Algorithme Artificial Bee Colony (ABC)

Laoualy Chaibou Habsatou
Université de Haute-Alsace, Mulhouse, France
habsatou.laoualy-chaibou@uha.fr

Soule Aarafat
Université de Haute-Alsace, Mulhouse, France
aarafat.soule@uha.fr

Chaki Mirah
Université de Haute-Alsace, Mulhouse, France
mirah.chaki@uha.fr

Supervisé par : Professeur. IDOUMGHAR Lhassane
Directeur de l'Institut IRIMAS
Lhassane.idoumghar@uha.fr
University of Haute-Alsace (UHA)

***Abstract*—Les algorithmes génétiques, classées sous le terme de métaheuristiques appliquent une "sélection naturelle artificielle" en favorisant les solutions de haute qualité et en éliminant progressivement les solutions moins performantes. Ce processus itératif mène à la découverte de solutions optimales. Ce document reflète l'etude de l'lgorithme colonie d'abeilles artificielle (ABC).l'idée est de comparer les resultats de l'implementation en en c++ et en python.**

***Index Terms*—Algorithmes évolutaires, génétiques, Optimisation par Colonies d'abeilles, fonctions objectifs, l'algorithme d'optimisation Artificial Bee Colony (ABC), algorithmes génétiques (GA).**

I. Introduction

A. Contexte de l'optimisation

L'optimisation vise à identifier la meilleure solution parmi plusieurs alternatives. Elle est utilisée dans divers domaines comme l'ingénierie, l'économie, la finance et la logistique. En mathématiques, l'optimisation regroupe l'ensemble des méthodes visant à déterminer l'optimum d'une fonction, avec ou sans contraintes.

On distingue deux grandes catégories :

- Optimisation linéaire : la fonction objective et les contraintes sont linéaires. Elle est couramment utilisée pour l'allocation des ressources et la planification.
- Optimisation non linéaire : la fonction objective ou certaines contraintes sont non linéaires. Elle intervient fréquemment dans l'ajustement de données et la conception technique.

Définition d'un problème d'optimisation

Un problème d'optimisation combinatoire est défini par un ensemble infini d'instances. En pratique, on résout numériquement une de ces instances à l'aide d'un algorithme.

Pour chaque instance :

- Un ensemble discret de solutions S est défini.
- Un sous-ensemble X de S représente les solutions admissibles (ou réalisables).
- Une fonction de coût f (appelée aussi fonction objectif ou fitness) est associée à chaque solution s de X.

L'objectif est de trouver une solution s^* dans X qui minimise ou maximise f :

$$s^* \in X \quad \text{tel que} \quad f(s^*) \leq f(s), \forall s \in X$$

La solution s^* est appelée **solution optimale** ou **optimum global**.

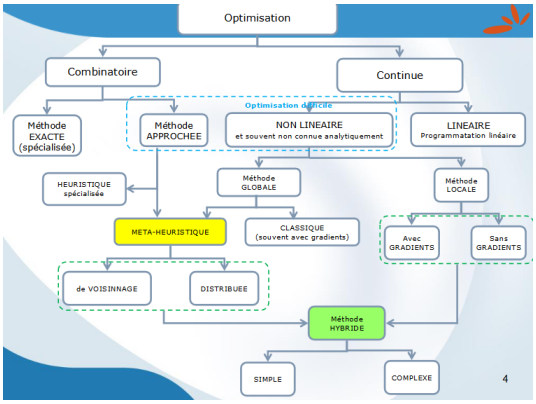

Fig. 1. Classification des méthodes d'optimisation.

Les approches métaheuristiques sont des procédés d'optimisation conçus pour traiter des problèmes complexes pour lesquels aucune méthode conventionnelle plus performante n'est connue. Elles s'inspirent fréquemment de phénomènes naturels, comme le recuit simulé, les algorithmes évolutionnaires ou encore l'optimisation par essaims de particules. Ces stratégies permettent d'identifier des solutions approximatives de haute qualité dans un délai acceptable, bien qu'elles ne garantissent pas d'atteindre l'optimum absolu.

B. Présentation des algorithmes bio-inspirés

Explication du concept des algorithmes inspirés de la nature:

Les algorithmes inspirés de la nature utilisent des mécanismes biologiques et comportementaux observés dans le monde vivant pour résoudre des problèmes complexes d'optimisation. Ces algorithmes se basent sur des concepts issus de l'évolution, de la sélection

naturelle et de la coopération entre individus pour trouver des solutions optimales dans des espaces de recherche vastes et souvent non linéaires.
Les algorithmes génétiques (GA) s'inspirent directement du processus d'évolution biologique. Dans la nature, les espèces évoluent au fil du temps en s'adaptant à leur environnement. De manière similaire, les algorithmes génétiques cherchent à optimiser des solutions en générant une population d'individus (solutions candidates) et en les faisant évoluer.

Dans la nature, ce processus est guidé par la sélection naturelle : les caractéristiques favorables à la survie et à la reproduction d'une espèce sont transmises de génération en génération, tandis que les caractéristiques moins adaptées disparaissent progressivement. Cette sélection permet d'améliorer l'espèce dans son ensemble.

