Rapport de projet :
Algorithme Artificial Bee Colony (ABC)

Groupe 6:

Professeur referent: M. IDOUMGHAR

POMMIER	*ZEROUAL*	*FANGER*	*SCHELLENBAUM*
Logan	*Samy*	*Cantin*	*Mathieu*
L3 Informatique	*L3 Informatique*	*L3 Informatique*	*L3 Informatique*
logan.pommier@uha.fr	*samy;68100@gmail.com*	*cantinfanger@gmail.com*	*mathieu.schellenbaum@gmail.com*

I. Contexte d'étude

Dans le cadre de notre cours d'Intelligence Artificielle (IA), nous avons eu l'opportunité d'étudier en profondeur les méthodes d'optimisation, en particulier les algorithmes évolutionnaires et génétiques. Ces méthodes, classées sous le terme de métaheuristiques, sont des techniques d'optimisation inspirées par des comportements naturels et utilisées pour résoudre des problèmes complexes.

A. *Classificartion des méthodes d'optimisation*

Dans la figure ci-dessous, on retrouve une palette de choix à la disposition des chercheurs pour résoudre des problèmes de manière efficace.

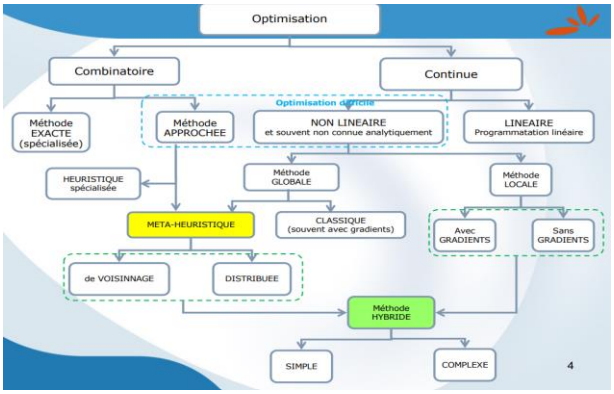


Figure 1: Méthodes d'optimisation

B. *Introduction aux méthodes Méthaheuristiques*

Les méthodes métaheuristiques sont des algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficiles pour lesquels on ne connaît pas de méthode classique plus efficace. Elles sont souvent inspirées par des systèmes naturels, tels que le recuit simulé, les algorithmes génétiques, et l'optimisation par essaims particulaires. Ces techniques permettent de trouver des solutions approchées de bonne qualité en un temps raisonnable, sans garantir l'optimalité de la solution obtenue.

C. *Algorithmes évolutaires et génétiques*

Parmi les méthodes métaheuristiques, les algorithmes évolutionnaires et génétiques occupent une place de choix. Ces algorithmes s'inspirent des processus biologiques de l'évolution et de la sélection naturelle pour générer et améliorer des solutions candidates. Les algorithmes génétiques, par exemple, utilisent des opérateurs tels que la sélection, le croisement et la mutation pour explorer l'espace de recherche et trouver des solutions optimales.

D. *Optimisation par Colonies d'abeilles*

Parmi les algorithmes évolutionnaires, on trouve une famille de méthodes métaheuristiques basées sur le comportement des abeilles dans la vie.

II. Introduction

A. Historique de l'algorithme ABC

L'algorithme Artifical Bee Colony (ABC) a été développé par B.BASTURK et D.JARABOGO en Turquie, en 2005, inspiré par le comportement des abeilles réelles dans leur recherche de nourriture. Il a été conçu pour surmonter les limitations des méthodes d'optimisation traditionnelles, offrant une approche robuste et efficace pour trouver des solutions optimales dans divers domaines d'application tels que l'optimisation de fonction numérique ou l'apprentissage des réseaux de neurones.

B. Comportement des abeilles et inspiration pour l'algortihme ABC

Comme les fourmis, les abeilles sont des créatures hautement évoluées capables de réaliser des tâches complexes en groupe. Parmi ces tâches, on trouve la communication, la construction de nids, le contrôle de l'environnement, la protection et la division du travail. Les abeilles, en particulier, démontrent des capacités extraordinaires pour trouver des solutions optimales dans la recherche de nourriture. Ces dernières possèdent des comportements spécifiques pour la recherche de nourriture : les abeilles éclaireuses partent à la recherche de nouvelles sources de nourriture et communiquent leurs découvertes aux autres abeilles de la colonie par des danses spécifiques. Les abeilles observatrices observent ces danses et se dirigent vers les sources de nourriture les plus prometteuses. Les abeilles employeuses, quant à elles, exploitent les sources de nourriture connues.

L'algorithme ABC s'inspire de ce comportement pour structurer son processus d'optimisation. Il utilise trois types d'abeilles artificielles: les abeilles employées, qui explorent l'espace de recherche pour trouver de nouvelles solutions ; les abeilles spectatrices, qui évaluent et sélectionnent les meilleures solutions basées sur les informations fournies par les abeilles employées ; et les abeilles scoutes, qui recherchent de nouvelles solutions aléatoirement lorsque les solutions actuelles ne peuvent plus être améliorées.

En s'inspirant du comportement naturel des abeilles, l'algorithme ABC parvient à explorer efficacement l'espace de recherche et à converger vers des solutions optimales, démontrant ainsi son efficacité et sa robustesse dans divers contextes d'optimisation.

III. Méthodologie

Nous avons structuré notre projet en plusieurs phases distinctes pour garantir une approche méthodique et rigoureuse. La première phase, d'analyse et d'enquête, a été divisée en deux groupes de deux personnes chacun. Le premier groupe s'est chargé de rassembler des articles scientifiques expliquant le principe de l'algorithme des Colonies d'Abeilles Artificielles (ABC) et des algorithmes génétiques de manière générale. Leur tâche consistait à rechercher et lire des articles académiques, synthétiser les informations pertinentes et préparer un rapport détaillé sur les principes et les applications de ces algorithmes. Le second groupe, quant à lui, a analysé et compris le code Python de l'algorithme ABC. Ils ont étudié le code source, identifié les différentes phases et fonctions du code, et documenté leurs observations et les points clés du code.

Après cette phase d'analyse et d'enquête, nous avons procédé à une phase de rassemblement où nous avons consolidé nos découvertes. Cette étape visait à comparer les résultats théoriques avec l'implémentation pratique en Python, discuter et ajuster les incohérences éventuelles, et préparer un rapport consolidé intégrant les découvertes des deux groupes. Cette phase était cruciale pour s'assurer que nos observations théoriques correspondaient bien à l'implémentation pratique.

