- Postsmoothing

Une fois sur la résolution la plus grossière, on peut résoudre le système linéaire par relaxation, ou par une méthode exacte.

Full Multigrid (FMG)

L'idée du *Full Multigrid* est de trouver une meilleure approximation initiale en sur lesquelles se basent les V-cycles.

L'algorithme commence par le calcul d'une approximation initiale sur la grille la plus grossière, et à chaque niveau un V-cycle. Pour cela, l'idée du multi-grille reste la même : en partant de la grille la plus grossière, on calcule une solution approximative avec un V-cycle, puis on l'interpole au niveau du dessus sur lequel on résout récursivement de la même façon. L'interpolation du FMG ne joue pas le même rôle que celle des V-cycles : dans le premier cas, c'est une interpolation de la solution approchée, tandis que dans le second cas c'est une interpolation de la correction. Souvent, il est commun d'utiliser une interpolation d'ordre plus élevé pour le FMG que pour les V-cycles.

Le choix des opérateurs de relaxation, d'interpolation et de restriction constituent en pratique des paramètres essentiels dans l'efficacité de l'algorithme.

3.4.3 Grilles.

Le choix d'un type de grille peut être assez subtil et influe sur les opérateurs ainsi que le traitement des conditions aux limites.

On étudie ici des grilles dont les valeurs sont placées aux sommets de la grille, dites *vertex-centered* (voir figure 6a, et les grilles dont les valeurs sont situées au centre des cellules, dites *cell-centered* (voir figure 6b).

Selon le type de grille choisi, les conditions aux limites peuvent porter ou non sur des points appartenant à la grille, ce qui en change le traitement. De même, les opérateurs d'interpolation et de restriction se doivent d'être modifiés en conséquence.

3.4.4 Opérateur de restriction.

Les opérateurs de restriction et d'interpolation principalement utilisés sont les suivants :

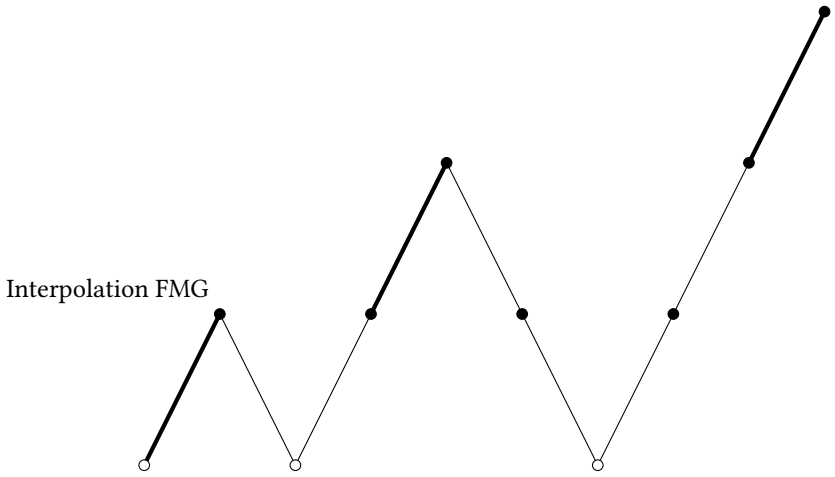
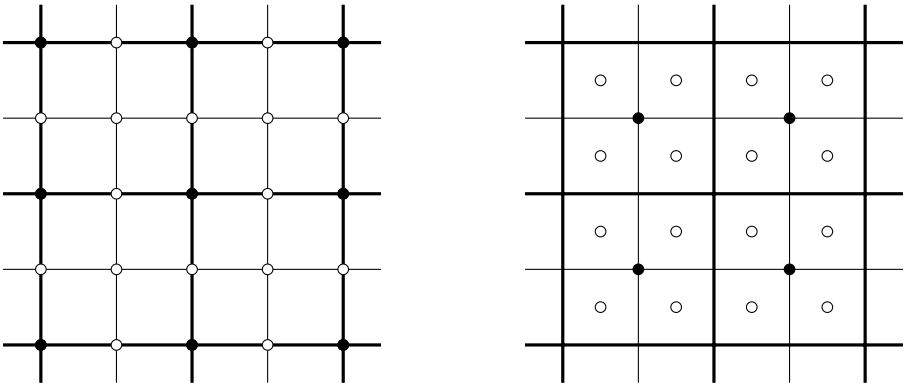