Mécanismes des Algorithmes Génétiques:

- Population : Un ensemble de solutions candidates (individus), chacune représentée sous forme de chromosome (souvent une chaîne binaire).
- Codage : Les solutions sont codées génétiquement, généralement sous forme de chaînes binaires ou autres structures adaptées au problème.
- Fonction Objectif : Chaque solution est évaluée selon sa qualité à l'aide d'une fonction objectif qui attribue une valeur numérique (fitness).
- Sélection : Les individus les plus performants sont sélectionnés pour se reproduire, généralement en fonction de leur fitness.
- Croisement (Crossover) : Les individus sélectionnés échangent des parties de leur code génétique pour créer de nouveaux individus (descendants).
- Mutation : Des changements aléatoires sont introduits dans le code génétique pour maintenir la diversité et éviter les optima locaux.
- Génération Suivante : Une nouvelle génération d'individus est formée, remplaçant l'ancienne, et le processus recommence.
- Répétition : Ce processus se répète sur plusieurs générations jusqu'à ce qu'une solution optimale ou quasi-optimale soit trouvée.

Ce processus itératif d'évaluation, de sélection et de reproduction conduit à l'amélioration progressive de la population de solutions, avec pour objectif de découvrir une solution optimale ou quasi-optimale.

En plus des algorithmes génétiques, d'autres mécanismes biologiques sont utilisés dans les algorithmes inspirés de la nature, comme les algorithmes de colonies de fourmis ou les algorithmes de swarm intelligence et les Algorithmes de Colonies d'Abeilles (ABC). Ces approches tirent parti de l'intelligence collective et de la coopération entre agents pour explorer efficacement l'espace de recherche et converger vers des solutions performantes.
De notre part, nous allons parler en particulier des Algorithmes de Colonies d'Abeilles (ABC).
L'algorithme Artificial Bee Colony (ABC), a été développé par Dervis Karaboga et Basturk en 2005, s'inspire du comportement de recherche de nectar des abeilles dans la nature. Cet algorithme est basé sur l'intelligence collective des abeilles et leur capacité à coopérer et à s'adapter pour trouver les meilleures sources de nourriture. L'idée principale de l'ABC est de modéliser l'exploration et l'exploitation des abeilles en une stratégie de recherche d'optimum dans un espace de solutions.
Dans la nature, les abeilles cherchent du nectar de manière collaborative : certaines abeilles explorent de nouvelles zones (exploration), tandis que d'autres exploitent les zones déjà identifiées comme prometteuses (exploitation).
L'algorithme ABC repose sur trois catégories d'abeilles artificielles, qui jouent chacune un rôle essentiel dans la recherche de solutions optimales. Les abeilles employées(Butineuses actives) explorent l'espace de recherche et génèrent de nouvelles solutions candidates, chaque abeille étant associée à une solution spécifique qu'elle met à jour en fonction des performances observées. Les abeilles spectatrices(Butineuses éclaireuses) évaluent les solutions identifiées par les abeilles employées, sélectionnent les plus prometteuses et contribuent à leur amélioration en se basant sur les informations reçues. Enfin, les abeilles scoutes (Butineuses inactives)interviennent lorsque certaines solutions cessent de s'améliorer, recherchant alors de nouvelles solutions de manière aléatoire afin de garantir la diversité et d'éviter que l'algorithme ne se bloque dans un optimum local. Grâce à cette répartition des rôles, l'algorithme ABC est particulièrement efficace pour équilibrer l'exploration et l'exploitation,il permet aux colonies d'abeilles de maximiser leur récolte de manière optimale.

C. Objectif et motivations du rapport

L'objectif de ce projet est d'implémenter l'algorithme ABC en C++ et en Python, puis évaluer ses performances afin de comparer les résultats obtenus.

D. Organisation du document

Ce rapport est structuré de manière à fournir une compréhension claire et détaillée de l'algorithme Artificial Bee Colony (ABC), son implémentation et son évaluation expérimentale.
Dans la deuxième section, nous présentons une description détaillée de l'algorithme ABC, en expliquant son fonctionnement et ses principes fondamentaux. Cette section inclut également les schémas et diagrammes nécessaires pour illustrer les différentes étapes de l'algorithme.

La troisième section est consacrée à l'implémentation de l'algorithme en C++ et en Python. Nous y détaillons les choix d'implémentation, les particularités des langages utilisés ainsi que les différences de performance entre ces deux versions.
Ensuite, dans la quatrième section, nous analysons les résultats expérimentaux obtenus. Nous y décrivons les benchmarks utilisés, le protocole expérimental, ainsi que les résultats sous forme de graphiques et de tableaux. Cette section met en évidence les performances et les avantages de l'algorithme ABC dans différentes conditions.
Enfin, la dernière section conclut le rapport en résumant les principaux enseignements de notre étude et en proposant d'éventuelles perspectives d'amélioration ou d'autres pistes de recherche pour optimiser davantage l'algorithme ABC.
Cette organisation nous permettra de mener une étude complète et méthodique, allant de la théorie à l'expérimentation, pour mieux comprendre les forces et les limites de l'algorithme ABC.

