# Optimisation de l'algorithme Particle Swarm Optimization (PSO) et application à la fonction Rastrigin

Lorthios Théo

## 1 Introduction

Particle Swarm Optimization (PSO) est un algorithme d'optimisation heuristique inspiré du comportement social des oiseaux et des poissons. Il a été développé par James Kennedy et Russel Eberhart en 1995. Cet algorithme est couramment utilisé pour résoudre de nombreux problèmes d'optimisation combinatoire et continue, et présente plusieurs avantages, notamment sa simplicité, sa rapidité de convergence et sa capacité à échapper aux minima locaux dans de nombreux cas.

Dans ce rapport, nous nous concentrons sur l'optimisation de l'algorithme PSO en termes de temps d'exécution, en explorant différentes techniques de multi-threading et de multi-processing. Ensuite, nous appliquons l'algorithme PSO optimisé pour résoudre un problème d'optimisation complexe, à savoir la fonction Rastrigin. La fonction Rastrigin est une fonction non linéaire, multimodale et continue, qui a été introduite par Léonid Rastrigin en 1974 et est considérée comme une fonction de test difficile en raison de ses nombreux minimums locaux.

Pour illustrer la progression en temps réel des particules sur la fonction Rastrigin et permettre la modification des hyperparamètres de l'algorithme, un tableau de bord interactif a été développé à l'aide de la bibliothèque Python Dash. Le tableau de bord est disponible dans le répertoire GitHub du projet sous le dossier "dash_board", avec un fichier "app.py" qui contient le code source du tableau de bord (Figure 1).

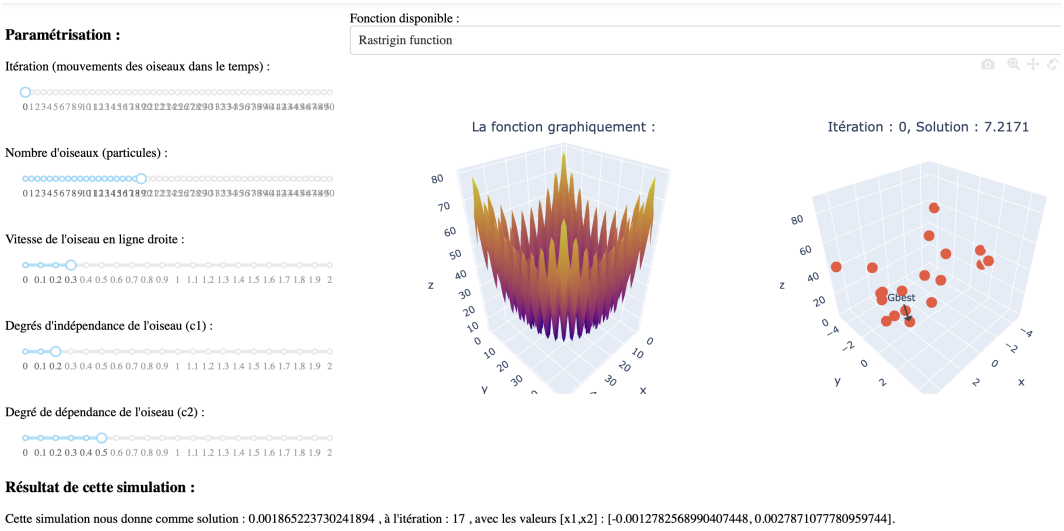


Figure 1: Visualisation du PSO sur la fonction Rastrigin avec un Dash. On remarque la bonne convergence des particules à l'itération 17.

# 2 Particle Swarm Optimization et la fonction Rastrigin

## 2.1 Particle Swarm Optimization

L'algorithme Particle Swarm Optimization (PSO) est basé sur la recherche collaborative d'une solution optimale par un groupe de particules qui se déplacent dans l'espace de recherche. Chaque particule représente une solution potentielle au problème d'optimisation.

