République Algérienne Démocratique et Populaire
Ministere de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Mohamed Khider - BISKRA
Faculté des Sciences Exactes, des Sciences Naturelles et de la Vie
Département d'Informatique

N d'ordre :GLSD4/M2/2020

Mémoire

Présenté pour l'obtention du diplôme de Master académique en

Informatique

Option : **Génie logiciel et systèmes distribués**

Parallélisation de l'Algorithme à Èvolution Différentielle d'Optimisation Mono-objectif

---

par :
SAHRAOUI Selma

Soutenu le 1/09/2020, devant le jury :

xxxxxx	xxx	Président
xxxxxx	xxx	Président
xxxxxx	xxx	Examinateur

Année universitaire : 2019/2020

Remerciements

Je remercie Allah le tout puissant de de m'avoir donné la volonté, le courage, la santé et la patience durant tous ces longues années d'études.

Je tiens à exprimer toute ma reconnaissance à mon encadreur, **Pr. KAHLOUL Laid**. Je le remercie de m'avoir encadré, orienté, aidé et conseillé.

Je remercie les professeurs de l'université de Biskra, qui m'ont fourni les outils nécessaires à la réussite de mes études universitaires.
Je tiens à remercier spécialement le chef de département d'informatique : **Pr. BABAHNINI Mohamed Chawki**.

Je désire aussi remercier les membres de jury : **xxxx** et **xxxx** pour la lecture et l'évaluation de ma thèse.

je remercie ma collègue, **Asma BESBAS** qui a toujours été là pour moi. Son soutien inconditionnel et son encouragement ont été d'une grande aide.

Enfin, tous mes remerciements et mon gratitude vont à mes très chers parents, mon mari et à ma précieuse famille pour leurs soutien et encouragement, et toute personne ayant contribué à l'élaboration de ce travail, par un conseil ou même par un sourire.

Résumé

L'optimisation **difficile** représente une classe de problèmes dont la résolution ne peut pas être obtenue par une méthode exacte en un temps raisonnable. Trouver une solution en un temps raisonnable oblige à trouver un compromis quant à son exactitude. Les algorithmes à évolution différentielle d'optimisation mono-objectifs (DE) sont des métaheuristiques de la famille des algorithmes évolutionnaire, destinées à la résolution des problèmes d'optimisation difficiles. Ces algorithmes visent à rechercher une valeur approchée à la solution optimale dans un temps raisonnable pour un problème d'optimisation donné. L'algorithme EDEV [23] est l'un des algorithmes à évolution différentielle les plus robustes proposé pour traiter des problèmes d'optimisation complexe. Il est composé de trois variantes de DE très efficaces : JADE [24], CoDE [22] et EPSDE [17].

Les DEs sont simples et performants, mais ils sont coûteux en temps de calcul. Minimiser le temps d'exécution, c'est l'une des raisons pour laquelle les algorithmes à évolution différentielle parallèles (PDE) sont utilisés. PDE permettent d'obtenir des résultats efficaces avec un temps d'exécution très inférieur à celui de leurs homologues séquentiels. Pour cela, nous proposons dans ce mémoire donc une version parallèle d'EDEV (PEDEV). Les résultats expérimentaux montrent que PEDEV proposé améliore EDEV en minimisant le temps de calcul.

Mots clés : *Optimisation, métaheuristiques, algorithmes évolutionnaires, évolution différentielle, PEA, PEDEV, temps d'exécution.*

Abstract

Hard optimization stands for a class of problems which solutions cannot be found by an exact method, with a reasonable time. Finding the solution in an acceptable time requires compromises about its accuracy. Mono-Objective Optimization Differential Evolution algorithms (DE) are metaheuristics from the evolutionary algorithms from the family of evolutionary algorithms intended to solving hard optimization problems. These algorithms aim to find an approximate value to the optimal solution in a reasonable time for a given optimization problem. EDEV algorithm is one of the most robust evolutionary algorithms proposed to deal with complex optimization problems. It is composed of three very efficient DE variants : JADE [24], CoDE [22] and EPSDE [17].

However, even if DEs are simple and efficient, but they are characterized by long execution time. One of the main reasons for adapting parallel evolutionary algorithms (PEAs) is to obtain efficient results with an execution time much lower than the one of their sequential. Thus, we propose in this thesis a parallel version of EDEV(i.e., PEDEV). As experimental results, the proposed PDMOEA enhances DMOEA in terms of minimization of computational time.

Key words : *Optimization, metaheuristics, evolutionary algorithms, differential evolution, PEA, PEDEV, execution time.*

Table des matières

Table des figures

List of Algorithms

Introduction Générale

Introduction Générale

Introduction

De nos jours, les chercheurs et les ingénieurs sont souvent face à des problèmes complexes. L'optimisation est un outil essentiel dans le processus de résolution de ces problèmes. Elle permet de trouver une configuration idéale, d'obtenir un gain d'effort, de temps, d'argent, d'énergie, ou encore de satisfaction. Pour certains problèmes, la solution optimale ne peut pas être obtenue par des méthodes exactes, à cause de leur complexité et du temps d'exécution prohibitif qu'ils nécessitent. C'est le cas des problèmes dits d'optimisation difficile. Dans le processus d'optimisation de ces problèmes, la méthode la plus appropriée sera généralement une métaheuristique.