II. Description détaillé de l'algorithme implémenté(diagrammes,etc.)

A. Modélisation et Fonctionnement de l'Algorithme Artificial Bee Colony (ABC)

L'algorithme Artificial Bee Colony (ABC) repose sur un système multi-agents, où chaque abeille est considérée comme un agent autonome. Chaque agent explore et exploite l'espace de recherche en fonction de ses expériences passées, contribuant ainsi à la recherche de solutions optimales dans un problème d'optimisation.
Dans cet algorithme, les abeilles représentent des solutions potentielles au problème à résoudre. Chaque abeille est associée à un vecteur de paramètres, ce qui correspond à une configuration particulière des variables de décision du problème. La fonction objective (ou fitness) joue un rôle clé en mesurant la qualité de chaque solution générée. Plus précisément, la fonction objective évalue la quantité de nectar (ou la qualité de la solution) à chaque position donnée dans l'espace de recherche, permettant ainsi de déterminer la performance des abeilles.

Les trois types d'abeilles et leur rôle dans l'algorithme ABC

Les abeilles employées : Ces abeilles représentent les agents actifs qui explorent une solution particulière. Chaque abeille employée est associée à une solution en cours d'exploration. Leur nombre est égal au nombre de solutions en exploration à un instant donné. Les abeilles employées mettent à jour leur état en fonction des performances observées, en exploitant les solutions et en tentant de les améliorer. Elles jouent un rôle crucial dans l'exploitation de l'espace de recherche.

Les abeilles observatrices : Ces abeilles sélectionnent les meilleures solutions en fonction de leur fitness (qualité). Une abeille spectatrice évalue les solutions générées par les abeilles employées et choisit celles qui sont les plus prometteuses pour une exploitation ultérieure. Elles sont donc essentielles pour le processus de sélection des solutions optimales et leur amélioration continue.

Les abeilles éclaireuses : Les abeilles éclaireuses génèrent de nouvelles solutions aléatoires dans l'espace de recherche lorsque certaines solutions stagnent ou ne donnent plus de résultats intéressants. Elles interviennent pour garantir la diversité dans l'espace de recherche et éviter que l'algorithme ne se bloque dans un optimum local. Leur rôle est donc primordial pour l'exploration du problème, en permettant d'élargir les possibilités de solutions lorsque les abeilles employées et spectatrices n'ont pas amélioré les solutions existantes.

Les abeilles communiquent entre elles à travers des danses, un mécanisme de communication spécifique. Ces danses indiquent aux autres abeilles l'emplacement des sources de nourriture (solutions potentielles) et leur distance par rapport à la ruche. Par exemple :

- Danse circulaire : indique une source de nectar proche (moins de 50 m).
- Danse en huit : signale une source de nectar plus éloignée (plus de 50 m).

Grâce à la combinaison de ces trois types d'abeilles et de leur capacité à communiquer, l'algorithme ABC équilibre efficacement l'exploration (découverte de nouvelles solutions) et l'exploitation (amélioration des solutions existantes), ce qui le rend particulièrement performant pour résoudre des problèmes d'optimisation complexes.
Pour expliquer le principe de cet algorithme, voici un schéma bloc présentant ses différentes étapes :

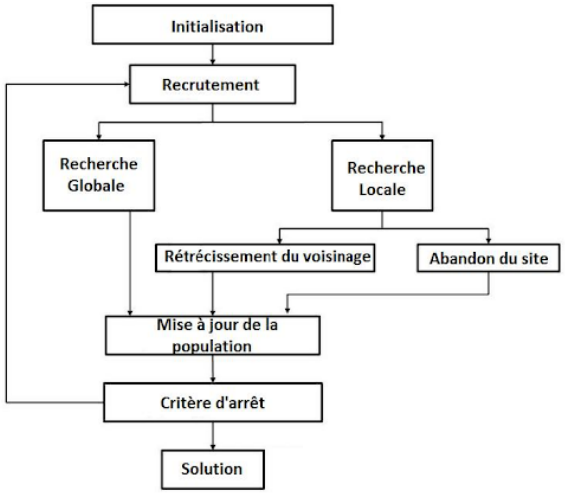

Fig. 2. un schéma bloc présentant les différentes étapes de l'algorithme.

1. Initialisation

- Génération aléatoire d'un ensemble de solutions candidates représentant des sources de nourriture.
- Attribution des abeilles employées aux solutions.
- **Paramètres :**
  - Population totale = abeilles employées + abeilles 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 atteint.

2. Recrutement (Phase des abeilles employées)

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

Chaque abeille employée génère une nouvelle solution en modifiant légèrement sa solution actuelle selon :

$$x_{ij}^{t+1} = x_{ij}^{t} + \phi_{ij} \cdot (x_{kj}^{t} - x_{ij}^{t}) \tag{1}$$

où :

- x_{ij}^{t} est la position actuelle.
- x_{kj}^{t} est la position d'une autre solution sélectionnée aléatoirement.
- ϕ_{ij} est un facteur aléatoire dans [-1,1].

Si la nouvelle solution est meilleure, elle remplace l'ancienne. Sinon, un compteur d'échecs est incrémenté.

3. Recherche locale (Phase des abeilles observatrices)

Objectif : Sélectionner les solutions les plus prometteuses pour une exploitation plus poussée.

Les abeilles spectatrices choisissent les meilleures solutions en fonction de leur fitness, avec une probabilité proportionnelle à la qualité de chaque solution :

$$P_i = \frac{fitness_i}{\sum fitness} \tag{2}$$

Elles génèrent ensuite de nouvelles solutions en appliquant la même équation que les employées. Si la nouvelle solution est meilleure, elle remplace l'ancienne.