Les particules sont mises à jour en suivant deux composantes principales :

1. **Composante cognitive** : La particule est attirée vers sa meilleure position personnelle trouvée (pBest) : Exploration.
2. **Composante sociale** : La particule est attirée vers la meilleure position globale trouvée par l'ensemble des particules (gBest) : Exploitation.

La mise à jour de la position et de la vitesse de chaque particule est effectuée en utilisant les formules suivantes :

$$v_{i,t+1} = w * v_{i,t} + c_1 * r_1 * (pBest_i - x_{i,t}) + c_2 * r_2 * (gBest - x_{i,t}) \quad (1)$$

$$x_{i,t+1} = x_{i,t} + v_{i,t+1} \quad (2)$$

où $v_{i,t}$ est la vitesse de la particule $i$ à l'instant $t$, $x_{i,t}$ est sa position, $w$ est le

facteur d'inertie, $c_1$ et $c_2$ sont les coefficients d'accélération cognitive et sociale, et $r_1$ et $r_2$ sont des nombres aléatoires uniformes.

### 2.1.1 Avantages et limites du PSO par rapport à la descente de gradient

**Avantages:**

- *Pas de gradient nécessaire:* Contrairement à la descente de gradient, le PSO ne nécessite pas de gradient analytique ni d'approximation numérique du gradient. Cela le rend approprié pour les problèmes où le calcul du gradient est difficile ou coûteux.

- *Robustesse face aux minimums locaux:* Le PSO est moins susceptible de se coincer dans des minimums locaux que la descente de gradient. L'exploration et l'exploitation dans l'espace de recherche permettent au PSO d'échapper aux pièges des minimums locaux et d'explorer de manière plus exhaustive l'espace des solutions.

- *Facilité de mise en œuvre:* Le PSO est relativement simple à comprendre et à mettre en œuvre, ce qui peut être un avantage en termes de temps de développement et de maintenance du code.

- *Parallélisation:* Le PSO est intrinsèquement parallèle, car chaque particule évolue indépendamment des autres. Cela permet d'exploiter facilement les architectures parallèles ou distribuées pour accélérer les calculs.

**Limites:**

- *Convergence lente:* Dans certains cas, le PSO peut converger plus lentement que la descente de gradient, surtout si les paramètres de l'algorithme ne sont pas bien adaptés au problème. Il peut être nécessaire d'effectuer un réglage des hyperparamètres pour améliorer la vitesse de convergence.

- *Problèmes de stagnation:* Le PSO peut parfois stagner si les particules se regroupent autour d'une solution sous-optimale. Cela peut être résolu en

ajustant les paramètres de l'algorithme ou en introduisant des mécanismes supplémentaires pour encourager la diversité des solutions.

- *Sensibilité aux hyperparamètres:* Bien que le PSO soit moins sensible aux hyperparamètres que la descente de gradient, le choix des paramètres, tels que le facteur d'inertie et les coefficients d'accélération cognitive et sociale, peut avoir un impact significatif sur les performances de l'algorithme.

### 2.2 Fonction Rastrigin

La fonction Rastrigin est une fonction mathématique couramment utilisée pour évaluer les performances des algorithmes d'optimisation. Elle a été introduite par Léonid Rastrigin en 1974 et est considérée comme une fonction de test difficile en raison de ses nombreux minimums locaux.

La fonction Rastrigin est une fonction non linéaire, multimodale et continue, définie comme suit pour un espace de recherche à $n$ dimensions :

$$f(\mathbf{x}) = A \times n + \sum_{i=1}^{n} \left[ x_i^2 - A \cos(2\pi x_i) \right] \tag{3}$$

où $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ est un vecteur de dimension $n$ représentant une solution candidate, et $A$ est une constante positive, généralement égale à 10.

La fonction Rastrigin est généralement évaluée sur une plage de valeurs pour chaque dimension, par exemple $x_i \in [-5.12, 5.12]$.