Les métaheuristiques à base d'une population de solutions (P-métaheuristiques) sont des algorithmes d'optimisation itératifs visant à résoudre des problèmes d'optimisation difficile. Ils permettent d'obtenir une valeur approchée à la solution optimale dans un temps raisonnable. Les P-métaheuristiques améliorent au fur et à mesure des itérations un ensemble de solutions (population) en appliquant un processus de recherche qui comporte trois étapes. Tout d'abord, la génération d'une population initiale. Ensuite, la reproduction des individus (solutions) de la population pour obtenir une nouvelle population de solutions. Enfin, cette nouvelle population est intégrée à la population courante à l'aide de certaines procédures de sélection. Le processus de recherche est arrêté lorsqu'une condition donnée est satisfaite (critère d'arrêt). Les algorithmes évolutionnaires appartiennent à cette classe de métaheuristiques.

Les algorithmes évolutionnaires *(AE)* sont des P-métaheuristiques inspirées de la théorie Darwinienne de l'évolution des espèces biologiques[20]. Ils sont basés sur l'évolution d'une population de solutions pour trouver la solution optimale ou la solution quasi optimale parmi les membres de la population. À chaque itération (génération), la population évolue grâce à un ensemble d'opérations : sélection, croisement et mutation.

L'algorithme à évolution différentielle *(DE)* est parmi les *AEs* les plus simples et les plus performants, développée par Storn et Price[19]. Il a été largement appliqué dans de nombreux domaines scientifiques et techniques : ordonnancement de tâches d'un satellite, traitement d'images, optimisation de processus chimiques, entraînement des réseaux de neurones, ...etc. En *DE*, un individu est codé sous forme d'un vecteur appelé vecteur cible. A chaque génération, les trois opérations (mutation, croisement et sélection) sont exécutées sur chaque vecteur cible. Après la mutation, un vecteur mutant est obtenu. Ensuite, un vecteur d'essai est produit en réalisant un croisement entre le vecteur mutant et son vecteur cible associé. En fin, La sélection garantit la survie du meilleur entre le vecteur cible et le vecteur d'essai. Le succès du *DE* dépend de manière cruciale du choix approprié des stratégies de génération des vecteurs d'essai et de leurs valeurs de paramètres de contrôle associées.

L'algorithme à évolution différentielle *EDEV* [23] est l'un des algorithmes les plus robustes proposé pour traiter des problèmes d'optimisation complexe. Il est composé de trois variantes de *DE* très efficaces : *JADE* [24], *CoDE* [22] et *EPSDE* [17]. Bien qu'*EDEV* fonctionne pour donner d'excellents résultats, être un *AE* signifie qu'il est certainement coûteux en temps de calcul. Un problème qui ne peut pas être ignoré par l'utilisateur qui cherche toujours les meilleurs résultats dans un temps réduit.

Dans cette thèse, une version parallèle d'EDEV est proposée pour obtenir des résultats meilleurs avec un temps d'exécution bien inférieur à celui de la version séquentielle.

Cette thèse débute par une introduction qui présente la problématique et le but du projet. Le mémoire est composé de deux parties, la première partie montre l'aspect théorique, la seconde partie montre notre contribution. Partie I est divisé en deux chapitres, chapitre 1 décrit l'algorithme à évolution différentielle utilisé dans ce travail (EDEV), les trois variantes de DE (JADE, CoDE, EPSDE) qui le composent et définit certains termes associés. chapitre 2 montre les concepts de base du calcul parallèle organisés en trois sections principales : les concepts liés à la parallélisation, le calcul parallèle et les modèles de programmation parallèle. Partie II se concentre sur l'implémentation et la parallélisation d'EDEV avec la démonstration de son efficacité. Elle est composé de 2 chapitres (c'est-à-dire chapitre 3 et chapitre 4). chapitre 3 décrit l'analyse et la conception de notre application en deux niveaux principaux (conception d'EDEV et conception de PEDEV) et chapitre 4 illustre l'implémentation des deux algorithmes (c-à-d. EDEV, PEDEV) et l'implémentation de l'ensemble de l'application avec la démonstration de l'efficacité d'EDEV.

La thèse se termine par une conclusion qui évalue les résultats obtenus et discute quelques perspectives.

Première partie

État de l'Art

Chapitre 1

Algorithmes à Évolution Différentielle (DE)

Chapitre 1

Algorithmes à Évolution Différentielle

Introduction

Les scientifiques, ingénieurs et décideurs sont confrontés quotidiennement à des problèmes où l'objectif primaire est de trouver une solution optimale. Pour les problèmes simples, L'utilisation des méthodes d'optimisation exactes permette d'obtenir une solution dont l'optimalité est garantie. Par contre, pour les problèmes complexes, la solution optimale ne peut pas être obtenue par des méthodes exactes en un temps raisonnable. Les métaheuristiques sont des algorithmes destinées aux problèmes d'optimisation complexe. Elles permettent d'obtenir une solution de bonne qualité en un temps de calcul réduit sans garantie d'optimalité. Les algorithmes évolutionnaires sont des métaheuristiques basées sur la population. Ils sont faciles à mettre en œuvre et fournissent d'excellents résultats à faibles coûts. L'algorithme à évolution différentielle *(DE)* est parmi les algorithmes évolutionnaires les plus simples et les plus performants qui sont destinés aux problèmes d'optimisation complexe.