4. Rétrécissement du voisinage Si aucune amélioration n'est observée après plusieurs tentatives, la taille du voisinage est réduite progressivement :

$$a(t+1) = 0.8 \times a(t) \tag{3}$$

Cela permet de concentrer la recherche sur une zone plus restreinte et d'éviter de gaspiller du temps dans des régions non prometteuses.

5. Abandon du site (Phase des abeilles éclaireuses) Si une solution stagne trop longtemps (c'est-à-dire que son compteur d'échecs dépasse un seuil fixé), elle est abandonnée. Une nouvelle solution est générée aléatoirement dans l'espace de recherche pour remplacer la solution abandonnée :

$$x_{new} = lb + rand(0,1) \cdot (ub - lb) \tag{4}$$

Objectif : Maintenir la diversité des solutions et éviter de rester coincé dans un optimum local.

6. Recherche globale Certaines abeilles éclaireuses sont envoyées au hasard dans l'espace de recherche pour explorer de nouvelles zones. Cela permet d'éviter un piégeage prématuré et d'accélérer la convergence vers une solution optimale.

7. Mise à jour et critère d'arrêt

- La meilleure solution trouvée est mémorisée et mise à jour.
- L'algorithme s'arrête lorsque :
  - Le nombre maximal d'itérations est atteint.
  - Une solution satisfaisante en termes de fitness est trouvée.

B. Algorithme en python

Dans cette section, nous allons examiner en détail chaque composant du code Python.

1. Importation des bibliothèques

Le programme commence par importer plusieurs bibliothèques essentielles :

```
import numpy as np
import math
from mealpy import FloatVar
from scipy import stats
from tabulate import tabulate
```

Explication des bibliothèques :

- **numpy** : utilisé pour les opérations mathématiques et la gestion des tableaux.
- math : fournit des fonctions mathématiques comme `cos` et `pi`.
- mealpy.FloatVar : définit les limites des variables continues utilisées dans les problèmes d'optimisation.
- scipy.stats : permet d'effectuer le test statistique de Wilcoxon, utile pour comparer les performances de Python et C++.
- tabulate : permet d'afficher les résultats sous forme de tableaux.

2. Définition de la classe OriginalABC

L'algorithme ABC est encapsulé dans une classe appelée `OriginalABC`, qui hérite de la classe `Optimizer` du module `mealpy`.

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

Explication des paramètres du constructeur :

```
def __init__(self, epoch: int = 10000,
pop_size: int = 100, n_limits:
int = 25, **kwargs: object) -> None:
```

- epoch : Nombre maximal d'itérations (par défaut 10 000).
- pop_size : Taille de la population (nombre d'abeilles employées et spectatrices combinées).
- n_limits : Limite d'essais avant abandon d'une source de nourriture par une abeille éclaireuse.

L'initialisation valide les paramètres et configure l'optimiseur :

```
self.epoch = self.validator.check_int
("epoch", epoch, [1, 100000])
self.pop_size = self.validator.check_int
("pop_size", pop_size, [5, 10000])
self.n_limits = self.validator.check_int
("n_limits", n_limits, [1, 1000])
self.trials = np.zeros(self.pop_size)
```

trials est un tableau de suivi des échecs pour chaque solution.

- trials :est un tableau de suivi des échecs pour chaque solution

3. Phase des abeilles employées

Chaque abeille employée explore une nouvelle solution basée sur la mise à jour suivante :

$$x_{ij} = x_{ij} + \varphi_{ij} \times (x_{tj} - x_{ij})$$

```
for idx in range(0, self.pop_size):
    rdx = self.generator.choice
    (list(set(range(0, self.pop_size)) -
    {idx}))
    phi = self.generator.
    uniform(low=-1, high=1, size=
    self.problem.n_dims)
    pos_new = self.pop[idx].
    solution + phi *
    (self.pop[rdx].solution -
    self.pop[idx].solution)
    pos_new = self.correct_solution(pos_new)
    agent = self.generate_agent(pos_new)

    if self.compare_target(agent.target,
    self.pop[idx].target,
    self.problem.minmax):
        self.pop[idx] = agent
        self.trials[idx] = 0
    else:
        self.trials[idx] += 1
```

Explication :

- Une autre abeille est choisie aléatoirement (`rdx`) pour générer une nouvelle solution.
- `phi` est un facteur aléatoire entre $[-1, 1]$.
- Une nouvelle solution est calculée et corrigée avec `correct_solution`.
- Si la nouvelle solution est meilleure, elle est adoptée.
- Sinon, le compteur d'échecs (`trials[idx]`) est incrémenté.

4. Phase des abeilles spectatrices

Les abeilles spectatrices sélectionnent les meilleures solutions en fonction d'une probabilité proportionnelle à leur fitness :

$$P_i = \frac{fitness_i}{\sum fitness}$$

Implémentation :

```
employed_fits = np.array([agent.target.fitness
for agent in self.pop])

for idx in range(0, self.pop_size):
    selected_bee = self.
    get_index_roulette_wheel_selection
    (employed_fits)
    rdx = self.generator.choice
    (list(set(range(0, self.pop_size))
    - {idx, selected_bee}))
    phi = self.generator.
    uniform(low=-1, high=1,
    size=self.problem.n_dims)
    pos_new = self.pop[selected_bee].
    solution + phi *
    (self.pop[rdx].solution -
    self.pop[selected_bee].solution)
    pos_new = self.correct_solution(pos_new)
    agent = self.generate_agent(pos_new)

    if self.compare_target
    (agent.target, self.pop[selected_bee].
    target,elf.problem.minmax):
        self.pop[selected_bee] = agent
        self.trials[selected_bee] = 0
    else:
        self.trials[selected_bee] += 1
```

Explication :

- La *roulette wheel selection* est utilisée pour choisir les meilleures solutions.
- La nouvelle solution est générée comme dans la phase des abeilles employées.
- Si la solution est meilleure, elle remplace l'ancienne.
- Sinon, le compteur d'échecs est augmenté.