L'optimum global de la fonction Rastrigin est atteint lorsque toutes les composantes du vecteur $x$ sont égales à zéro, c'est-à-dire $x_i = 0$ pour $i = 1, \ldots, n$. La valeur minimale de la fonction Rastrigin est alors égale à 0 :

$$f(x^*) = f(0, 0, \ldots, 0) = 0 \tag{4}$$

## 3 Expérimentation pour l'optimisation computationnelle du PSO

### 3.1 Description de l'expérimentation

L'objectif de cette expérimentation est d'optimiser la performance computationnelle de l'algorithme PSO en utilisant différentes techniques d'optimisation, notamment le multi-threading et le multi-processing. L'expérimentation consiste à tester trois structures de code différentes pour l'algorithme PSO, chacune intégrant ces techniques d'optimisation, et à comparer leurs performances.

## 3.2 Méthodologie

La méthodologie de l'expérimentation consiste à évaluer les performances des différentes structures de code en faisant varier trois paramètres clés (les autres paramètres restant fixes) :

1. Nombre d'itérations (mémoire vive)

2. Nombre de simulations de Monte-Carlo (répétitions boucle)

3. Nombre de particules (dimension des vecteurs)

Les performances des différentes structures de code sont évaluées en mesurant le **temps d'exécution en secondes** pour des paramètres fixes et **la moyenne des moyennes des positions des particules à chaque simulation de Monte-Carlo** à la dernière itération. La seconde métrique indique la convergence de l'algorithme ; l'objectif est d'obtenir un algorithme à la fois rapide et convergent.

### 3.2.1 Structures de code testées

Les trois structures de code testées pour l'algorithme PSO sont les suivantes :

1. **PSO_JSON :** Toutes les informations inter-simulation-itération sont stockées dans un fichier JSON massif qui se remplit au fur et à mesure de l'exécution de l'algorithme. Cette version permet de conserver l'historique des particules pour chaque simulation.

2. **PSO Low Memory :** Seules les informations essentielles pour l'algorithme sont conservées en mettant à jour un fichier JSON fixe pour chaque simulation de Monte-Carlo.

3. **PSO Low Memory avec variables locales :** Les informations essentielles pour l'algorithme sont conservées en mettant à jour des variables locales pour chaque simulation de Monte-Carlo.

### 3.2.2 Versions disponibles et testées

Les versions suivantes de l'algorithme PSO ont été testées :

- V1 : version naïve avec des boucles for simples
- V2 : version avec Multi-Threading (non disponible pour le Code 2)
- V3 : version avec Multi-Processing

Les résultats de l'expérimentation ont été obtenus en exécutant le script *run_comparaison.py* et sont stockés dans des dataframes (data).

# 4 Résultats et remarques

## 4.1 Impact du nombre d'itérations

Dans cette section, nous examinons l'impact de la variation du nombre d'itérations de l'algorithme PSO sur les performances. Nous constatons que les performances des versions naïves et multi-threadées sont proportionnelles au nombre d'itérations. L'utilisation du multi-threading n'améliore pas le temps d'exécution, car le stockage n'est pas impacté. En revanche, il est intéressant de noter que le multi-processing devient avantageux à partir de 4500 itérations ou plus, mais est moins performant pour un faible nombre d'itérations comparé aux versions naïves. (Figure 2)
L'intégration de l'optimisation NUMBA pourrait améliorer l'efficacité du paramètre

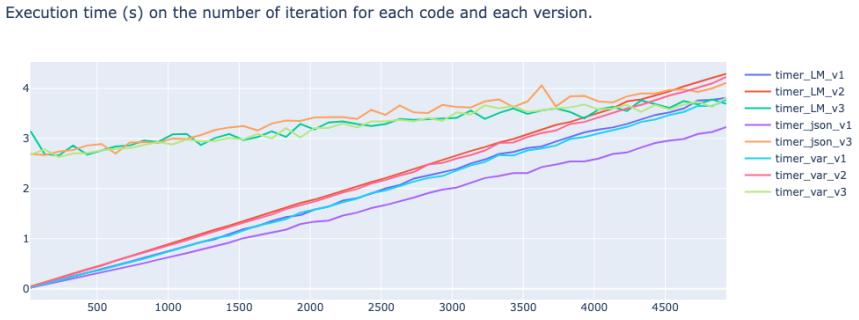


Figure 2: Nombre de seconde par nombre d'itération à configuration fixe pour l'ensemble des codes et des versions

en question, mais cela nécessiterait de modifier le code pour remplacer les fichiers JSON par des DataFrames pandas.