Dans ce chapitre, nous présenterons quelques terminologies liées aux métaheuristiques, algorithmes évolutionnaires *(AEs)* et aux algorithmes à évolution différentielle*(DEs)*. En particulier l'algorithme à évolution différentielle *EDEV* qui permet de trouver des solutions optimales en peu de temps, composé de trois variantes de *DE* très efficaces à savoir, *JADE*[24], *CoDE*[22] et *EPSDE*[17].

1.1 Problèmes d'Optimisation

Définition [10] : Un problème d'optimisation consiste à rechercher, parmi un ensemble de solutions (appelé aussi espace de décision ou espace de recherche), une solution optimale qui minimise ou maximise une fonction mesurant la qualité de cette solution. Cette fonction est appelée fonction objective.
Formellement, étant donné une fonction f définie sur un domaine P, il existe une solution optimale ($x^* \in P$),où :

- **max *f(x)* :** $f(x) \leq f(x^*)$, $\forall\ x \in P$. où : *f(x^*)* est la solution optimale et il n'y a pas d'autre solution plus grande.

$\Longrightarrow$ Ou

- **min *f(x)* :** $f(x) \geq f(x^*)$, $\forall\ x \in P$. où : *f(x^*)* est la solution optimale et il n'y a pas d'autre solution plus petite.

1.2 Métaheuristiques

Le mot métaheuristique est dérivé de la composition de deux mots grecs : méta, signifiant « à un plus haut niveau » et heuristique. Ce terme a été mentionné pour la première fois par Fred Glover[11]. Ce sont des algorithmes itératifs, possédant une composante aléatoire et parcourant l'espace de recherche par différentes techniques de génération de solutions. Les métaheuristiques permettent d'obtenir une solution de très bonne qualité pour des problèmes complexes dont on ne connaît pas des méthodes efficaces pour les traiter, ou bien quand la résolution du problème nécessite un temps de calcul élevé.

Le rapport entre le temps d'exécution et la qualité de la solution trouvée par une métaheuristique reste très intéressant par rapport aux différents types d'approches de résolution existants. De nombreux problèmes concrets ont été optimisés via des metaheuristiques : optimisation de réseaux mobiles UMTS, gestion du trafic aérien, optimisation des plans de chargement des cœurs de réacteurs nucléaires,...etc [4]

.

1.3 Métaheuristiques à Population de Solutions

Les métaheuristiques à base de population de solutions (les P- métaheuristiques) [20] sont des algorithmes d'optimisation itératifs. Ils permettent d'obtenir une valeur approchée à la solution optimale dans un temps raisonnable. les P- métaheuristiques manipulent un ensemble de solutions appelé population. Ils améliorent au fur et à mesure des itérations cette population en appliquant un processus de recherche qui comporte trois étapes. Tout d'abord, la génération d'une population initiale. Ensuite, la reproduction des individus (solutions) de la population courante pour obtenir une nouvelle population de solutions. Enfin, cette nouvelle population est intégrée à la population courante à l'aide de certaines procédures de sélection. Le processus de recherche est arrêté lorsqu'une condition donnée est satisfaite (critère d'arrêt).
Le pseudo code de P-métaheuristique est illustré dans l'algorithme 1.

Algorithm 1 High-level template of P-metaheuristics(de [20])

$P = P_0$; /*Generation of the initial population*/
$t = 0$;
repeat
  Generate ($P_t^{'}$) ; /*Generation a new population */
  P_{t+1}= Select-Population ($P_t \cup P_t^{'}$) ; /* Select new population */
  $t = t + 1$;
until Stopping criteria satisfied
$OUTPUT$: Best solution(s) found.

On distingue dans cette catégorie, les algorithmes évolutionnaires, qui sont une famille d'algorithmes issus de la théorie de l'évolution par la sélection naturelle, énoncée par Charles Darwin [7].

1.4 Algorithmes Évolutionnaires

Les algorithmes évolutionnaires (AE) sont des P-métaheuristiques d'optimisation basés sur la théorie Darwinienne de l'évolution des espèces biologiques [7]. Ils visent à trouver la solution optimale ou la solution quasi optimale parmi les membres d'une population. La solution est trouvée selon une fonction objective. Plus précisément, la structure des AEs prend

la forme de générations de population. Au départ, la population est généralement générée de manière aléatoire. Chaque individu de la population est la version codée d'une solution. Une fonction objective évalue la performance de chaque individu et lui associe une valeur. La population évolue durant plusieurs générations en sélectionnant, à chaque génération les individus les plus performants. Les individus sélectionnés sont reproduits à l'aide d'opérateur de croisement et de mutation pour générer de nouveaux individus. Enfin, un remplacement est appliqué pour déterminer les individus de la population qui vont survivre. Le processus est répété jusqu'à un certain critère d'arrêt [20].
Le pseudo code de l'algorithme évolutionnaire est illustré dans l'algorithme 2.

Algorithm 2 Template of an evolutionary algorithm.(de [20])

Generate ($P(0)$) ; /*Initial population*/
$t = 0$;
while not Termination Criterion($P(t)$) **do**
  Evaluate($P(t)$) ;
  $P^{'}(t)$ = Selection ($P(t)$) ;Evaluate ($P(t)$) ;
  $P^{'}(t)$ = Reproduction ($P^{'}(t)$) ;
  $P(t+1)$ = Replace ($P(t), P^{'}(t)$) ;
  $t = t+1$;
$OUTPUT$: Best individual or best population found.