Fig. 5. Structure d'un FMG



(a) Grille vertex-centered (b) Grille cell-centered

Fig. 6. Exemple d'une grille vertex-centered et d'une grille cell-centered et leurs restrictions

Grilles vertex-centered
Full Weighting (FW)

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}_h^{2h}$$

Half Weighting (HW)

$$\frac{1}{8}\begin{bmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{bmatrix}_h^{2h}$$

Grilles cell-centered

Table 1. Performances d'un V-cycle pour une grille $N \times N$ (en temps moyen par point de grille)

Solveur	N=256	N=512	N=1024
Fortran	3.25e-8 s	3.52e-8 s	4.85e-8 s
Python PyTorch non compilé	7.54e-7 s	3.47e-7 s	2.95e-7 s
Python PyTorch (CPU)	3.83e-7 s	1.69e-7 s	1.50e-7 s
Python PyTorch (GPU)	3.87e-7 s	1.04e-7 s	3.83e-8 s

Calculé sur l'exemple d'un bruit gaussien

Moyenne 4 points

$$\frac{1}{4}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}_h^{2h}$$

Attention : $\Omega_H \not\subset \Omega_h$ pour les grilles centrées

3.4.5 Opérateur d'interpolation.

Dans les deux cas, une interpolation bilinéaire est utilisée :

$$\frac{1}{4}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}_h^{2h}$$

4 TRAVAIL

4.1 Implémentation

Le solveur est structuré sous forme d'une classe, à l'initialisation, les opérateurs coûteux en calcul sont pré-compilés avec la méthode `torch.jit.trace`[1] et stockés en tant qu'attributs de la classe. Ainsi, la structure-même du solveur multi-grille reposant uniquement sur ces opérateurs en est indépendante, et donc très facilement modulable.

4.1.1 Performances.

On compare ici les performances de ces différents solveurs (voir 1, 2) dans les mêmes conditions et avec les mêmes paramètres (GS-RB, deux pré/post-relaxations et même résolution basse échelle).

- Solveur MG Fortran
- Solveur spectral Python
- Solveur MG Python (sans compilation)
- Solveur MG Python (avec compilation)

Le gain de performance après la compilation PyTorch est notable, environ deux fois plus rapide. Cette performance reste tout même plus lente que celle du solveur Fortran, mais parvient à l'égaler avec GPU en haute résolutions. Toutefois, les méthodes multi-grilles restent bien moins performantes que les méthodes spectrales, mais offrent bien plus de possibilités et sont générales à toute une variété de problèmes.

[1] https://pytorch.org/docs/stable/generated/torch.jit.trace.html

Table 2. Performances des solveurs pour une grille $N \times N$ (en temps moyen par point de grille)

Solveur	N=256	N=512	N=1024
Fortran (V-cycles)	5.18e-7 s	x	x
Solveur spectral Python (CPU)	2.93e-7 s	3.119e-7 s	3.73e-7 s
Python PyTorch (V-cycles) (CPU)	1.12e-6 s	3.79e-7 s	3.72e-7 s
Python PyTorch (V-cycles) (GPU)	1.02e-6 s	1.58e-7 s	9.30e-8 s

Calculé dans la cas d'une équation d'Euler (voir 4)

4.1.2 Élimination des conditions aux limites.

Conditions de Dirichlet homogènes : Dans ce cas, comme les points de grilles extérieurs sont ceux sur lesquels la condition porte, il suffit simplement d'imposer une valeur nulle à ces points de grille.