D'autre part, nous constatons que l'algorithme converge assez rapidement vers l'optimum, rendant ainsi les améliorations apportées par le multi-processing ou le multi-threading moins bénéfiques dans cette configuration et pour cette fonction spécifique.

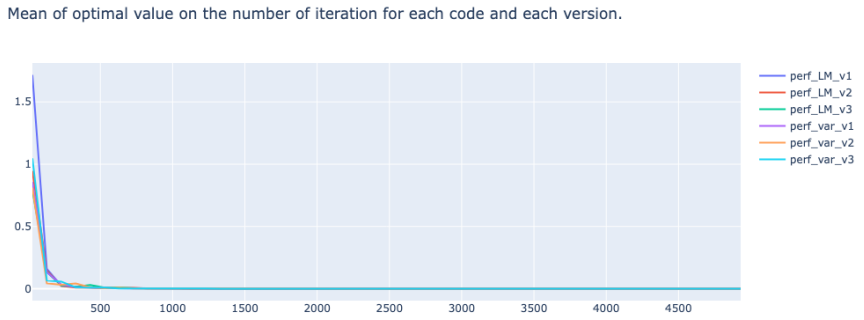


Figure 3: Moyenne des meilleures performances par nombre d'itération à configuration fixe pour l'ensemble des codes et des versions

## 4.2 Ajustement du nombre de simulations de Monte-Carlo

Il est connu que l'algorithme PSO a plus de chances d'atteindre l'extrema global (minimum ou maximum) de la fonction si le nombre de simulations de Monte-Carlo est augmenté. En effet, chaque simulation commence avec un tirage aléatoire des positions initiales. En augmentant le nombre de tirages aléatoires, nous explorerons de manière plus approfondie la fonction et son domaine.

Dans ce cas, nous observons une tendance similaire à celle du nombre d'itérations : le multi-processing améliore les performances d'exécution. Cependant, pour le nombre de simulations de Monte-Carlo, le point de basculement qui favorise le multi-processing se situe à partir de 150 simulations.

Nous constatons également que lorsque le nombre de simulations est important, tous les cœurs sont occupés et surchargés, ce qui entraîne une augmentation linéaire du temps d'exécution par la suite. Néanmoins, cette augmentation reste moins rapide que celle des processus naïfs.

Recommandation : Utilisez le multi-processing pour un grand nombre de simulations de Monte-Carlo (plus de 150) si la fonction à optimiser présente des contraintes importantes sur ses entrées ou si la fonction a un large domaine d'image. L'objectif est de couvrir le plus d'espace possible dans l'exploration. Figure 4

En ce qui concerne les performances, nous constatons que la moyenne des

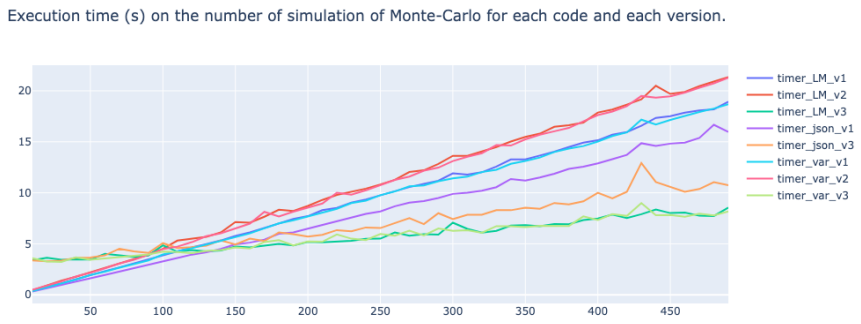


Figure 4: Nombre de seconde par le nombre de simulation de monte-carlo à configuration fixe pour l'ensemble des codes et des versions

résultats à la dernière étape reste globalement la même, tandis que la variance diminue mécaniquement avec l'augmentation du nombre de simulations.