La phase de développement a suivi, où nous avons conservé les mêmes groupes. Le premier groupe s'est chargé d'implémenter et de tester l'algorithme ABC en Python sur un PC. Leurs tâches incluaient l'implémentation de l'algorithme, les tests unitaires et de performance, ainsi que la documentation des résultats et des observations. Le second groupe, de son côté, a implémenté un algorithme C++ de l'algorithme ABC et l'a testé. Leurs tâches comprenaient l'implémentation de l'algorithme en C++, les tests unitaires et de performance, et la comparaison des résultats avec l'implémentation Python.

Nous avons ensuite eu un premier entretien avec notre professeur référent, qui nous a apporté des corrections et des conseils sur notre manière de présenter et sur notre analyse du code. Ses suggestions incluaient l'amélioration de la clarté et de la structure des rapports, des recommandations pour optimiser le code et les tests, ainsi que des conseils sur la présentation des résultats et des analyses.

La phase de tests a été consacrée à l'implémentation des fonctions objectifs Rastrigin, Ackley, Schwefel et Rosenbrock. Nous avons réalisé des tests pour différentes valeurs de n_limits (10, 50, 100) et calculé les moyennes et les écarts types des résultats. Ces résultats ont été conservés dans un fichier Excel pour analyse et comparaison entre les deux algorithmes.

Ensuite, nous avons entamé la phase de rédaction, où nous avons consolidé toutes les informations et les résultats obtenus pour rédiger cet article. Cette phase comprenait la rédaction des différentes sections de l'article, l'intégration des graphiques et des tableaux de résultats, et la vérification de la cohérence et de la clarté du contenu.

Enfin, nous avons préparé la soutenance prévue pour le 16 janvier 2025. Cette préparation incluait la création des diapositives de présentation, la répétition de la présentation orale, et la préparation des réponses aux questions potentielles.

Cette méthodologie structurée nous a permis de mener à bien notre projet d'optimisation en utilisant l'algorithme ABC, tout en assurant une analyse rigoureuse et une implémentation efficace dans deux environnements de programmation différents.

IV. DESCRIPTION DES ALGORITHMES

A. Processus de l'algorithme ABC

Pour comprendre comment l'algorithme ABC fonctionne, il est essentiel de définir certaines notions utilisées dans l'algorithme.

Fonction objective dite fitness: mesure quantitative qui évalue la qualité des solutions trouvées par les abeilles. Dans le contexte de l'ABC, cette fonction représente la quantité de nectar (ou la valeur de la solution) à une position donnée dans l'espace de recherche.

Source de nourriture: correspond à une solution potentielle au problème d'optimisation

Abeilles : représentent des solutions potentielles au problème d'optimisation. Chaque abeille est caractérisée par un vecteur de paramètre qui correspond à une configuration spécifique des variables de décision du problème.

Il existe 3 types d'abeilles dans l'algorithme ABC:

- **abeilles employées**: Représentent les agents qui exploitent une solution particulière. Leur nombre est égal au nombre de solutions en cours d'exploration.

- **abeilles observatrices**: Choisissent les meilleures solutions en fonction de leur fitness pour une exploitation ultérieure.

- **abeilles éclaireuses**: Génèrent de nouvelles solutions aléatoires dans l'espace de recherche, remplaçant celles qui stagnent.

Mathématiquement:

Supposons que nous voulons optimiser une fonction $f(x_1,x_2)$ à deux variables x_1,x_2.

Chaque abeille employée a_i représente une paire de valeurs (x_1,x_2) dans l'espace de recherche. Les abeilles observatrices évaluent ces paires en utilisant la fonction de fitness $f(x_1,x_2)$. Si une solution ne peut plus être améliorée après un certain nombre d'itérations, l'abeille employée devient une abeille éclaireuse et explore de nouvelles régions (x_1,x_2) de l'espace de recherche de manière aléatoire.

Voici dans un premier temps le schéma général que l'algorithme ABC suit:

Phase d'initialisation

RÉPÉTER

Phase des abeilles employées
Phase des abeilles observatrices
Phase des abeilles éclaireuses
Mémoriser la meilleure solution obtenue jusqu'à present

JUSQU'À (Critère d'arrêt satisfait ou Nombre max d'itérations atteint)

Figure 2: schéma général de l'algorithme ABC

Phase d'initialisation

1. Création de la population initiale:
   - Génération aléatoire d'un ensemble de solutions, chaque solution représentant une source de nourriture.
   - Les abeilles employées sont associées à ces solutions. Le nombre total d'abeilles employées est égal au nombre de sources de nourriture.
   - Représentation mathématique :

$$U_j = U_j^{\{min\}} + r \cdot \left(U_j^{\{max\}} - U_j^{\{min\}}\right), r \in [0,1]$$

   où U_j est une dimension du vecteur de solution.

2. Évaluation de la fitness:
   - Chaque solution est évaluée en fonction d'une fonction objectif.
   - La fitness, souvent inversement proportionnelle à la valeur de la fonction objectif, est calculée pour mesurer la qualité de la solution.

3. Paramètres de l'algorithme :
   - Population totale: Nombre d'abeilles employées + spectatrices.
   - Limite d'abandon: Nombre maximal d'itérations sans amélioration avant d'abandonner une solution.
   - Critère d'arrêt : Nombre maximal d'itérations ou seuil de convergence.

Phase des abeilles employées

→ Objectif: Explorer le voisinage des solutions actuelles pour les améliorer.

1. Mouvement des abeilles :
   - Chaque abeille génère une nouvelle solution en modifiant légèrement sa solution actuelle.
   - Le déplacement est donné par:

$$x_{\{ij\}}^{\{t+1\}} = x_{\{ij\}}^{\{t\}} + \phi_{\{ij\}} * \left(x_{\{ij\}}^{\{t\}} - x_{\{kj\}}^{\{t\}}\right)$$

   où :
   - $x_{\{ij\}}^{\{t\}}$: Position actuelle.
   - $x_{\{kj\}}^{\{t\}}$: Position d'une autre solution aléatoire.
   - $\phi_{\{ij\}}$: Facteur aléatoire dans [-1, 1].

2. Évaluation et sélection :
   - Si la nouvelle solution est meilleure, elle remplace l'ancienne.
   - Sinon, le compteur d'échecs de cette solution est incrémenté.

Phase des abeilles observatrices

→ Objectif: Sélectionner les solutions les plus prometteuses pour une exploration plus poussée.

1. Sélection basée sur la fitness:
   - Une probabilité est calculée pour chaque solution en fonction de sa fitness:

$$P_i = 0.9 * \left(\frac{Fitness_i}{\max(Fitness)}\right) + 0.1$$

   - Les solutions avec une meilleure fitness ont une plus grande chance d'être sélectionnées.