Conditions de Neumann homogènes : Les conditions portent ici sur les dérivées normales au bord. La difficulté est que dans le cas d'une grille décalée comme ici, cette dérivée est définie non pas sur les points de grilles, mais entre ceux-ci. Pour y remédier, on peut au choix éliminer la condition en modifiant directement l'opérateur discrétisé aux bords, ou bien introduire de manière équivalente des *ghost points*, c'est à dire introduire artificiellement des nouveaux points en dehors de la grille, et agir sur l'intérieur de cette nouvelle grille qui se trouve être Ω_h. Ici, on impose la valeur de ces nouveaux points à celle de la couche extérieure, pour satisfaire la condition de dérivée nulle. Bien qu'équivalentes, cette seconde méthode est en pratique plus rapide (en vectoriel), car ne rajoutant pas de nombreux cas à traiter.

4.1.3 Résolution basse échelle. On remarque en pratique (d'autant plus en vectoriel) que la résolution à basse échelle est plus rapide en utilisant des méthodes déjà existantes de résolution de système linéaires. Notamment, l'opérateur L_H étant linéaire, on peut simplement calculer la matrice de son inverse L_H^{-1} pour une résolution grossière (en pratique de l'ordre de 32×32 environ) et ainsi résoudre exactement le système à la plus basse échelle, au lieu d'un certain nombre de relaxations. Dans le cas de simulations où une résolution est effectuée à chaque pas de temps (ce avec le même opérateur), le coût du calcul de cette inverse est compensé par le gain de temps qu'il apporte.

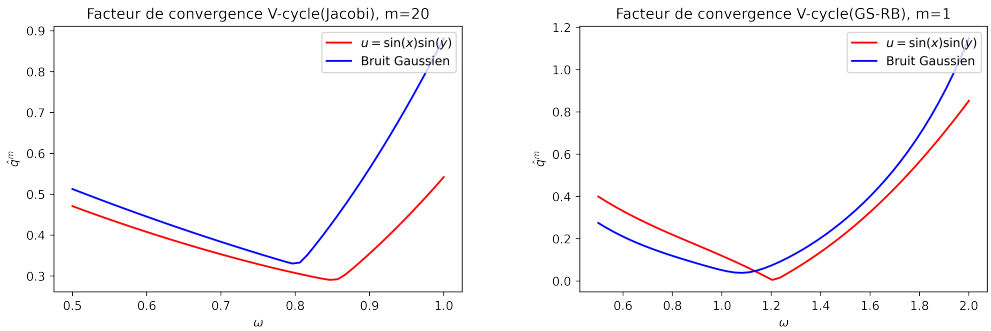
4.2 Optimisation des paramètres

On remarque empiriquement assez vite que la façon dont sont choisis certains paramètres est cruciale quant au bon fonctionnement de l'algorithme.

On s'appuiera sur ces exemples particuliers, la résolution d'une équation de Poisson $\Delta u = f$ sur $\Omega = [0, 2\pi]$, pour différentes tailles de grille, avec :

- $u(x, y) = \sin(x)\sin(y) \quad (\Omega)$
  Avec conditions de Dirichlet homogènes, un cas assez spécifique qui permet d'étudier le comportement sur les basses fréquences.
- $u(x, y) = \cos(x)\cos(y) \quad (\Omega)$
  L'équivalent du premier exemple avec conditions de Neumann homogènes.
- Un bruit Gaussien sur Ω permettant d'étudier le comportement sur l'ensemble du spectre des fréquences.

4.2.1 Paramètre de relaxation ω.



(a) Jacobi (b) Gauss-Seidel rouge-noir

Fig. 7. Facteurs de convergence en fonction du paramètre ω

On présente ici un choix du paramètre ω optimisant la facteur de convergence moyen $\hat{q}^{(m)}$ sur m itération d'un V-cycle. Où :

$$\hat{q}^{(m)} = \sqrt[m]{\frac{\|r_h^m\|}{\|r_0^m\|}}$$

Jacobi

On trouve un facteur de convergence minimal lorsque le facteur de relaxation vaut $\omega^* \approx 0.78$, très proche de celui optimal d'un point de vue thérorique $\omega_{th}^* = 0.8$

Gaus-Seidel Red-Black

De même, on a $\omega^* \approx 1.14$ et $\omega_{th}^* = 1.2$

De manière générale, ces paramètres sont fortement dépendants des densités spectrales des solutions, il ne semble pas exister un choix optimal d'un point de vue général. Toutefois, il pourrait être intéressant de chercher à optimiser ces paramètres pour un problème particulier, sur un ensemble de solutions types. Sans que cela ne fasse beaucoup gagner en rapidité, cela peut avoir une influence sur la précision de l'algorithme.