5. Phase des abeilles éclaireuses

Si une solution stagne au-delà de `n_limits`, elle est remplacée par une nouvelle solution aléatoire.

Implémentation :

```
abandoned = np.where(self.trials >=
self.n_limits)[0]
```

```
for idx in abandoned:
    self.pop[idx] = self.generate_agent()
    self.trials[idx] = 0
```

Explication :

- Les indices des solutions stagnantes sont récupérés avec `np.where(self.trials >= self.n_limits)`.
- Ces solutions sont remplacées par une solution générée aléatoirement.
- Le compteur d'échecs est remis à zéro.

6. Définition des fonctions de benchmark

Les performances de l'algorithme sont évaluées sur trois fonctions d'optimisation classiques :

Rosenbrock

```
def rosenbrock(solution):
    return np.sum(100 * (solution[1:] -
    solution[:-1]**2)**2 +
    (solution[:-1] - 1)**2)
```

Forme une vallée étroite, difficile à optimiser.

Rastrigin

```
def rastrigin(solution):
    return 10 * len(solution) +
    np.sum(solution**2 - 10 *
    np.cos(2 * np.pi * solution))
```

Beaucoup de minima locaux, testant la robustesse de l'algorithme.

Ackley

```
def ackley(solution):
    d = len(solution)
    return -20 * np.exp(-0.2 * np.sqrt
    (np.sum(solution ** 2) / d))
    - np.exp(np.sum(np.cos
    (2 * np.pi * solution)) / d)
    + 20 + np.exp(1)
```

Topographie complexe, nécessitant une exploration approfondie.

7. Comparaison Python vs C++ avec Wilcoxon

Le test statistique de Wilcoxon est utilisé pour comparer les performances entre Python et C++.

Implémentation :

```
statistic, p_value = stats.wilcoxon
(python_results, cpp_results)

if p_value < 0.05:
    print("Différence significative
    entre Python et C++")
else:
    print("Pas de différence significative")
```

Explication :

- Si $p\_value < 0.05$, il y a une différence significative.
- Sinon, les résultats sont statistiquement similaires.

C. Algorithme en C++

L'implémentation de l'algorithme ABC est divisée en trois fichiers principaux :

- **main.cpp** : Ce fichier contient le programme principal qui exécute l'algorithme sur différentes fonctions d'optimisation.
- **OriginalABC.h** : Déclare les classes et prototypes des méthodes utilisées dans l'algorithme.
- **OriginalABC.cpp** : Implémente les méthodes déclarées dans `OriginalABC.h`, définissant ainsi le fonctionnement des abeilles dans l'algorithme ABC.

1) Bibliothèques utilisées: L'implémentation en C++ utilise plusieurs bibliothèques essentielles :

- `<algorithm>` : Contient des fonctions utilitaires comme `std::max` pour la manipulation des valeurs numériques.
- `<cmath>` : Fournit des outils mathématiques essentiels pour les calculs complexes.
- `<ctime>` : Permet de générer des nombres aléatoires en utilisant l'horloge système.
- `<limits>` : Définit les valeurs minimales et maximales des types numériques.
- `<vector>` : Utilisé pour stocker les solutions et les paramètres de l'algorithme.
- `<iostream>` : Permet l'affichage des résultats et l'interaction avec l'utilisateur.
- `<fstream>` : Utilisé pour lire et écrire des fichiers contenant les résultats.
- `<numeric>` : Permet des opérations avancées sur des ensembles de valeurs numériques.
- `<functional>` : Permet de manipuler les fonctions passées en argument (par exemple, la fonction objectif).
- `<random>` : Fournit des générateurs de nombres aléatoires plus sophistiqués que `<ctime>`.
- `<utility>` : Fournit des fonctionnalités d'aide pour la gestion des paires de valeurs.

2) Classes et méthodes principales: **Classe Agent**

La classe `Agent` représente une abeille, c'est-à-dire une solution candidate avec une valeur de fitness associée.

```
class Agent {
public:
    Agent(int dim) : solution(dim, 0.0),
        fitness(std::numeric_limits<double>::
        max()) {}
    std::vector<double> solution;
    double fitness;
```

```
};
```

Chaque agent possède un vecteur de solutions initialisé à zéro et une valeur de fitness définie au maximum possible pour favoriser la minimisation.

Classe OriginalABC

Cette classe gère l'algorithme ABC et ses différentes phases d'exécution.

Constructeur de la classe

Le constructeur de la classe `OriginalABC` initialise l'algorithme avec les paramètres suivants :

- **epoch** : Nombre maximal d'itérations pour la convergence.
- **pop_size** : Taille totale de la population d'abeilles.
- **n_limits** : Nombre maximal de tentatives avant qu'une source soit abandonnée.