2. Amélioration des solutions:
   - Les observatrices génèrent des nouvelles solutions en utilisant la même équation de déplacement que les employées.
   - Si la nouvelle solution est meilleure, elle remplace l'ancienne.

Phase des abeilles éclaireuses

→ Objectif: Explorer de nouvelles zones de l'espace de recherche en remplaçant les solutions stagnantes.

1. Abandon des solutions stagnantes: Si une solution n'est pas améliorée après un certain nombre d'itérations, elle est abandonnée.

2. Génération de nouvelles solutions: Une nouvelle solution est générée aléatoirement dans l'espace de recherche:

$$x_{\{ij\}} = x_{\{ij\}}^{\{min\}} + r * \left(x_{\{ij\}}^{\{max\}} - x_{\{ij\}}^{\{min\}}\right), r \in [0,1]$$

Mémorisation de la meilleure solution

Mise à jour continue:

- La meilleure solution trouvée jusqu'à présent est mémorisée.
- Cela permet de conserver la meilleure solution même si elle n'est pas modifiée dans les itérations suivantes.

Critère d'arrêt

➔ L'algorithme se termine lorsqu'un des critères suivants est atteint:
- Nombre maximal d'itérations.
- Convergence à une solution satisfaisante en termes de fitness.

B. Description de l'algorithme Python étudié

Dans cette partie, nous allons détailler chaque partie du code python.

Importation des bibliothèques et classes

```
7  import numpy as np
8  from mealpy.optimizer import Optimizer
```

➔ **numpy**: utilisé pour les calculs mathématique avancés

➔ **Optimizer**: classe parente provenant de la bibliothèque **mealpy**, servant de base pour la classe **OriginalABC**

Définition de la classe OriginalABC

```
11  class OriginalABC(Optimizer):
```

➔ Hérite de la classe **Optimizer**, qui fournit des fonctionnalités de base pour les algorithmes d'optimisation

Constructeur de la classe **OriginalABC**

1. Paramètres du constructeur :
   - epoch : nombre maximal d'itérations
   - pop_size : taille de la population
   - n_limits : nombre maximal de tentatives
   - **kwargs : paramètres supplémentaires à transmettre à la classe parente **Optimiser**

2. Validation de paramètres : les lignes ci-dessous valident les valeurs en vérifiant qu'elles respectent les plages spécifiées

```
53          self.epoch = self.validator.check_int("epoch", epoch, [1, 100000])
54          self.pop_size = self.validator.check_int("pop_size", pop_size, [5, 10000])
55          self.n_limits = self.validator.check_int("n_limits", n_limits, [1, 1000])
```

3. Autres attributs :
   - "self.is_parallelizable = False": indique que cet algorithme n'est pas conçu pour un traitement parallèle
   - "self.set_parameters(["epoch", "pop_size", "n_limits"])" : enregistre les paramètres dans la classe
   - "self.sort_flag = False" : utilisé pour désactiver le tri des solutions

Initialisation des variables

```
60      def initialize_variables(self):
61          self.trials = np.zeros(self.pop_size)
```

➔ Initialise un tableau “trials” de taille “pop_size”, contenant le nombre de tentatives pour chaque abeilles

Méthode principale “evolve”

➔ C’est ici que les étapes principales de l’algorithme ABC sont exécutées. Elle prend un paramètre “epoch”, correspondant à l’itération actuelle.

1. <u>Phase des abeilles employées</u>

```
70      for idx in range(0, self.pop_size):
71          # Choose a random employed bee to generate a new solution
72          rdx = self.generator.choice(list(set(range(0, self.pop_size)) - {idx}))
73          # Generate a new solution by the equation x_{ij} = x_{ij} + phi_{ij} * (x_{tj} - x_{ij})
74          phi = self.generator.uniform(low=-1, high=1, size=self.problem.n_dims)
75          pos_new = self.pop[idx].solution + phi * (self.pop[rdx].solution - self.pop[idx].solution)
76          pos_new = self.correct_solution(pos_new)
```

➔ Chaque abeille génère une nouvelle solution:
- Une abeille aléatoire différente (index “rdx”) est sélectionnée.
- Un vecteur directionnel aléatoire “phi” est généré
- La nouvelle position “pos_new” est calculée par une formule tenant compte des solutions actuelles et sélectionnées

```
77          agent = self.generate_agent(pos_new)
78          if self.compare_target(agent.target, self.pop[idx].target, self.problem.minmax):
79              self.pop[idx] = agent
80              self.trials[idx] = 0
81          else:
82              self.trials[idx] += 1
```

➔ Puis une nouvelle solution est générée (“generate_agent”) et évaluée :
- Si elle est meilleure, elle remplace l’ancienne solution et le compteur de tentatives est remis à zéro
- Sinon, le compteur de tentatives est incrémenté

2. <u>Phase des abeilles observatrices</u>

```
85      employed_fits = np.array([agent.target.fitness for agent in self.pop])
86      # probabilities = employed_fits / np.sum(employed_fits)
87      for idx in range(0, self.pop_size):
88          # Select an employed bee using roulette wheel selection
89          selected_bee = self.get_index_roulette_wheel_selection(employed_fits)
```

➔ Les abeilles observatrices sélectionnent les solutions à explorer selon leur probabilité, calculée à partir de la fitness des abeilles employées.

```
91          rdx = self.generator.choice(list(set(range(0, self.pop_size)) - {idx, selected_bee}))
92          # Generate a new solution by the equation x_{ij} = x_{ij} + phi_{ij} * (x_{tj} - x_{ij})
93          phi = self.generator.uniform(low=-1, high=1, size=self.problem.n_dims)
94          pos_new = self.pop[selected_bee].solution + phi * (self.pop[rdx].solution - self.pop[selected_bee].solution)
95          pos_new = self.correct_solution(pos_new)
```

➔ Une nouvelle solution est calculée de manière similaire à la phase des abeilles employées

3. Phase des abeilles éclaireuses

```
104     abandoned = np.where(self.trials >= self.n_limits)[0]
105     for idx in abandoned:
106         self.pop[idx] = self.generate_agent()
107         self.trials[idx] = 0
```

➔ Si une abeille a dépassé sa limite de tentatives, elle abandonne sa source actuelle. Une nouvelle solution est générée pour elle.