De la même manière qu'est introduit un paramètre de relaxation pour les V-cycles, il peut être intéressant de relaxer différemment lors d'une descente et d'une remontée. On introduit donc ces facteurs de relaxation différents ω_1 et ω_2 lors des étapes de descente et de remontée. Avec une étude similaire sur ces paramètres, on trouve :

Jacobi : $\omega_{pre}^* = 1.17$ et $\omega_{post}^* = 0.59$

Gaus-Seidel Red-Black : $\omega_{pre}^* = 1.02$ et $\omega_{post}^* = 1.14$

4.2.2 Nombres de relaxations.

Le nombre de relaxations effectuées, à la fois lors d'une descente ou d'une montée d'un V-cycle est un autre paramètre qu'il peut être intéressant d'étudier, dans le but de trouver un compromis entre précision et rapidité. On présente dans la table 3 le temps de calcul, ainsi que le nombre de V-cycles effectués pour parvenir en dessous d'une certaine tolérance. On observe que le choix d'un cycle V(2,2) est judicieux.

Table 3. Efficacité d'un V-cycle en fonction des nombres de relaxations

Cycle	Itérations	Temps (ms)
V(0,1)	17	249
V(1,0)	16	212
V(1,1)	9	127
V(1,2)	8	130
V(2,1)	8	124
V(2,2)	7	123

Où $V(n_{pre},n_{post})$ représente un V-cycle avec n_{pre} relaxations à la descente et n_{post} relaxations à la montée (tolérance relative de 1e-11).

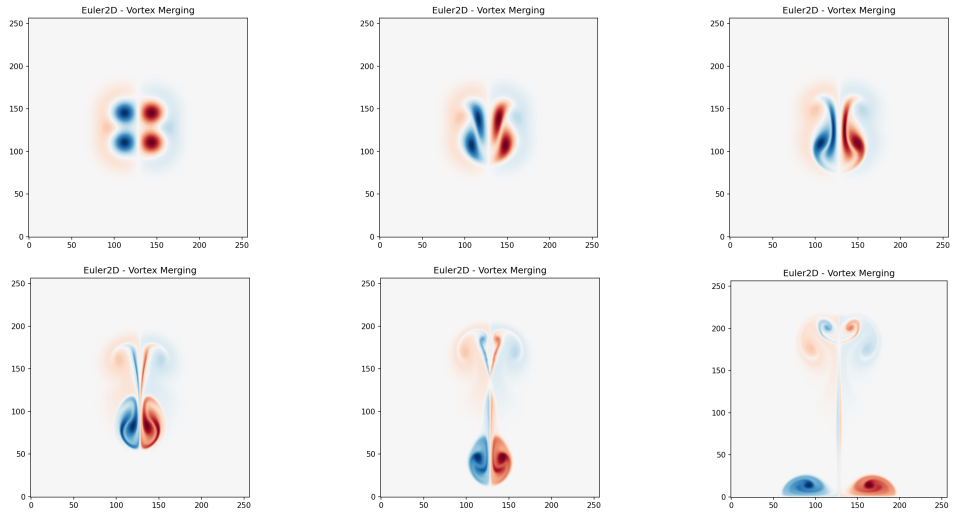


Fig. 8. Vortex merging : snapshots de la vorticité pour $t \in \{1, 20, 40, 90, 150, 240\}$

4.3 Utilisation

On présente dans cette section un modèle sur lequel le solveur multi-grille a été employé et comparé.