Comme mentionné précédemment, les fonctions ayant un large domaine peuvent poser problème au PSO si les particules ne sont pas bien réparties dès le départ (en raison de la randomisation ou d'une vitesse trop faible combinée à un trop petit nombre d'itérations). Chaque simulation apporte de l'information, réduisant ainsi mécaniquement la variance des résultats finaux.

Il est donc important de combiner la métrique de moyenne classique à celle de la variance pour s'assurer que le PSO a bien convergé vers la même valeur à travers un grand nombre de simulations. Cette approche permet de confirmer que l'extrema trouvé est bien global, compte tenu des contraintes imposées. Figure 5

## 4.3 Le nombre de particules

Dans cette section, nous allons faire varier le nombre de particules utilisées dans la fonction. Intuitivement, plus le nombre de particules augmente, plus les calculs vectoriels pour mettre à jour les positions et les vitesses seront complexes et lourds. Sachant que nous utilisons le multi-processing et le multi-threading pour les simulations de Monte-Carlo, nous ne devrions pas observer d'amélioration significative des performances.

À notre avis, l'optimisation computationnelle pourrait être réalisée en décomposant les calculs vectoriels sur les cœurs de processeur en utilisant des tenseurs ou

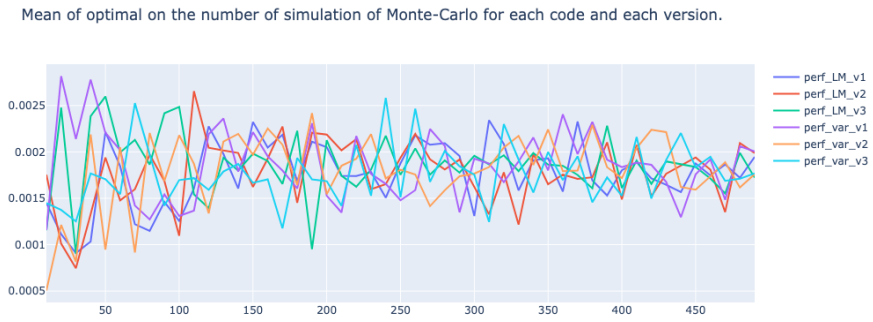


Figure 5: Moyenne des meilleures performances par nombre de simulation de Monte-Carlo à configuration fixe pour l'ensemble des codes et des versions

d'autres techniques de calcul vectoriel. Toutefois, cette approche n'a pas été explorée en raison de contraintes de temps.

Conformément à notre intuition, les deux techniques d'optimisation ne se révèlent pas efficaces pour ce paramètre. De plus, nous observons que l'augmentation du nombre de particules n'alourdit pas significativement le temps de calcul.

Dans une configuration fixe, le multi-processing rallonge le temps d'exécution, tandis que le multi-threading demeure à un niveau similaire à la méthode naïve.

Parmi les trois codes possibles, nous constatons que le stockage dans un fichier JSON contenant toutes les informations est le plus rapide. Cela peut être dû au fait que le nombre de simulations (20) et d'itérations (200) est relativement faible. La mémoire occupée par le fichier JSON n'interfère pas avec les performances de calcul. Figure 6

En ce qui concerne les performances, nous observons que l'augmentation du nombre de particules est cruciale pour la convergence vers l'optimum. L'ensemble des processus converge vers la solution avec un peu plus de 150 particules. Figure 7

Étant donné que nous ne sommes pas confrontés à des contraintes computationnelles importantes, il est possible de fixer ce paramètre à une valeur relativement élevée afin de garantir la convergence de l'algorithme.