De plus, le constructeur initialise un générateur de nombres aléatoires pour garantir une exécution reproductible.

```
OriginalABC::OriginalABC(int epoch,
int pop_size, int n_limits)
: epoch(epoch), pop_size(pop_size),
n_limits(n_limits), problem_dim(0) {
std::srand(static_cast(std::time(nullptr)));
}
```

Méthode `generate_new_solution`

Cette méthode génère une nouvelle solution en ajustant chaque dimension de la solution actuelle en fonction d'un coefficient aléatoire compris entre -1 et 1.

```
std::vector OriginalABC::generate_
new_solution(
const std::vector& solution,
const std::vector& neighbor_solution) {
std::vector new_solution(solution.size());
for (size_t i = 0; i < solution.size(); ++i) {
double phi = uniform(-1.0, 1.0);
new_solution[i] = solution[i] + phi *
(solution[i] - neighbor_solution[i]);
}
return new_solution;
}
```

Méthodes auxiliaires

Plusieurs fonctions auxiliaires sont utilisées pour garantir la stabilité et l'efficacité de l'algorithme :

- **correct_solution** : Vérifie que la solution générée reste dans les bornes définies.
- **uniform** : Génère un nombre aléatoire dans un intervalle donné .
- **compare_target** : Compare deux valeurs de fitness pour déterminer la meilleure solution.
- **get_best_fitness** : Retourne la meilleure valeur de fitness parmi la population.

Méthode `initialize_variables`

Cette méthode initialise les paramètres essentiels :

- Les bornes des variables définissant l'espace de recherche.
- La population d'abeilles avec des solutions générées aléatoirement.
- Un compteur de tentatives pour chaque abeille initialisé à zéro.

Elle utilise la fonction `uniform` pour générer les solutions initiales aléatoires.

```
void OriginalABC::initialize_variables(
const std::vector<std::pair<double, double>>
& bounds) {
this->bounds = bounds;
this->problem_dim = bounds.size();
population = std::vector(pop_size,
Agent(problem_dim));
trials = std::vector(pop_size, 0);
}
```

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 par les abeilles employées.**
3) **Gestion des solutions abandonnées par les éclaireuses.**
4) **Amélioration progressive des solutions jusqu'à convergence.**

Cette méthode est appelée pour optimiser une fonction objectif donnée.

3) Phases de l'algorithme ABC: **1. Phase des abeilles employées :**

- Les abeilles employées explorent l'espace des solutions en générant de nouvelles solutions proches de celles existantes.
- Chaque abeille sélectionne une solution voisine et génère une nouvelle proposition en fonction de celle-ci.
- La nouvelle solution est validée avec `correct_solution` pour rester dans les limites autorisées.
- Si la nouvelle solution est meilleure, elle remplace l'ancienne ; sinon, le compteur d'échecs `trials` est incrémenté.

2. Phase des abeilles observatrices :

- Les abeilles observatrices sélectionnent les solutions les plus prometteuses en fonction de leur qualité.
- Bien que cette phase ne soit pas explicitement distincte dans le code, elle est intégrée dans la phase d'exploration.

3. Phase des abeilles éclaireuses :

- Lorsqu'une solution stagne au-delà du seuil `n_limits`, elle est remplacée par une nouvelle solution aléatoire.

- Ce processus permet d'explorer constamment de nouvelles zones de l'espace de recherche.

4) Méthodes principales: **Initialisation des variables**

```
void OriginalABC::initialize_variables
    const std::vector<std::pair<double,
    double>>
    & bounds) {
    this->bounds = bounds;
    this->problem_dim = bounds.size();
    population = std::vector<Agent>(
    pop_size,
        Agent(problem_dim));
    trials = std::vector<int>(pop_size, 0);
}
```

Cette méthode initialise les abeilles avec des solutions aléatoires dans les bornes spécifiées.

Génération de nouvelles solutions

```
std::vector<double> OriginalABC::genera
te_new_so
lution(
 const std::vector<double>& solution,
 const std::vector<double>& neighbor_so
 lution) {
std::vector<double> new_solution(solu
tion.size());
for (size_t i = 0; i < solution.size();
++i) {
        double phi = uniform(-1.0, 1.0);
        new_solution[i] = solution[i]
        + phi *
        (solution[i] - neighbor_solution[i
    }
    return new_solution;
}
```

Les abeilles employées génèrent de nouvelles solutions en ajustant les valeurs des dimensions en fonction d'un voisin choisi aléatoirement.

Tableau comparatif entre l'implémentation en C++ et en Python.

Aspect	**C++**	**Python**
Extensibilité	Moins flexible, difficile à étendre rapidement.	Très extensible, facile à modifier et à maintenir.
Lisibilité	Code plus complexe	Plus simple et clair
Gestion de la mémoire	Gestion manuelle via `std::vector`.	Automatique grâce au garbage collector.
Structure du code	Trois fichiers : `main.cpp` `OriginalABC.h` (déclarations) et `OriginalABC.cpp` (implémentations).	Un seul fichier `OriginalABC.py` centralise tout.