4.3.1 Euler2D.

$$\frac{D\omega}{Dt} = \frac{\partial\omega}{\partial t} + (u \cdot \nabla)\omega = 0 \tag{4}$$

avec $\omega = \nabla \times u = \Delta\Psi$, où ω est la vorticité et Ψ la fonction courant.
On présente un exemple d'animation d'une simulation de ces équation figure 8.

5 CONCLUSION

Nous avons présenté dans ce papier les motivations et le principe d'une méthode multi-grille, et le rôle de chacun des opérateurs intervenant dans sa réalisation. Celle-ci a été implémentée en Python, en tirant profit des nouvelles architectures et des bibliothèques de calculs vectoriels optimisées et est

disponible à cette adresse : https://github.com/gauvain-thomas/Geometric-Multigrid. La technique la plus importante utilisée est celle de la compilation (jit), et permet d'obtenir des performances du même ordre de grandeur qu'un solveur Fortran par exemple. L'avantage de cette implémentation relève donc de sa facilité de compréhension et d'adaptabilité. Cette accessibilité et facilité d'emploi permet en toute simplicité d'adapter et d'améliorer certaines fonctionnalités. On peut toutefois concevoir une implémentation mêlant Python pour l'interface et Fortran pour le calcul numérique intensif, ce qui permet ainsi de réunir le meilleur de chacun de ces deux langages dans la même implémentation.

Les performances de ce solveur ont ensuite été étudiées plus en détail, ainsi que l'optimisation de certains paramètres caractérisant le solveur. Enfin, une application se servant de ce solveur a ensuite été présentée.

Différentes extensions s'offrent alors naturellement : une adaptation des opérateurs pour gérer un cas 3D ainsi qu'a des géométries plus complexes, de même que la possibilité de traiter plus de sortes de grilles et de conditions aux limites.

Un autre apport pourrait être de permettre de traiter directement un cas où la solution et les termes de forçage ne sont pas sur la même grille, directement en modifiant la discrétisation. Cela permettrait notamment d'éviter tout problème au niveau des interpolations d'une grille à l'autre, souvent coûteuses, ou rajoutant une perte d'information non nécessaire.

REFERENCES

[1] Nathan Bell, Luke N. Olson, and Jacob Schroder. 2022. PyAMG: Algebraic Multigrid Solvers in Python. *Journal of Open Source Software* 7, 72 (2022), 4142. https://doi.org/10.21105/joss.04142

[2] Tom Bertalan, Akand Islam, Roger Sidje, and Eric Carlson. 2012. OpenMG: A New Multigrid Implementation in Python. 69–75. https://doi.org/10.25080/Majora-54c7f2c8-00c

[3] W.L. Briggs, V.E. Henson, and S.F. McCormick. 2000. *A Multigrid Tutorial: Second Edition.* Society for Industrial and Applied Mathematics. https://books.google.fr/books?id=oSTGBm64o1AC

[4] Jun Zhang. 1996. Acceleration of five-point red-black Gauss-Seidel in multigrid for Poisson equation. *Appl. Math. Comput.* 80, 1 (1996), 73–93. https://doi.org/10.1016/0096-3003(95)00276-6

[5] Abhilash Reddy Malipeddi. 2017. GeometricMultigrid. https://github.com/AbhilashReddyM/GeometricMultigrid

[6] Ali Ramadhan, Gregory LeClaire Wagner, Chris Hill, Jean-Michel Campin, Valentin Churavy, Tim Besard, Andre Souza, Alan Edelman, Raffaele Ferrari, and John Marshall. 2020. Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *Journal of Open Source Software* 5, 53 (2020), 2018. https://doi.org/10.21105/joss.02018

[7] U. Trottenberg, C.W.O.A.S. Ulrich Trottenberg, C.W. Oosterlee, A. Schuller, A. Brandt, P. Oswald, and K. Stüben. 2001. *Multigrid.* Elsevier Science. https://books.google.fr/books?id=-og1wD-Nx_wC