1.5 Évolution Différentielle

L'algorithme a évolution différentielle (ED) est inspiré des stratégies évolutionnaires et des algorithmes génétiques. Il a été proposé par Rainer Storn et Kenneth Price en 1997 [19]. Il met en œuvre trois opérations issues des algorithmes évolutionnaires : la mutation, le croisement et la sélection. La méthode de génération d'un nouvel individu se fait en trois temps. En premier lieu, une mutation engendre un individu a partir d'un ensemble d'autres sélectionnés aléatoirement parmi la population. De multiples schémas de mutation ont été définis et seront détaillés par la suite. Il y a ensuite une phase de croisement, ou différentes stratégies peuvent être employées pour générer un individu issu du croisement entre le parent et l'individu généré précédemment. La sélection par remplacement est alors effectuée.

1.5.1 Principe

L'évolution différentielle est un algorithme à population, constituée de NP individus. Un individu $x_{i,G}$ est un vecteur de dimension D, ou D est la dimension du problème et G représente la génération. Une population initiale $x_{i,0}, i \in \{1, 2, ..., NP\}$ est créée par tirage aléatoire. L'algorithme effectue alors l'évolution de la population jusqu'à convergence. L'évolution se traduit itérativement par la création d'une nouvelle génération à partir des individus de la génération en cours, en appliquant les opérations de mutation et de croisement. La sélection s'effectue alors pour ne garder à la génération suivante que les meilleurs individus. Les différentes opérations impliquées dans la création d'une nouvelle génération sont les suivantes :

1.5.2 Mutation

Pour chaque individu cible $x_{i,G}$, un vecteur mutant $v_{i,G+1}$ est engendré selon :

$$v_{i,G+1} = x_{r1,G} + F.(x_{r2,G} - x_{r3,G}) \tag{1.1}$$

où $r_1, r_2, r_3 \in \{1, 2, ..., NP\}$ sont trois entiers différents entre eux et différents de i. Dans l'équation (1), F une constante appelée facteur d'amplification qui contrôle l'amplitude de la différence $(x_{r2,G} - x_{r3,G})$.

1.5.3 Croisement

Après la mutation, une opération de croisement entre les individus $v_{i,G+1}$ et $x_{i,G}$ est effectuée afin d'introduire de la diversité. Pour un opérateur de croisement binomial (bin), un vecteur $u_{i,G+1} = \{u1i, G+1, ..., uDi, G+1\}$ est généré par l'équation suivante :

$$u_{i,G+1} = \begin{cases} v_{ji,G+1} & \text{si} \quad (rand_j(0,1) \leq CR) \quad ou \quad (j = rnbr(i)), \\ x_{ji,G} & \text{si} \quad (rand_j(0,1) > CR) \quad ou \quad (j \neq rnbr(i)) \end{cases} \tag{1.2}$$

$$\forall j \in \{1, ..., D\}$$

Ici, $rand_j(0,1)$ est un nombre aléatoire $\in [0\,;1]$ tiré selon une loi de distribution uniforme pour la dimension j. $CR \in [0, 1]$ est une constante appelée taux de croisement et $rnbr(i)$ est un indice de dimension choisi aléatoirement permettant de s'assurer qu'au moins un paramètre de l'individu mutant $v_{i,G+1}$ est transmis à $u_{i,G+1}$.

1.5.4 Sélection

Afin de déterminer si l'individu généré appartiendra à la génération suivante (G+1) de la population, un critère de sélection est appliqué. Pour un problème de minimisation, on a :

$$x_{i,G+1} = \begin{cases} u_{i,G+1} & \text{si} \quad f(u_{i,G+1}) \leq f(x_{i,G}) \\ x_{i,G} & \text{sinon} \end{cases} \tag{1.3}$$

‘

Cet algorithme possède donc un nombre restreint de paramètres : la taille de la population NP ainsi que les coefficients F et CR. La taille de la population NP permet, lorsque sa valeur est élevée, de davantage explorer l'espace de recherche et augmente la diversité de directions possibles (i.e. les différences entre paires d'individus). Néanmoins, une valeur élevée diminue la capacité de convergence de l'algorithme. Le paramètre CR contrôle l'influence de l'individu cible, F influe sur l'importance de la trajectoire (i.e. la différence calculée en équation (1.1)). Le processus général de DE est décrit par l'algorithme 3.

Algorithm 3 Algorithme DE/rand/1/bin(de [20])

Phase d'initialisation de la population, G=0
Initialisation des N individus selon une distribution uniforme
// Phase d'évolution de la population
while critère de fin non satisfait **do**
  // Création de la $(G+1)^{ieme}$ génération
  for $i = 1 : NP$ **do**
    Sélection aléatoire de trois individus distincts de x_i et distincts en entre eux : $x_{r1,G}$, $x_{r2,G}$, $x_{r3,G}$
    // Mutation
    Génération de l'individu $v_{i,G+1}$, selon (1.1)
    // Croisement
    Création de l'individu $u_{i,G+1}$, selon (1.2)
  // Sélection
  for $i = 1 : NP$ **do**
    $x_{i,G+1}$ devient le meilleur individu entre $u_{i,G+1}$ et $x_{i,G}$ selon (1.3)

1.5.5 Schémas de Mutation

Le schéma de mutation présenté en équation (1.1) est un des schémas proposés par les auteurs. Il existe plusieurs variations de l'équation de mutation [20], décrites ci-après, basées sur le modèle suivant : DE/x/y/z où :

- x désigne le mode de sélection de l'individu auquel on applique la mutation. Dans l'équation (1.1), il est choisi aléatoirement, la méthode est nommée *rand*, ou par exemple *best* pour le meilleur individu de la population,
- y désigne le nombre de différences utilisées,
- z désigne le schéma de croisement, la variante présentée ici est bin pour binomiale.