TABLE I
COMPARAISON ENTRE C++ ET PYTHON

III. RÉSULTATS EXPÉRIMENTAUX

A. Description des Benchmarks utilisés

Dans ce projet, trois fonctions de benchmark classiques en optimisation ont été utilisées :

- **Rosenbrock** : Également connue sous le nom de "Vallée de Rosenbrock", cette fonction est non convexe et présente une vallée étroite menant au minimum global. Son optimisation est difficile en raison de sa topographie, qui exige une exploration fine pour converger efficacement vers la solution optimale.
- **Rastrigin** : Cette fonction multimodale est caractérisée par un grand nombre de minima locaux, ce qui complique la tâche des algorithmes d'optimisation. Elle permet d'évaluer la capacité d'un algorithme à éviter les optima locaux et à trouver le minimum global.
- **Ackley** : Avec ses composantes exponentielles et trigonométriques, cette fonction crée un paysage complexe parsemé de nombreuses oscillations locales. Elle est particulièrement utilisée pour tester la robustesse des algorithmes face aux pièges des minima locaux et leur capacité à converger efficacement.

Les figures 3, 4 et présentent les graphes des fonctions étudiées, illustrant leurs structures et la difficulté d'optimisation qu'elles posent.

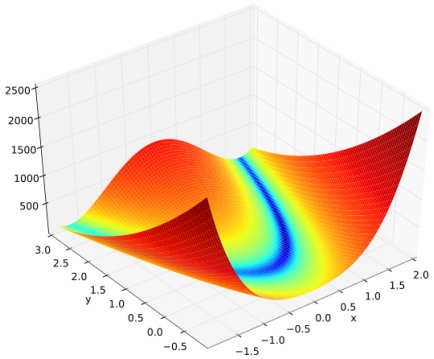

Fig. 3. Représentation de la fonction de Rosenbrock

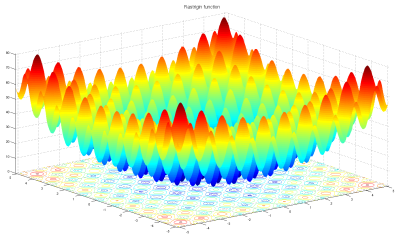

Fig. 4. Représentation de la fonction de Rastrigin

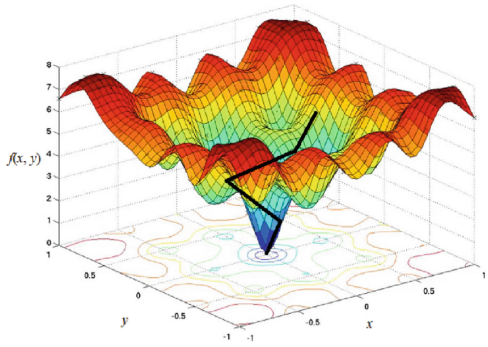

Fig. 5. Représentation de la fonction d'Ackley

B. *Protocole expérimental*

L'expérimentation vise à évaluer la performance de l'algorithme *Artificial Bee Colony* (ABC) sur trois fonctions de benchmark : Rosenbrock, Rastrigin et Ackley. Pour garantir la robustesse des résultats, plusieurs exécutions ont été réalisées en suivant le protocole suivant :

Paramètres d'optimisation : L'algorithme ABC a été configuré avec un nombre d'itérations (*epoch*) fixé à 5000 et une population de 30 individus. Le paramètre *n_limits*, qui représente le nombre maximal d'essais avant l'abandon d'une solution, a été fixer à 100 afin de laisser le temps à l'algorithme de tester sa valeur.

Exécutions répétées : Chaque test a été réalisé 10 fois pour chaque fonction afin d'obtenir une évaluation statistiquement significative des performances. La moyenne et l'écart-type de la fitness est faite sur ces 10 résultats obtenus. La moyenne est l'écart-type permettent de s'assurer d'avoir des valeures cohérentes, car selon l'inititalisation des points aléatoires, on pourrait obtenir des résultats différents après chaque initialisation.

Évaluation des performances : Les résultats ont été analysés en calculant la moyenne et l'écart-type des valeurs de fitness obtenues. De plus, une comparaison entre les implémentations en Python et C++ a été effectuée pour évaluer les différences de performance. Il a été constaté que le code en C++ permet d'obtenir des résultats plus performants. Cette différence de performance pourrait être attribuée à plusieurs facteurs, notamment l'optimisation du langage C++ qui permet une exécution plus rapide en raison de sa gestion plus fine des ressources systèmes, par rapport à Python qui est un langage interprété. Les différences entre les résultats obtenus avec Python et C++ peuvent être dues aux générateurs de nombres aléatoires utilisés. En Python, le module `random` utilise un générateur de type Mersenne Twister, qui produit des séquences déterministes. En C++, la bibliothèque standard C++11 et versions ultérieures utilisent `std::mt19937` (basé sur Mersenne Twister) ou `std::random_device`, offrant des résultats plus proches de l'aléatoire véritable. Cependant, ces générateurs sont également déterministes dans certaines conditions, ce qui peut expliquer les variations observées entre les deux langages. Une photo du tableau comparatif entre les résultats trouvés en Python et ceux en C++ est inclus pour illustrer cette analyse. On remarque que plus la population augmente, plus le pourcentage de différence entre les deux langages diminue. Cela est logique, car effectuer des tests sur de petites populations n'est pas toujours représentatif et peut entraîner des variations plus importantes en raison de la nature aléatoire des résultats. textbfÉvaluation des performances : Les résultats ont été analysés en calculant la moyenne et l'écart-type des valeurs de fitness obtenues après optimisation. De plus, une comparaison entre les implémentations en Python et C++ a été effectuée pour évaluer les différences de performance. Il a été constaté que le code en C++ permet d'obtenir des résultats plus performants. Cette différence de performance pourrait être attribuée à plusieurs facteurs, notamment l'optimisation du langage C++ qui permet une exécution plus rapide en raison de sa gestion plus fine des ressources systèmes, par rapport à Python qui est un langage interprété.