Méthodes auxiliaires

- “correct_solution(pos_new) “: Corrige la solution pour s'assurer qu'elle reste dans les bornes du problème.
- “generate_agent()” : Crée une solution aléatoire ou basée sur une position donnée.
- “compare_target()” : Compare deux solutions pour voir laquelle est meilleure.
- “get_index_roulette_wheel_selection()” : Implémente un mécanisme de sélection

C. Description de l'algorithme C++ implémenté

Nous avons voulu respecter les noms employés dans le code Python et nous avons aussi gardé la logique des classes. Dans cette partie, nous ne montrons pas le code. Il nous a été demandé de subdiviser le code en 3 fichiers :

- Le Header (OriginalABC.h) : définit les classes **Agent** (une solution) et **OriginalABC** (l'algorithme)
- L'implémentation (Original.cpp) : contient les définitions des fonctions où notre analyse va se porter
- Le main (main.cpp) : contient les tests et les fonctions qui ont été testés

Importations des bibliothèques

- <algorithm>: pour les fonctions utilitaires comme std::max
- <cmath>: pour les calculs mathématiques
- <ctime>: pour la génération de nombre aléatoires
- <limites>: pour définir les valeurs maximales et minimales des types numériques

Constructeur de la classe Agent

➔ Initialise un agent (une abeille) avec:
  - "solution": un vecteur de taille "dim" rempli de zéros
  - "fitness": une variable initialisée à la valeur maximale d'un **double** car l'objectif est de minimiser la function

Classe OriginalABC

1. Constructeur de la classe

➔ Initialise l'algorithme :
  - "epoch": nombre maximal d'itérations
  - "pop_size": nombre total d'abeilles
  - "n_limits": nombre maximal de tentatives avant que l'abeille éclaireuse intervienne

➔ Génère des résultats aléatoires pouvant être reproductibles

2. Méthode "generate_new_solution"
  - Génère une nouvelle solution en modifiant chaque dimension de la solution actuelle avec un coefficient aléatoire "phi"

3. Méthodes auxiliaires
  - "correct_solution": ajoute une solution pour qu'elle reste dans les bornes définies
  - "uniform": génère un nombre aléatoire dans un intervalle donné [min,max]
  - "compare_target": compare deux fitness pour déterminer si une solution est meilleure
  - "get_best_fitness": retourne la meilleure valeur de fitness parmi les agents

4. Méthode "initialize_variables"

➔ Initialise:
  - Les bornes des variables
  - Les abeilles de la population avec des solutions aléatoires dans les bornes définies
  - Les compteurs de tentatives pour chaque abeille à 0

➔ Utilise la méthode "uniform" pour générer des valeurs aléatoires dans les bornes

5. Méthode "solve"

➔ La méthode solve est le **cœur de l'algorithme ABC**.

➔ Elle orchestre les différentes phases:

1. **Évaluation initiale des solutions.**
2. **Exploration des solutions avec les abeilles employées.**
3. **Gestion des solutions abandonnées par les éclaireuses.**
4. **Amélioration progressive des solutions jusqu'à atteindre un critère d'arrêt.**

C'est cette méthode qui doit être appelée pour optimiser une fonction objective donnée. Nous allons maintenant détailler chaque partie de cette méthode.

➔ Phase des abeilles employees
  - Les abeilles employées explorent l'espace des solutions possibles en générant de nouvelles solutions proches de celles qu'elles connaissent déjà
  - Chaque abeille employée sélectionne un voisin (autre abeille) et génère une nouvelle solution basée sur cette relation.
  - La nouvelle solution est ajustée avec "correct_solution" pour rester dans les bornes du problème.
  - Si la nouvelle solution est meilleure, elle remplace l'ancienne. Sinon, le compteur d'échecs (trials) est incrémenté.

➔ Phase des abeilles observatrices
  - Les abeilles spectatrices choisissent une source à explorer en fonction de la qualité des solutions des abeilles employées.
  - Bien que non explicitement séparée dans ce code, cette phase est incluse dans le calcul des solutions par les abeilles employées.

➔ Phase des abeilles éclaireuses
  - Les abeilles éclaireuses interviennent lorsqu'une source est abandonnée après trop d'échecs (définis par n_limits)
  - Si une solution atteint la limite d'échecs, elle est remplacée par une nouvelle solution générée aléatoirement dans les bornes.

D. *Comparaison structurelle entre les deux algorithmes*

Pour montrer clairement les différences entre les implémentations en C++ et Python de l'algorithme ABC, nous avons décidé de les regrouper dans le tableau ci-dessous:

Aspect	Python	C++
Structure du code	Un seul fichier (**OriginalABC.py**) centralise tout.	Deux fichiers : **OriginalABC.h** (déclarations) et **OriginalABC.cpp** (implémentations).
Représentation d'une solution	Liste de vecteurs **NumPy** dans "self.pop".	Classe **Agent** contenant un vecteur solution et un fitness
Initialisation des solutions	Réalisée via "initialize_variables" en utilisant NumPy pour générer des valeurs aléatoires.	Réalisée dans "initialize_variables" en utilisant une boucle et std::rand.
Phase des abeilles observatrices	Incluse implicitement dans "evolve" avec des probabilités calculées à partir des fitness.	Implémentée indirectement dans "solve" sans calcul explicite des probabilités.
Génération de nouvelles solutions	Opérations vectorielles avec NumPy pour modifier les solutions.	Méthode "generate_new_solution " itérant sur chaque dimension du vecteur.
Validation des paramètres	Validation explicite des valeurs via "self.validator" avec des bornes définies.	Validation implicite, la responsabilité incombe à l'utilisateur.
Typage des données	Dynamique avec annotations facultatives (epoch: int, etc.).	Statique avec des types explicitement déclarés dans le fichier d'en-tête.

V. DESCRIPTION DES FONCTIONS OBJECTIFS UTILISÉES

Pour évaluer les performances de nos algorithmes, il nous a été demandé de les tester sur les fonctions Rastrigin, Ackley, Schwefel et Rosenbrock. Ces fonctions sont très populaires pour évaluer l'optimisation d'algorithmes car elles couvrent des scénarios variés, allant des minima locaux (Rastrigin, Ackley, Schwefel) à la précision requise pour suivre des trajectoires complexes (Rosenbrock). Nous détaillerons dans la partie suivante comment ont été utilisées ces fonctions et quels résultats nous en avons tiré. Nous allons dans cette partie donner une description détaillée de ces fonctions.