Le schéma de la méthode d'évolution par défaut est alors DE/rand/1/bin. Une liste non exhaustive des mutations les plus rencontrées dans la littérature est présentée :

- **DE/rand/2/bin**

  une deuxième différence est ajoutée au calcul, cinq individus distincts sont nécessaires.

$$v_{i,G+1} = x_{r1,G} + F.(x_{r2,G} - x_{r3,G}) + F.(x_{r4,G} - x_{r5,G}). \tag{1.4}$$

- **DE/best/1**

  l'individu auquel est appliquée la mutation est le meilleur de la population.

$$v_{i,G+1} = x_{best,G} + F(x_{r1,G} - x_{r2,G}). \tag{1.5}$$

- **DE/best/2**

  identique au précédent, en ajoutant une différence entre deux individus distincts supplémentaires.

$$v_{i,G+1} = x_{best,G} + F(x_{r1,G} - x_{r2,G}) + F(x_{r3,G} - x_{r4,G}). \tag{1.6}$$

— **DE/rand-to-best/2**

on ajoute ici à une solution tirée aléatoirement la différence correspondant à la direction entre l'individu cible et le meilleur individu de la population.

$$v_{i,G+1} = x_{r1,G} + F(x_{r2,G} - x_{r3,G}) + F(x_{r4,G} - x_{r5,G}) + F(x_{best,G} - x_{i,G}). \quad (1.7)$$

— **DE/current-to-best/1**

identique au précédent, à ceci prés que l'individu auquel est appliquée la mutation est l'individu cible, et l'on ajoute la différence correspondant à la direction entre ce dernier et le meilleur individu de la population.

$$v_{i,G+1} = x_{i,G} + F(x_{r1,G} - x_{r2,G})F(x_{best,G} - x_{i,G}). \quad (1.8)$$

— **DE/current-to-rand/1**

on fait intervenir la direction entre l'individu cible et un individu supplémentaire, distinct.

$$v_{i,G+1} = x_{i,G} + K(x_{r3,G} - x_{i,G})F^{'}(x_{r1,G} - x_{r2,G}). \quad (1.9)$$

Le coefficient K est le coefficient de combinaison, choisi aléatoirement dans [0,1] et $F^{'} = K.F$.

1.6 L'algorithme à Évolution Différentielle EDEV

L'algorithme basé sur plusieurs populations (EDEV) est proposé par [23]. Il est composé de trois variantes de DE (JADE, CoDE, EPSDE). L'algorithme EDEV divise la population entière en quatre sous-populations. Trois sous-populations indicatrices et une sous-population de récompense. Les sous-populations indicatrices sont de taille égale et beaucoup plus petites que la sous-population de récompense. L'opérateur de partition est déclenché à chaque génération. Les trois sous-populations indicatrices sont désignées par *pop1*, *pop2* et *pop3* et la sous-population de récompense est représentée par *pop4*. Aux trois sous-populations indicatrices, sont affectés les trois variantes de DE JADE, CoDE et EPSDE. Au fur et à mesure que l'algorithme EDEV progresse, il évalue les performances de chaque sous-population. Après chaque certain nombre de générations (ng), la sous-population de récompense est affectée à la variante de DE la plus performante. De cette manière, différentes variantes de DE évoluent en coopération et celle qui fonctionne le mieux au cours du processus évolutif obtiendra le plus de ressources (représentées par les populations).
Soit NP la taille de la population et NP_i la taille de pop_i. λ_i désigne la proportion entre pop_i et la population pop.
On a donc :

$$NP_i = \lambda_i.NP \quad (1.10)$$

$$\sum_{i=1,...,4} \lambda_i = 1 \quad (1.11)$$

On a $\lambda_1 = \lambda_2 = \lambda_3$. Initialement, chaque sous-population indicatrice est attribuée de manière aléatoire à une variante de DE et la sous-population de récompenses est également attribuée à une variante de DE. À chaque génération, l'opération de partition de la population est exécutée une seule fois. Au fur et à mesure que l'algorithme progresse, après chaque nombre de générations, nous déterminons la variante DE la plus efficace (i_{best}) au cours de la dernière période en fonction du rapport entre les améliorations de la condition physique accumulées et les évaluations de la fonctions objective consommées.

$$i_{best} = argmax_{i=1,2,3}(\frac{\Delta f_i}{\Delta f_{esi}}) \quad (1.12)$$

où, Δf_i désigne les améliorations cumulées de de la fonction objective au cours des ng dernières générations attribuées par la $ieme$ variante de DE. Δf_{esi} est le nombre d'évaluations de la fonction objective.
Au cours des prochaines générations de ng, la sous-population de récompense sera affecté à la variante de DE la plus performante. Les opérateurs de détermination de la variante de DE et d'attribution de sous-population de récompense sont effectués périodiquement, ng étant la période. Le pseudo-code d'EDEV est illustré dans l'algorithme 4

Algorithm 4 Le pseudo-code de l'algorithme EDEV. (de [23])