Dimension	Function	Moyenne Python	Écart-type Python	Moyenne C++	Écart-type C++	Diff (%)	Better Implementation
30	Rosenbrock	6970	433.737	1876.71	397.63	73.07%	C++
30	Rastrigin	449	19.0889	331.586	18.9776	26.15%	C++
30	Ackley	20	0.1421	16.6436	1.1323	16.78%	C++
50	Rosenbrock	14489	1108.36	7830.03	1173.89	45.96%	C++
50	Rastrigin	768	42.9799	645.013	24.2568	16.01%	C++
50	Ackley	20	0.0451	20.086	0.2743	-0.43%	Python

Fig. 6. Tableau comparatif entre les résultats obtenus en Python et en C++.

100	Rosenbrock	36454	2840.14	30769	1823.47	15.59%	C++
100	Rastrigin	1650	42.9979	1509.63	31.4353	8.51%	C++
100	Ackley	21	0.063	20.8629	0.0674	0.65%	C++

Fig. 7. Tableau comparatif entre les résultats obtenus en Python et en C++.

Test statistique : Afin de vérifier la significativité des différences entre les implémentations, un test de Wilcoxon a été réalisé sur les résultats obtenus avec les différentes dimensions du problème (30, 50 et 100). Ce test permet d'identifier si une implémentation surpasse significativement l'autre. Le test de Wilcoxon a été choisi pour sa capacité à comparer des paires de résultats

sur des échantillons appariés, tout en étant non paramétrique, robuste aux distributions non normales et aux valeurs extrêmes, ce qui est crucial dans le contexte de l'analyse des performances des deux implémentations.

C. Resultats:graphiques, tables

Dans cette section, nous présentons les résultats obtenus à travers des représentations graphiques et des tableaux comparatifs afin d'évaluer la performance des différentes approches.

D. Tableaux de Résultats

Les graphiques obtenus illustrent la convergence de l'algorithme sur les fonctions de benchmark Rosenbrock, Rastrigin et Ackley. Les courbes montrent l'évolution de la meilleure fitness au fil des itérations, mettant en évidence la stabilité et la vitesse de convergence de l'algorithme. Une comparaison entre les implémentations en Python et en C++ révèle des différences notables en termes de performance. Les résultats numériques indiquent que l'implémentation C++ atteint une meilleure précision et une convergence plus rapide, notamment pour les fonctions de Rosenbrock et Rastrigin. En revanche, pour la fonction d'Ackley, les deux implémentations présentent des performances similaires. Les tests statistiques de Wilcoxon confirment ces observations en mettant en évidence une différence significative pour les fonctions où C++ est plus performant, tandis que pour Ackley, aucune différence notable n'est détectée.

IV. Conclusion

Cette étude a comparé les performances de l'algorithme *Artificial Bee Colony* (ABC) implémenté en `C++` et `Python` sur trois fonctions objectives classiques : Rosenbrock, Rastrigin et Ackley. L'analyse a mis en évidence les avantages et les limitations de chaque langage en termes de **rapidité d'exécution**, **précision** et **stabilité**.

Les résultats expérimentaux montrent que `C++` offre une meilleure efficacité computationnelle avec une convergence plus rapide et une gestion optimisée des ressources, notamment sur les fonctions **Rosenbrock** et **Rastrigin**. En revanche, `Python` présente une approche plus accessible et flexible, bien que son exécution soit plus lente. Pour la fonction **Ackley**, les performances des deux langages sont comparables.

Le choix entre `Python` et `C++` dépend des exigences spécifiques du projet :

- **Python** est mieux adapté aux *analyses exploratoires* et aux *tests rapides*.
- **C++** est préférable pour les *calculs intensifs* et les *applications nécessitant une réponse rapide*.

En conclusion, cette étude ouvre des perspectives d'optimisation de l'algorithme ABC, notamment en explorant des ajustements de paramètres et en testant d'autres variantes. Une meilleure gestion des mécanismes d'exploration et d'exploitation pourrait encore améliorer ses performances sur des problèmes complexes.

References

[1] B. Mascret, "Bees and AI", http://bruno.mascret.fr/ia/bees/index.html, 2025.

[2] C. Celien, "IDOUMGHAR Lhassane", https://e-partage.uha.fr/service/home/ /?auth=coloc=frid=8980part=2, 2025.

[3] StudySmarter, "Optimisation en Mathématiques Appliquées", https://www.studysmarter.fr/resumes/mathematiques/mathematiques-appliquees/optimisation/, Consulté le 04 février 2025.

[4] M. Hamadi, "Optimisation par Algorithmes de Colonies d'Abeilles", https://repository.enp.edu.dz/jspui/bitstream/123456789/7136/1/HAMADI.Mohamed. 2025.

[5] B. Mascret, "IA et Algorithmes Inspirés des Abeilles", http://bruno.mascret.fr/ia/bees/index.html, 2025.