A. *Rastrigin*

Utilisation: utilisée pour tester la capacité des algorithmes d'optimisation à éviter les pièges des optimums locaux et à converger vers le minimum global dans un espace de recherche multidimensionnel.
Formule:

$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

Caractéristiques:

- Dimension: d (nombre de variables)
- Domaine: $x_i \in [-5.12,5.12]$
- Minimum global: $f(x^*) = 0$ à $x^* = (0,0,...,0)$

B. *Ackley*

Utilisation: tester la capacité des algorithmes d'optimisation à naviguer dans un espace de recherche avec de nombreux minima locaux et à converger vers le minimum global.
Formule:

$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Caractéristiques:

- Dimension: d (nombre de variables)
- Domaine: $x_i \in [-32.768,32.768]$
- Minimum global: $f(x^*) = 0$ à $x^* = (0,0,...,0)$

C. *Schwefel*

Utilisation: tester la capacité des algorithmes d'optimisation à trouver le minimum global dans un espace de recherche avec de nombreux minima locaux.
Formule:

$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

Caractéristiques:

- Dimension: d (nombre de variables)
- Domaine: $x_i \in [-500,500]$
- Minimum global: $f(x^*) = 0$ à $x^* = (420.9687,420.9687,...,420.9687)$

D. Rosenbrock

Utilisation: tester la capacité des algorithmes d'optimisation à converger vers le minimum global dans une vallée étroite et parabolique. Elle est particulièrement utile pour évaluer la précision et la robustesse des algorithmes d'optimisation.
Formule:

$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Caractéristiques:

- Dimension: d (nombre de variables)
- Domaine: $x_i \in [-5,10]$
- Minimum global: $f(x^*) = 0$ à $x^* = (1,1,...,1)$

VI. RÉSULTATS EXPÉRIMENTAUX ET DISCUSSION

Dans un premier temps, pour réaliser ces tests, nous avons implémenté les fonctions objectifs en C++ et en Python (voir annexe 1 à 4, page 12). L'objectif était de garantir une comparaison précise entre les deux langages en conservant les mêmes paramètres et méthodologies d'implémentation. Pour faciliter la collecte et le traitement des résultats, nous avons fait en sorte de les enregistrer systématiquement dans des fichiers au format .txt. Cette organisation nous a permis de reprendre et analyser facilement les données ultérieures.

Ensuite, en faisant varier le paramètre "n_limits", nous avons obtenus une série de valeurs qui représentent les performances de chaque fonction objectif dans différentes configurations. Ces valeurs, incluant les moyennes et écart-types des résultats, ont été consolidées et stockées dans un fichier au format .csv (voir annexe 5 page 13). Ce format a été choisi pour sa facilité d'intégration avec des outils d'analyse et de visualisation comme Python.

Dans ce tableau, une première constatation s'impose: les valeurs trouvées pour chaque fonction sont logiques entre les deux implémentations, en Python et en C++. Bien que quelques différences soient visibles, elles restent cohérentes et s'expliquent par les variations d'implémentation entre les deux langages.

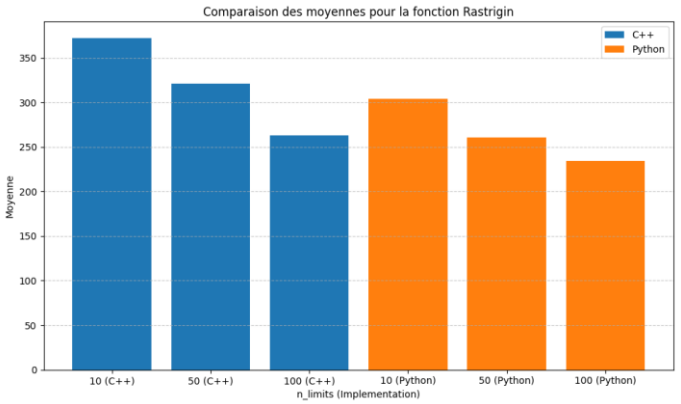
Par la suite, nous avons écrit une fonction en Python (voir annexes 6 et 7, page 13) spécialement conçue pour créer et afficher des graphiques. Cette fonction utilise les données préalablement stockées dans le fichier .csv et produit des visualisations claires et comparatives pour chaque fonction objectif. Elle permet notamment de tracer des graphiques pour les moyennes ainsi que pour les écart-types, en fonction des différentes valeurs de "n_limits". Cet outil s'est révélé précieux pour analyser rapidement et efficacement les performances des fonctions implémentées.

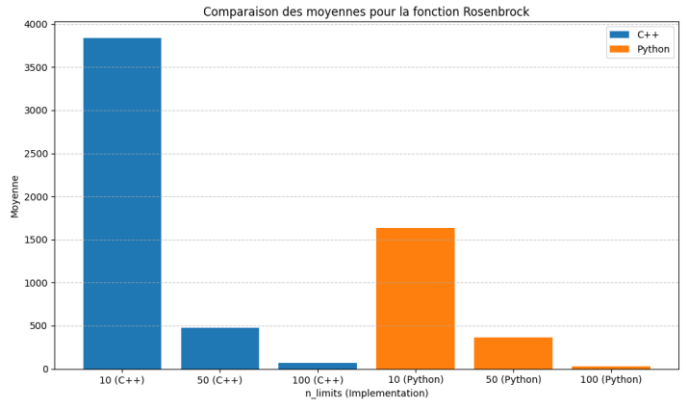
Nous allons maintenant analyser chaque graphique afin d'étudier les moyennes et écart-types obtenus pour chaque fonction objectif. Cette analyse permettra d'identifier les tendances et différences entre les implémentations en C++ et Python, et de mieux comprendre l'impact des variations du paramètre "n_limits" sur les résultats.

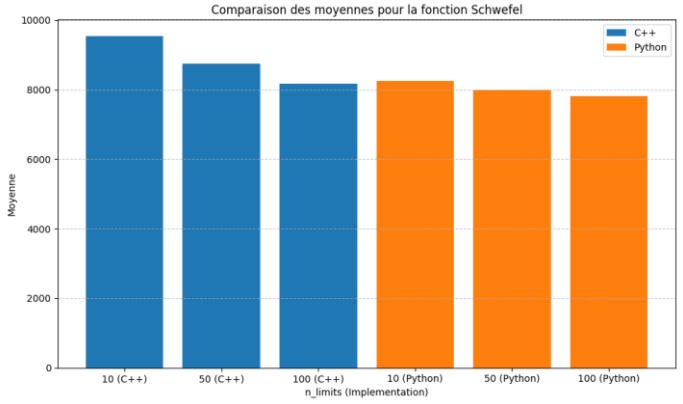
A. Analyse graphique des moyennes

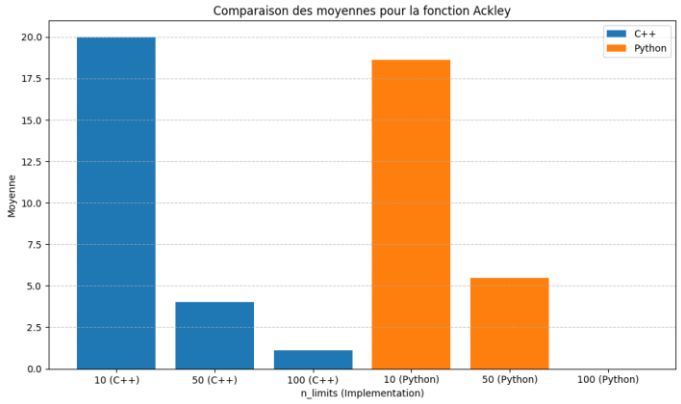
Pour les différentes fonctions objectifs (Rosenbrock, Rastrigin, Ackley, et Schwefel), les moyennes obtenues selon la valeur de "n_limits" présentent des variations qui illustrent les performances des deux implémentations (C++ et Python) dans le cadre de l'algorithme ABC.
Les graphiques que nous avons réalisés sont des graphiques en Barres des Moyennes qui ont pour objectif de comparer les moyennes obtenues par les deux algorithmes pour chaque fonction objectif et et chaque valeur de "n_limits".

