Institut National Supérieur
d'Informatique

Mini-projet 6 : Méthode du Gradient Conjugué pour Matrices Creuses

Spécialité : I2AD
Niveau : Master 1
Semestre : 7
UE : Algèbre linéaire
Enseignant : Dr. Dimby
Étudiante : ANDRIATSIFERANA No Kanto Lorida
Étudiant : ZO Manampisoa Hermann
Année académique : 2025 - 2026

17 juin 2025

Résumé

Ce mini-projet porte sur l'implémentation et l'étude de la méthode du gradient conjugué pour la résolution de systèmes linéaires dont la matrice est creuse, symétrique et définie positive. Le projet s'appuie sur des fondements mathématiques solides et se décline en plusieurs volets : génération de matrices creuses types (notamment la matrice de Poisson 2D), résolution par gradient conjugué avec et sans préconditionneur, analyse de la convergence, et comparaison entre les bibliothèques `NumPy` et `SciPy`.

1 Concepts mathématiques

1.1 Définition du problème

Nous cherchons à résoudre un système linéaire de la forme :

$$Ax = b \tag{1}$$

où $A \in \mathbb{R}^{n \times n}$ est une matrice symétrique définie positive et creuse, x le vecteur inconnu, et b un vecteur donné.

1.2 Génération de la matrice creuse (Poisson 2D)

Contexte : Le Laplacien 2D est un opérateur différentiel discret utilisé pour modéliser des phénomènes physiques comme la diffusion ou la chaleur.

Discrétisation : On discrétise le domaine 2D (par exemple, un carré unité) avec une grille régulière. Cela transforme l'équation :

$$-\Delta u(x, y) = f(x, y)$$

en un système linéaire :

$$Au = f$$

où A est une matrice creuse symétrique définie positive de taille $n^2 \times n^2$.

1.3 Méthode du Gradient Conjugué (CG)

Problème : Résoudre $Ax = b$ quand A est symétrique définie positive.
Principe : C'est une méthode itérative qui construit une suite de vecteurs $x_0, x_1, x_2, \ldots$ convergeant vers la solution.

Algorithme du Gradient Conjugué

1. Initialisation :
$$x_0 = 0, \quad r_0 = b - Ax_0, \quad p_0 = r_0$$
2. Pour $k = 0, 1, 2, \ldots$:
$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$$
$$x_{k+1} = x_k + \alpha_k p_k$$
$$r_{k+1} = r_k - \alpha_k A p_k$$
Si $\|r_{k+1}\|$ est petit, on arrête.
$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$
$$p_{k+1} = r_{k+1} + \beta_k p_k$$

Remarque : À chaque itération, la nouvelle direction p_k est choisie de manière à être conjuguée aux précédentes. Cela permet d'accélérer la convergence.

1.4 Visualisation de la convergence

Erreur : On peut visualiser :
— L'erreur absolue $\|x_k - x^*\|$ si la solution exacte x^* est connue.
— Le résidu $\|r_k\|$ si x^* est inconnu.
Graphe : On peut tracer la courbe $\log(\|r_k\|)$ en fonction du nombre d'itérations pour observer la vitesse de convergence.

1.5 Préconditionneurs

Objectif : Accélérer la convergence du gradient conjugué en transformant le système. On résout :
$$M^{-1}Ax = M^{-1}b$$
au lieu de $Ax = b$, où M est un approximant de A plus simple à inverser.

Exemples classiques

— **Préconditionneur de Jacobi :** $M = diag(A)$ On divise chaque composante du résidu par la diagonale de A.
— **Préconditionneur SSOR (Symmetric Successive Over-Relaxation) :** Ce préconditionneur repose sur la décomposition $A = D+L+L^T$, avec D la diagonale, L la partie triangulaire inférieure. Il est plus efficace que Jacobi mais plus coûteux.

2 Méthodologie

2.1 Génération de la matrice de Poisson 2D

Le problème considéré est la résolution de l'équation de Poisson :

$$-\Delta u(x, y) = f(x, y), \quad (x, y) \in \Omega = [0, 1]^2$$

avec conditions de Dirichlet homogènes. La discrétisation par différences finies sur une grille régulière de taille $n \times n$ donne un système linéaire $Ax = b$, où A est une matrice creuse de taille $n^2 \times n^2$, symétrique définie positive.