1 : Initial parameters of EDEV including $ng, NP, \lambda_i, MaxG$ and $MaxFes$;
2 : Initial the parameters for JADE, CoDE and EPSDE ;
3 : Set $\Delta f_i = 0$ and Δf_{esi} for $i = 1, 2, 3$;
4 : Initialize the pop randomly distributed in the solution space ;
5 : Set $NP_i = \lambda_i.NP$;
6 : Randomly divide pop into pop_1 , pop_2 , pop_3 and pop_4 with respect to their sizes ;
7 : Randomly select a subpopulation pop_i(i = 1, 2, 3) and combine pop_i with pop_4.Let $pop_i = pop_i \bigcup pop_4$ and $NP_i = NP_i + NP_4$;
8 : Set g=0 ;
while $g \leq MaxG$ **do**
10 : $g = g + 1$;
11 : Execute JADE on pop_1 , update pop_1 and calculate Δf_1;
12 : Execute CoDE on pop_2 , update pop_2 and calculate Δf_2;
13 : Execute EPSDE on pop_3 , update pop_3 and calculate Δf_3;
14 : Combine updated pop_1, pop_2 and pop_3 into $pop, i.e., pop = \bigcup_{i=1,2,3} pop_i$;
if $mod(g, ng) == 0$ **then**
16 : $K = arg(max_{i=1,2,3}(\frac{\Delta f_i}{ng.NP_i}))$;
18 : Randomly partition pop into pop_1, pop_2, pop_3 and pop_4 ;
19 : Let $pop_k = pop_k \bigcup pop_4, K \in$ 1,2,3 ;

1.6.1 Variantes de DE

Pour atteindre des performances plus élevées d'EDEV, il est essentiel de s'assurer que les variantes de DE qui le composent sont puissantes tout en ayant des capacités différentes. Afin qu'elles puissent se soutenir mutuellement au cours du processus évolutif plutôt que de simplement se disputer des ressources. De nombreuses études ont montré l'importance d'avoir différents opérateurs et différentes stratégies dans un seul algorithme [12]. Ici, trois variantes de DE très classiques et efficaces sont prises comme algorithmes constitutifs, JADE [24], CoDE [22] et EPSDE [17]. La raison pour laquelle ces trois algorithmes sont choisis comme composants est que des études ont montré que JADE est une variante de DE polyvalente et qu'elle domine souvent d'autres variantes DE dans la résolution des problèmes d'optimisation unimodale. CoDE est très efficace dans la résolution de certains problèmes d'optimisation multimodale simples. Tandis qu'EPSDE présente des performances extraordinaires dans le traitement de certains problèmes d'optimisation composite très complexes [22]. Une brève introduction de ces trois variantes de DE est donnée ci-dessous.

Algorithme JADE

JADE, initialement développé par Zhang et Sanderson[24]. C'est une variante de DE simple mais efficace dans la résolution des problèmes d'optimisation unimodale. Dans JADE, une nouvelle stratégie de mutation current-to-pbest/1 est utilisée. La stratégie de mutation

"current-to-pbest/1" est indiquée ci-dessous.

$$v_{i,G} = x_{i,G} + F_i.(x_{pbest,G} - x_{r1,G}) + F_i.(x_{r2,G} - x_{r3,G}) \tag{1.13}$$

Une autre version de "current-to-pbest/1" avec une archive A est intégrée à la précédente. La version archivée de "current-to-pbest/1" est décrite comme suit :

$$v_{i,G} = x_{i,G} + F_i(x_{pbest,G} - x_{i,G}) + F_i(x_{r1,G} - \tilde{x}_{r2,G}) \tag{1.14}$$

où $x_{i,g}$ est le vecteur cible, $x_{r1,g}$ est sélectionné aléatoirement de la population, $x^p_{best,g}$ est sélectionné aléatoirement parmi les 100% meilleurs individus de la population en cours (P) avec p $\in$ (0, 1]. Tandis que $\tilde{x}_{r2,G}$ est choisi aléatoirement de l'union $P \cup A$, de la population en cours et de l'archive. F_i est le facteur de mutation associé à X_i, tirée aléatoirement à chaque génération. Initialement, l'archive A est vide. Ensuite, après chaque génération, les solutions parentes qui échouent dans le processus de sélection sont ajoutées à l'archive. Si la taille de l'archive dépasse un certain seuil, disons NP, alors certaines solutions sont supprimées aléatoirement de l'archive pour conserver la taille de l'archive à NP. L'archive fournit des informations sur la direction de la progression et il est également capable d'améliorer la diversité de la population.
Le pseudo-code de JADE avec archive est illustré dans l'algorithme 5.

Algorithm 5 Procédure de JADE avec Archive (de [24])

Set $\mu_{CR} = 0.5$; $\mu_F = 0.5$; $A = \emptyset$
Create a random initial population [$x_{i,0}$|i=1,2,...,NP]
for $g = 1 : G$ **do**
  $S_F = \emptyset$; $S_{CR} = \emptyset$;
  for $i = 1 : NP$ **do**
    Generate $CR_i = randn_i(\mu_{CR}, 0.1)$, $F_i = randc_i(\mu_F, 0.1)$
    Randomly choose $x^p_{best,g}$ as one of the 100P% best vectors
    Randomly choose $x_{r1,gi,g}$ from current population P
    Randomly choose $\tilde{x}_{r2,g} \neq x_{r1,g} \neq x_{i,g}$ from $P \cup A$
    $v_{i,g} = x_{i,g} + F_i.(x^p_{best,g} - x_{i,g}) + F_i.(x_{r1,g} - \tilde{x}_{r2,g})$
    Generate j_{rand} =$randint[1, D]$
    for $j = 1 : D$ **do**
      if $j = j_{rand}$ or rand(0,1)< CR **then**
        $u_{j,i,g} = v_{j,i,g}$
      else
        $u_{j,i,g} = x_{j,i,g}$
    if $f(x_{i,g}) \leq f(u_{i,g})$ **then**
      $x_{i,g+1} = x_{i,g}$
    else
      $x_{i,g+1} = u_{i,g}$; $x_{i,g} \longrightarrow A$; $CR_i \longrightarrow S_{CR}, F_i \longrightarrow S_F$
  Randomly remove solutions from A so that $|A| \leq NP$
  $\mu_{CR} = (1 - c)$. $\mu_{CR} + c$. $mean_A(S_{CR})$
  $\mu_F = (1 - c)$. $\mu_F + c$. $mean_L(S_F)$