Voici dans un premier temps les graphiques que nous avons obtenu:









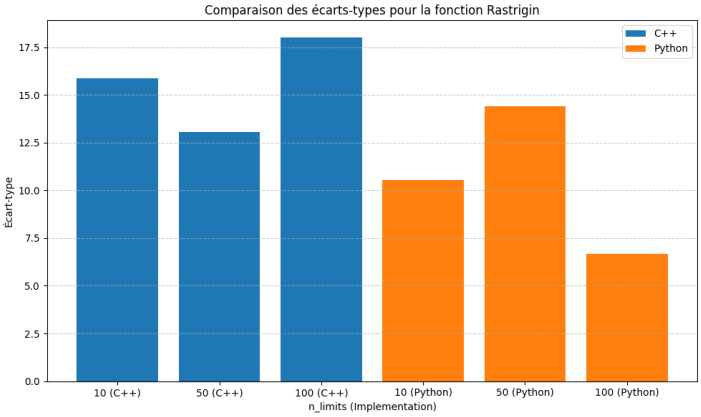
Les graphiques montrent que l'implémentation en Python de l'algorithme ABC semble offrir une meilleure performance que celle en C++ pour toutes les fonctions objectifs testées (Rosenbrock, Rastrigin, Ackley, Schwefel) et pour toutes les valeurs de “n_limits” (10, 50, 100). La différence de performance est plus marquée pour les valeurs plus faibles de “n_limits”et diminue légèrement pour les valeurs plus élevées de “n_limits”. Ces résultats suggèrent que l'implémentation en Python pourrait être plus efficace pour ces fonctions objectifs.

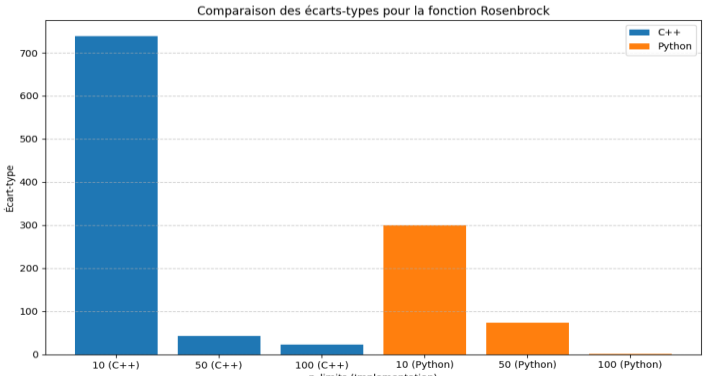
B. Analyse graphique des écart-types

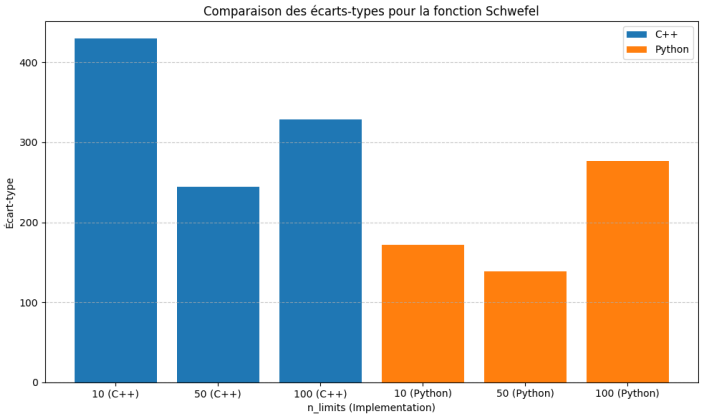
De même que pour les moyennes, les écart-type obtenues selon la valeur de “n_limits” présentent des variations qui illustrent les performances des deux implémentations (C++ et Python) dans le cadre de l'algorithme ABC.

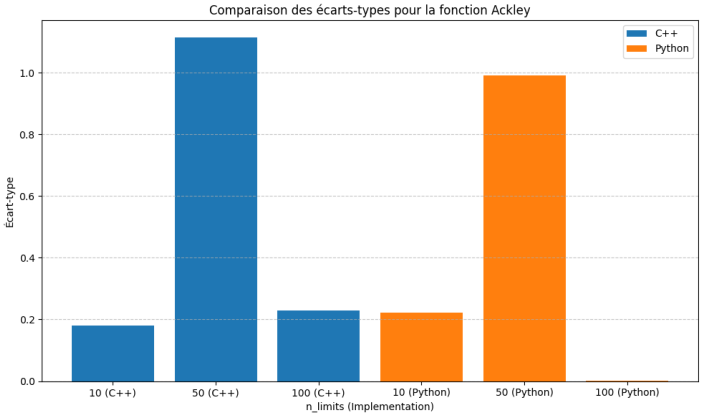
Les graphiques que nous avons réalisés sont aussi des graphiques en Barres qui ont pour objectif de comparer les écart-types obtenues par les deux algorithmes pour chaque fonction objectif et et chaque valeur de “n_limits”.

Voici dans un premier temps les graphiques que nous avons obtenu:









Les mêmes constatations se font : l'implémentation en Python de l'algorithme ABC est généralement plus performante, surtout pour les fonctions avec des paysages complexes comme Rosenbrock et Ackley. Cependant, pour des fonctions comme Schwefel, C++ offre une meilleure robustesse pour des “n_limits” élevées.

C. Dissucsion

A. Impact de la complexité des fonctions objectifs

Les performances varient de manière significative selon la fonction objective utilisée. La fonction Rosenbrock, connue pour ses propriétés de vallées étroites, a présenté des moyennes plus élevées, notamment pour de faibles valeurs de “n_limits”. Cela reflète la difficulté de l’algorithme à converger rapidement vers des solutions optimales pour cette fonction. En revanche, la fonction Ackley, qui combine des oscillations fréquentes avec un optima global clairement défini, a montré des moyennes nettement plus faibles, signalant une convergence rapide et efficace, surtout pour “n_limits” = 100.