2.2 Méthode du Gradient Conjugué

Le gradient conjugué (CG) est utilisé pour résoudre $Ax = b$, avec les variantes suivantes :

— **Sans préconditionneur** : CG classique.
— **Avec préconditionneur Jacobi** : $M = \text{diag}(A)$.
— **Avec préconditionneur SSOR** : basé sur la décomposition $A = D + L + L^T$.

Le critère d'arrêt est basé sur la norme du résidu $\|r_k\| < 10^{-8}$. Les performances sont mesurées en nombre d'itérations et temps d'exécution.

2.3 Implémentations NumPy et SciPy

Deux implémentations ont été comparées :

— **Implémentation NumPy** : code manuel basé sur l'algorithme CG.
— **Fonction SciPy** : `scipy.sparse.linalg.cg` avec support natif pour les préconditionneurs via l'option `M=`.

3 Résultats et Analyse

3.1 Convergence : Implémentation NumPy

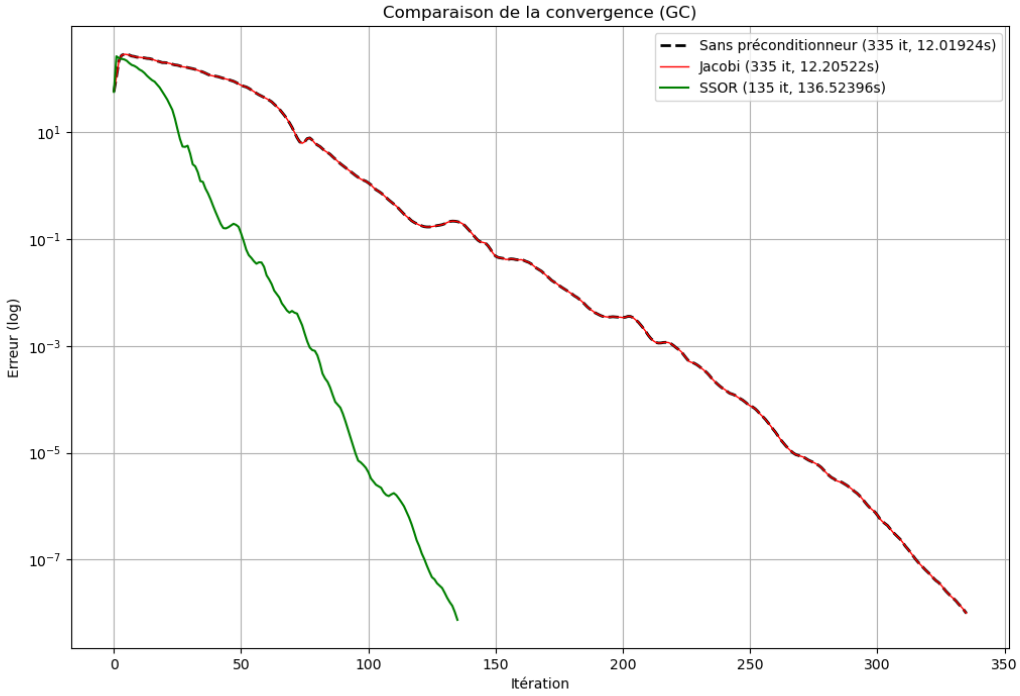

FIGURE 1 – Convergence (implémentation NumPy) : CG sans préconditionneur, Jacobi, et SSOR

Les résultats montrent que :

— Sans préconditionneur et avec Jacobi, le nombre d'itérations est identique (335), avec un léger avantage en temps pour la version sans préconditionneur.

— Le préconditionneur SSOR réduit drastiquement les itérations (135) mais son coût de calcul élevé (`136 s`) le rend peu optimal dans ce contexte.