## 4.4 Synthèse des résultats

Au cours de cette étude, nous avons analysé l'impact de différents paramètres sur les performances de l'algorithme PSO pour optimiser la fonction de Rast-

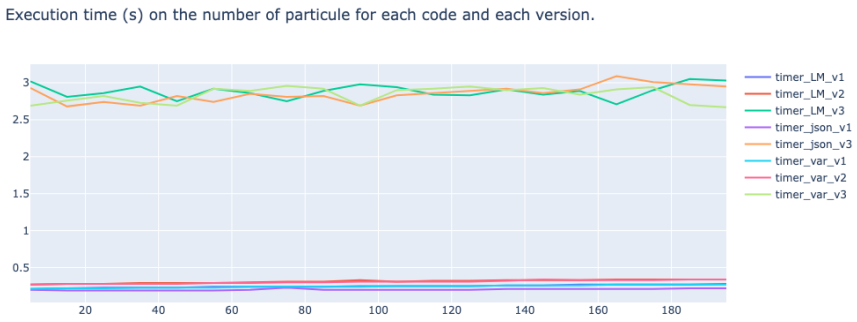


Figure 6: Nombre de seconde par le nombre de particule à configuration fixe pour l'ensemble des codes et des versions

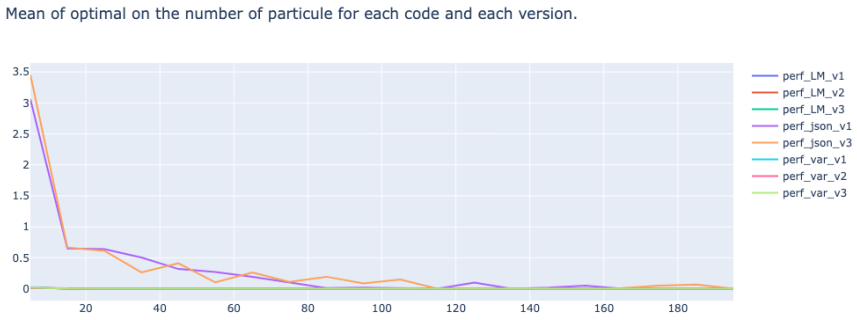


Figure 7: Moyenne des meilleures performances par nombre de particule à configuration fixe pour l'ensemble des codes et des versions

rigin. Nous avons constaté que le multi-processing améliore les performances d'exécution lorsqu'il y a un grand nombre d'itérations (plus de 4500) ou de simulations de Monte-Carlo (plus de 150). Cependant, pour le nombre de particules, les techniques d'optimisation n'ont pas montré d'amélioration significative des performances. Malgré cela, un nombre de particules relativement élevé (plus de 150) garantit une meilleure convergence vers l'optimum.

# 5 Conclusion

Au cours de cette étude, nous avons exploré l'algorithme d'optimisation par essaim de particules (PSO) et ses avantages, tels que sa capacité à gérer des

problèmes non linéaires et non convexes, ainsi que sa simplicité et sa rapidité de convergence. Nous avons également étudié l'application de l'algorithme PSO pour optimiser la fonction de Rastrigin, une fonction complexe et multimodale.

Afin d'améliorer les performances computationnelles de l'algorithme, nous avons appliqué différentes méthodologies d'optimisation, notamment le multi-threading et le multi-processing. Nous avons également examiné l'impact de divers paramètres, tels que le nombre d'itérations, le nombre de simulations de Monte-Carlo et le nombre de particules, sur les performances de l'algorithme.

En somme, cette étude met en évidence l'efficacité et la flexibilité de l'algorithme PSO pour résoudre des problèmes d'optimisation complexes. Les améliorations apportées aux performances computationnelles permettent de tirer pleinement parti de l'algorithme PSO, en l'adaptant aux besoins spécifiques de chaque problème d'optimisation.

# References


[1] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43, 1995.

[2] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," pp. 69–73, 1998.

[3] S. S. Rao, *Engineering Optimization: Theory and Practice*. John Wiley & Sons, 2009.

[4] L. Rastrigin, "Systems of extremal control," *Mir*, vol. 3, pp. 49–61, 1963.