Les fonctions Rastrigin et Schwefel se situent dans une zone intermédiaire. La première, avec ses nombreux optima locaux, a mis en évidence une performance relativement stable mais légèrement inférieure à celle obtenue pour la fonction Ackley. La fonction Schwefel, plus complexe, a maintenu des moyennes élevées, particulièrement en C++, suggérant une difficulté accrue pour explorer efficacement l’espace de recherche.

B. Comparaison des implémentations : C++ vs Python

1. Moyennes et stabilité

Les résultats révèlent une supériorité notable de l’implémentation Python sur celle en C++ pour la plupart des fonctions objectives, notamment en termes de moyennes obtenues. Cette différence est particulièrement marquée pour la fonction Rosenbrock, où Python atteint des moyennes très faibles (26.78 pour “n_limits” = 100) contre 68.01 pour C++. Cela peut être attribué à des optimisations internes dans l’implémentation Python ou à une meilleure gestion des calculs numériques pour certaines configurations.
Cependant, l’implémentation en C++ a montré une meilleure stabilité (écarts-types plus faibles) pour certaines fonctions, comme Schwefel et Rastrigin. Cela pourrait s’expliquer par une gestion plus rigoureuse de la mémoire et des calculs en C++, réduisant les fluctuations d’exécution.

2. Temps d’attente des réponses

Une différence notable entre les deux implémentations réside dans le temps nécessaire pour obtenir une réponse. L’implémentation en C++ est significativement plus rapide que celle en Python. Ce constat s’explique par plusieurs facteurs :

Performance intrinsèque des langages : C++ est un langage compilé, ce qui lui confère un avantage en termes de rapidité d’exécution. Python, en revanche, est interprété, ce qui engendre une surcharge due à la traduction du code en instructions machine à chaque exécution.
Gestion de la mémoire et des structures de données : C++ permet un contrôle plus précis sur la gestion de la mémoire, évitant ainsi les surcoûts liés à des mécanismes comme le garbage collection, présent en Python.
Optimisations spécifiques : Les bibliothèques standard de C++ (comme <vector> ou <algorithm>) sont souvent mieux optimisées pour des calculs intensifs que les équivalents en Python.
Ainsi, bien que Python offre une meilleure convivialité et une plus grande simplicité pour la mise en œuvre, C++ se distingue par sa rapidité, ce qui peut être crucial dans des scénarios où le temps de réponse est une contrainte importante, comme les systèmes embarqués ou les applications en temps réel.

C. Influence de n_limits

L'augmentation de la valeur de n_limits a eu un effet positif sur les moyennes, avec une amélioration notable pour toutes les fonctions objectives. Par exemple, pour la fonction Ackley, la moyenne passe de 19.97 (C++, “n_limits” = 10) à 1.09 (C++, “n_limits” = 100). Cela illustre la capacité de l’algorithme à optimiser davantage ses solutions lorsqu’il dispose de plus de ressources de calcul. Toutefois, l’amélioration marginale diminue au-delà de certaines valeurs de “n_limits”, comme observé pour la fonction Schwefel.

D. Limites de l’étude

Malgré des résultats prometteurs, certaines limitations doivent être reconnues. D’abord, les tests ont été effectués sur un nombre limité de fonctions objectives, ce qui peut restreindre la généralisation des conclusions à d'autres problèmes d’optimisation. Ensuite, les implémentations C++ et Python, bien que similaires dans leur structure algorithmique, pourraient inclure des différences subtiles qui influencent les performances, notamment en termes de gestion des calculs numériques. Enfin, le temps d’attente des réponses pourrait être influencé par des facteurs externes, tels que la configuration matérielle ou le compilateur utilisé pour C++.

E. Perspectives et implications

Les résultats de cette étude soulignent l’importance du choix de l’implémentation et de la configuration des paramètres dans l’application pratique des algorithmes d’optimisation. Pour des problèmes nécessitant une grande précision et une stabilité accrue, l’utilisation de C++ pourrait être avantageuse, notamment pour des systèmes nécessitant une réponse rapide. En revanche, pour des besoins de convergence rapide et des solutions globales de meilleure qualité, Python semble plus adapté.

VII. Conclusion

Ce rapport a exploré les performances comparatives de l'algorithme d'optimisation Artificial Bee Colony (ABC) implémenté en C++ et en Python, en utilisant quatre fonctions objectives emblématiques : Rosenbrock, Rastrigin, Ackley et Schwefel. Les résultats obtenus offrent des perspectives riches en enseignements, à la fois sur la qualité des solutions produites, leur stabilité, et sur le temps de réponse des implémentations.

L'analyse a révélé que Python, grâce à sa simplicité et sa flexibilité, tend à produire des solutions globalement meilleures pour certaines fonctions complexes comme Rosenbrock et Ackley, tout en présentant des moyennes plus faibles. Cependant, l'implémentation en C++ se démarque par sa rapidité d'exécution, essentielle dans des scénarios où le temps de réponse est critique, et par une stabilité accrue pour des fonctions comme Schwefel et Rastrigin, due à une gestion plus fine de la mémoire et des calculs.

Le choix entre ces deux implémentations dépend donc des priorités spécifiques d'un projet : Python se révèle particulièrement adapté pour des applications nécessitant une convergence rapide et des solutions de qualité globale, tandis que C++ s'impose dans des contextes exigeant des calculs intensifs et des réponses en temps réel.

Enfin, cette étude ouvre la voie à plusieurs pistes futures, notamment l'extension des tests à d'autres fonctions objectives et la mise en œuvre de stratégies d'optimisation adaptées à chaque langage. Ces travaux constituent une base précieuse pour guider les choix techniques dans le domaine de l'optimisation, offrant un équilibre entre performance, rapidité et précision.

En somme, les résultats soulignent l'importance d'un choix réfléchi des outils et des paramètres en fonction des exigences spécifiques de l'application, tout en mettant en lumière les forces respectives de Python et C++ pour répondre à des besoins diversifiés en optimisation.

Remerciements

Nous souhaitons exprimer notre profonde gratitude à notre professeur, **M. IDOUMGHAR Lhassane**, pour son accompagnement et ses conseils avisés tout au long de ce projet. Son expertise, sa disponibilité, et ses encouragements constants ont été essentiels à la réussite de ce travail.

Nous lui sommes particulièrement reconnaissants d'avoir partagé généreusement ses connaissances, son expérience et d'avoir répondu à nos interrogations avec patience et bienveillance. Son engagement à créer un environnement d'apprentissage motivant et stimulant nous a permis d'approfondir nos compétences et de progresser avec assurance dans le cadre de ce projet.