Adaptation des paramètres

A chaque génération g, la probabilité de croisement CR_i de chaque individu x_i est générée indépendamment selon une distribution normale d'une moyenne μ_{CR} et d'écart type 0,1.

$$CR_i = randn_i(\mu_{CR}, 0.1) \tag{1.15}$$

La probabilité de croisement est arrondi à [0, 1]. S_{CR} désigne l'ensemble de toutes les probabilités de croisement CR_i réussies à la génération g. Le médiane μ_{CR} est initialisé à 0,5 puis mis à jour à la fin de chaque génération comme suit :

$$\mu_{CR} = (1 - c).\mu_{CR} + c.mean_A(S_{CR}) \tag{1.16}$$

où c est une constante positive entre 0 et 1 et $mean_A$ $(\cdot)$ est la moyenne arithmétique habituelle.
De même, à chaque génération g, le facteur de mutation F_i de chaque individu x_i est généré indépendamment selon une distribution de Cauchy avec un paramètre de localisation μ_F et un échelle paramètre 0,1.

$$F_i = randc_i(\mu_F, 0.1) \tag{1.17}$$

Lle facteur de mutatione facteur de mutation est arrondi à 1 si Fi ≥ 1 ou régénéré si Fi ≤ 0. Désignons S_F comme l'ensemble de tous les facteurs de mutation réussis dans la génération g. Le paramètre de localisation μ_F de la distribution de Cauchy est initialisé à 0,5 puis mis à jour à la fin de chaque génération comme :

$$\mu_F = (1 - c).\mu_F + c.mean_L(S_F) \tag{1.18}$$

où $mean_L$ $(\cdot)$ est la moyenne de Lehmer.

Un mécanisme sophistiqué d'adaptation des paramètres est exploité dans JADE. Les valeurs F et Cr qui contribuent à la génération de solutions prometteuses sont enregistrées et utilisées pour la nouvelle génération de paramètres et ainsi de meilleures valeurs de paramètres de contrôle sont propagées aux générations successives.

Algorithme CoDE

Le DE est un algorithme d'optimisation itératif qui suit le schéma classique des algorithmes évolutionnaires. Il peut être d'écrit en quatre phases : l'initialisation, la mutation, le croisement et la sélection. CoDE est une variante de DE proposé par Wang et al[22], dont l'idée principale est de combiner au hasard plusieurs stratégies de génération de vecteurs d'essai, avec un certain nombre de réglages de paramètres de contrôle à chaque génération pour créer de nouveaux vecteurs d'essai. L'idée est illustrée dans la figure (1.1).

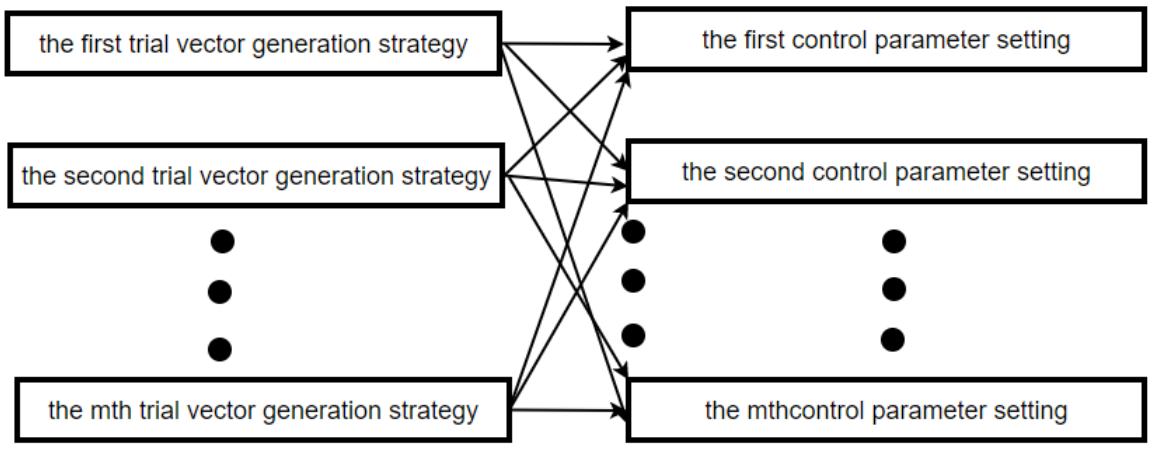


FIGURE 1.1 – Illustration de la combinaison de stratégies de génération de vecteurs d'essai avec les paramètres de contrôle.(de [22])

Dans CoDE, trois stratégies de génération de vecteurs d'essai sont choisis : «rand/1/bin», «rand/2/bin» et «current-to-rand/1» et trois combinaisons de paramètres de contrôle sont utilisées : [F = 1.0, Cr = 0.1], [F = 1.0, Cr = 0.9] et [F = 0.8, Cr = 0.2]. À chaque génération, chacune des trois stratégies de génération de vecteur d'essai est utilisée pour créer un nouveau vecteur d'essai avec un réglage de paramètre de contrôle choisi au hasard de l'ensemble de paramètres candidats. Par conséquent, trois vecteurs d'essai sont générés pour chaque vecteur cible. Le meilleur des trois survit pour la prochaine génération s'il est meilleur que son vecteur cible correspondant.
Le pseudo-code de CoDE est illustré dans l'algorithme 6.