3.2 Convergence : Fonction SciPy

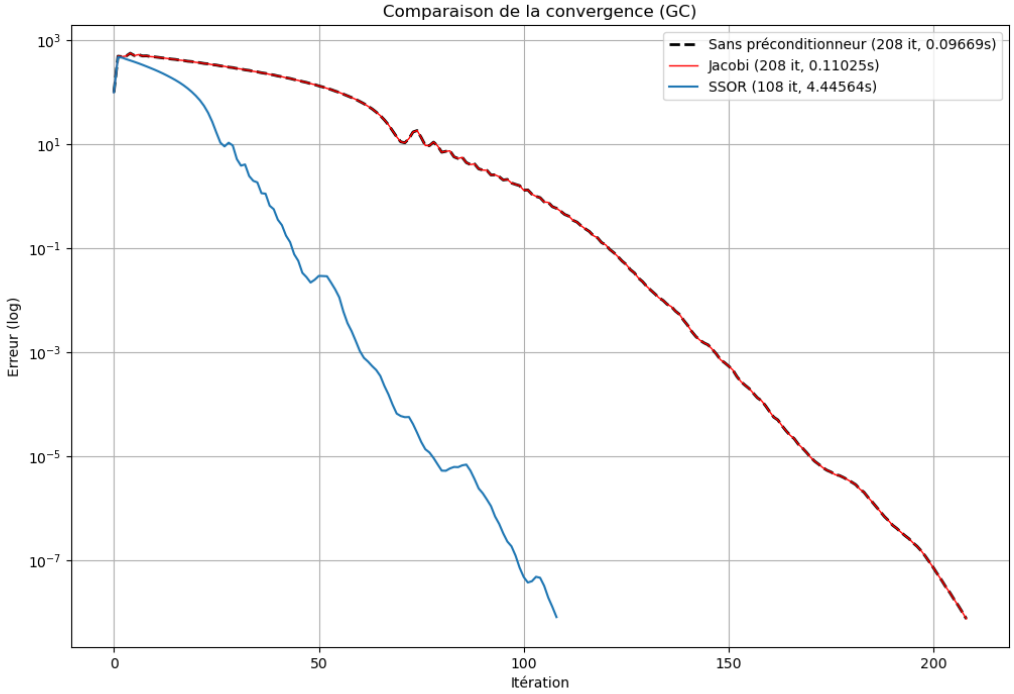

FIGURE 2 – Convergence (fonction SciPy) : CG sans préconditionneur, Jacobi, et SSOR

Les observations sont similaires à celles obtenues avec NumPy. SciPy donne :

— Même nombre d'itérations pour Jacobi et sans préconditionneur (208), avec un léger gain de temps pour la version sans préconditionneur.

— Le préconditionneur SSOR divise presque par deux le nombre d'itérations (108) mais reste plus coûteux en temps d'exécution.

4 Comparaison des approches NumPy et SciPy

La table suivante résume les différences essentielles entre les deux approches du projet :

Critère	**NumPy (manuel)**	**SciPy (optimisé)**
Langage utilisé	Python + NumPy	Python + SciPy
Fonction CG	Codée à la main	`scipy.sparse.linalg.cg`
Préconditionneur Jacobi	Implémenté manuellement	Utilisé via `LinearOperator`
Préconditionneur SSOR	Implémenté manuellement	Utilisé via `LinearOperator`
Nombre d'itérations (Jacobi)	335	208
Nombre d'itérations (SSOR)	135	108
Facilité d'utilisation	Moins pratique	Très pratique
Contrôle de l'algorithme	Total	Limité
Performance	Bonne mais lente (SSOR)	Optimisée (multithreadée)

Analyse : SciPy offre un environnement plus rapide et pratique pour les grands systèmes linéaires creux. Cependant, l'implémentation manuelle en NumPy permet de comprendre en profondeur le fonctionnement de l'algorithme, ce qui est essentiel en contexte pédagogique.

5 Conclusion

Les tests effectués montrent que le choix du préconditionneur a un impact majeur sur la convergence du gradient conjugué :

— Le préconditionneur Jacobi est simple à implémenter mais n'améliore pas le nombre d'itérations.
— SSOR accélère la convergence en termes d'itérations, mais au prix d'un temps de calcul très élevé.
— SciPy permet une implémentation rapide et fiable, mais reste sensible au même compromis entre précision et performance.

L'algorithme du gradient conjugué reste donc très dépendant du préconditionneur et du contexte d'utilisation (temps réel ou précision extrême).