Sources

Surjanovic, S & Bingham, D (2013). Site : *Virtual Library of Simulation Experiments : Test Functions and Datasets*. https://www.sfu.ca/~ssurjano/index.html

Karaboga, D (2011). Article : *Artificial bee colony algorithm*.http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm

Debret, J. (2020, 28 avril). *Structure de l'article scientifique et conseils essentiels*. Scribbr. https://www.scribbr.fr/article-scientifique/structure-article-scientifique/

Mouassa, S. (2012,19 juin). Mémoire : *Optimisation de l'écoulement de puissance par une méthode métaheuristique (technique des abeilles) en présence d'une source renouvelable (éolienne) et des dispositifs FACTS*.http://dspace.univ-setif.dz:8888/jspui/bitstream/123456789/2080/1/Mémoire%20Magister%20Souhil%20%20final%20version.pdf

Dik, A.(2023). Article : *Algorithmes d'optimisation de la population : Colonie d'Abeilles Artificielles (ABC)*.https://www.mql5.com/fr/articles/11736

Cours d'IA de M.IDOUMGHAR Lhassane

ANNEXES

```python
# Définir la fonction Rosenbrock
def rosenbrock(solution):  1 usage
    d = len(solution)
    sum_val = 0
    for i in range(d - 1):  # Parcours de 0 à d-2
        xi = solution[i]
        xnext = solution[i + 1]
        sum_val += 100 * (xnext - xi**2)**2 + (xi - 1)**2
    return sum_val

# Définir la fonction Rastrigin
def rastrigin(solution):  1 usage
    d = len(solution)
    sum = 0.0
    for s in solution:
        sum += (s * s - 10 * math.cos(2 * math.pi * s))
    return 10 * d + sum
```

```python
# Définir la fonction Ackley
def ackley(solution):  1 usage
    a = 20
    b = 0.2
    c = 2 * np.pi

    d = len(solution)
    sum1 = np.sum(solution ** 2)
    sum2 = np.sum(np.cos(c * solution))
    term1 = -a * np.exp(-b * np.sqrt(sum1 / d))
    term2 = -np.exp(sum2 / d)
    return term1 + term2 + a + np.exp(1)

# Définir la fonction Schwefel
def schwefel(solution):  1 usage
    d = len(solution)
    sum_value = np.sum(solution * np.sin(np.sqrt(np.abs(solution))))
    return 418.9829 * d - sum_value
```

Annexe 1 & 2: code Python des fonctions objectifs

```cpp
// Fonctions d'optimisation
double rosenbrock(const std::vector<double>& solution) {
    double sum_val = 0.0;
    for (size_t i = 0; i < solution.size() - 1; ++i) {
        double xi = solution[i];
        double xnext = solution[i + 1];
        sum_val += 100.0 * std::pow(xnext - xi * xi, 2) + std::pow(xi - 1, 2);
    }
    return sum_val;
}

double rastrigin(const std::vector<double>& solution) {
    double sum = 0.0;
    for (double s : solution) {
        sum += s * s - 10.0 * std::cos(2.0 * M_PI * s);
    }
    return 10.0 * solution.size() + sum;
}
```

Annexe 3: code C++ des fonctions objectifs Rosenbrock & Rastrigin

```cpp
double ackley(const std::vector<double>& solution) {
    const double a = 20.0;
    const double b = 0.2;
    const double c = 2.0 * M_PI;

    double sum1 = 0.0, sum2 = 0.0;
    for (double x : solution) {
        sum1 += x * x;
        sum2 += std::cos(c * x);
    }
    size_t d = solution.size();
    return -a * std::exp(-b * std::sqrt(sum1 / d)) - std::exp(sum2 / d) + a + std::exp(1);
}

double schwefel(const std::vector<double>& solution) {
    double sum_value = 0.0;
    for (double x : solution) {
        sum_value += x * std::sin(std::sqrt(std::abs(x)));
    }
    return 418.9829 * solution.size() - sum_value;
}
```

Annexe 4: code C++ des fonctions objectifs Ackley & Schwefel

	Function_objective	Implementation	n_limits	Mean	Std
2	Ackley	C++	10	19.9788	0.1801
3	Ackley	C++	50	4.02969	1.11447
4	Ackley	C++	100	1.09679	0.2279
5	Ackley	Python	10	18.6152	0.2228
6	Ackley	Python	50	5.4733	0.9905
7	Ackley	Python	100	0.0032	0.0010
8	Rastrigin	C++	10	371.862	15.8858
9	Rastrigin	C++	50	320.893	13.069
10	Rastrigin	C++	100	262.654	18.0157
11	Rastrigin	Python	10	303.9353	10.5480
12	Rastrigin	Python	50	260.5063	14.3951
13	Rastrigin	Python	100	234.6294	6.6563
14	Rosenbrock	C++	10	3835.43	738.82
15	Rosenbrock	C++	50	473.157	43.1156
16	Rosenbrock	C++	100	68.0116	22.3002
17	Rosenbrock	Python	10	1633.2163	299.0246
18	Rosenbrock	Python	50	363.4298	73.1742
19	Rosenbrock	Python	100	26.7841	0.1758
20	Schwefel	C++	10	9531.81	430.158
21	Schwefel	C++	50	8743.72	244.518
22	Schwefel	C++	100	8169.22	328.611
23	Schwefel	Python	10	8243.39	171.3882
24	Schwefel	Python	50	7996.34	138.8468
25	Schwefel	Python	100	7821.09	276.2724

Annexe 5 : Tableau des résultats des moyennes (Mean) et écart-types (Std) pour chaque fonction objectif sur Excel

```python
import pandas as pd
import matplotlib.pyplot as plt

# Charger les données
data = pd.read_csv("data.csv")

# Liste des fonctions objectives
functions = data["Function_objective"].unique()

# Générer un graphique pour chaque fonction
for function in functions:
    plt.figure(figsize=(10, 6))

    # Filtrer les données pour la fonction actuelle
    function_data = data[data["Function_objective"] == function]

    # Générer des barres pour chaque implementation (C++ et Python)
    for implementation in ["C++", "Python"]:
        impl_data = function_data[function_data["Implementation"] == implementation]
        plt.bar(
            impl_data["n_limits"].astype(str) + f" ({implementation})",
            impl_data["Std"],
            label=f"{implementation}"
        )
```

```python
    # Ajouter des détails au graphique
    plt.title(f"Comparaison des écarts-types pour la fonction {function}")
    plt.xlabel("n_limits (Implementation)")
    plt.ylabel("Écart-type")
    plt.legend()
    plt.grid(axis="y", linestyle="--", alpha=0.7)

    # Enregistrer le graphique ou l'afficher
    plt.tight_layout()
    plt.savefig(f"graph_{function}.png")
    plt.show()
```

Annexes 6 & 7 : code Python pour la création des graphiques des moyennes et écart-types