Algorithm 6 Procédure de CoDE (de [22])

function $CoDE()$
Input : NP : the number of individuals at each generation,ie.,the population size
MAX_FES : maximum number of function evaluations
the strategy candidate pool : "rand/1/bin", "rand/2/bin" and "current-to-rand/1".
the parameter candidate pool : $[F = 1.0, C_r = 0.1], [F = 1.0, C_r = 0.9], [F = 0.8, C_r = 0.2]$.
generate an initial population $P_0 = \{\vec{x}_{1,0}, ..., \vec{x}_{NP,0}\}$
evaluate the objective function values $f(\vec{x}_{1,0}), ..., f(\vec{x}_{NP,0})$
$FES = NP$
while $FES < MAX\_FES$ **do**
$P_{g+1} = \emptyset$
for $i = 1 : NP$ **do**
- use the three trial vector generation strategies,each with a control parameter setting randomly selected from the parameter candidate pool, to generate three trial vectors $\vec{u}_{i_1,G}$, $\vec{u}_{i_2,G}$ and $\vec{u}_{i_3,G}$ for the target vector $\vec{x}_{i,G}$;
- evaluate the objective function values of the three trial vectors $\vec{u}_{i_1,G}$, $\vec{u}_{i_2,G}$ and $\vec{u}_{i_3,G}$;
- choose the best trial vector (denoted as $\vec{u}^*_{i,G}$) from the three trial vectors $\vec{u}_{i_1,G}$, $\vec{u}_{i_2,G}$ and $\vec{u}_{i_3,G}$;
$P_{g+1} = P_{g+1} \cup$ select$(\vec{x}_{i,G}, \vec{u}^*_{i,G})$
$FES = FES + 3$
$G = G + 1$

CoDE a montré des performances élevées dans la résolution de divers types de problèmes d'optimisation et en particulier dans le traitement de certains problèmes d'optimisation multimodale [22].

Algorithme EPSDE

L'efficacité de DE pour résoudre un problème d'optimisation dépend de la stratégie de mutation choisie et de ses valeurs de paramètres associées. Cependant, différents problèmes d'optimisation nécessitent différentes stratégies de mutation avec différentes valeurs de paramètres de contrôles. Motivés par ces observations, les chercheurs ont proposé l'algorithme EPSDE[17]. Il est une variante de DE dans laquelle un ensemble de stratégies de mutation, ainsi qu'un ensemble de valeurs correspondant à chaque paramètre associé sont en compétition pour produire une population de descendants réussies.

L'ensemble de stratégies de mutation est : "DE/best/2/bin", "DE/rand/1/bin" et "DE/current-to-rand/1/bin". l'ensemble des valeurs CR est pris dans l'intervalle [0,1-0,9] par pas de 0,1. l'ensemble des valeurs F est compris entre 0,4 et 0,9 par pas de 0,1.

Principe

Dans EPSDE, une stratégie de mutation aléatoire et des valeurs de paramètres tirées de l'ensemble de paramètres, sont affecté à chaque membre de la population initiale. Les membres de la population (vecteurs cibles) produisent des descendants (vecteurs d'essai), en

utilisant la stratégie de mutation et les valeurs de paramètres attribuées. Si le vecteur d'essai produit est meilleur que le vecteur cible, la stratégie de mutation et les valeurs des paramètres sont conservées avec le vecteur d'essai qui devient le parent (vecteur cible) dans la génération suivante. La combinaison de la stratégie de mutation et les valeurs des paramètres qui ont produit un descendant meilleur que le parent sont stockées. Si le vecteur cible est meilleur que le vecteur d'essai, alors le vecteur cible est réinitialisé au hasard avec une nouvelle stratégie de mutation et des valeurs de paramètres associées à partir des ensembles respectifs ou à partir des combinaisons réussies stockées avec une probabilité égale. Cela conduit à une probabilité de produire de meilleurs descendants grâce à une meilleure combinaison de la stratégie de mutation et des paramètres de contrôle associés dans les générations futures.
Le pseudo-code d'EPSDE est illustré dans l'algorithme 7.

Algorithm 7 Procédure de EPSDE (de [17])

function $EPSDE()$

- Set the generation number $G = 0$, and randomly initialize a population of NP individuals $P_G = \{X_{i,G}, ..., X_{NP,G}\}$ with $X_{i,G} = \{X^1_{i,G}, ..., X^D_{NP,G}\}\ i = 1, ..., NP$ uniformly distributed in the range X_{min}, X_{max} , where $X_{min} = \{X^1_{min}, ..., X^D_{min}\}$ and $X_{max} = \{X^1_{max}, ..., X^D_{max}\}$

-Select a pool of mutation strategies and a pool of values for each associated parameters corresponding to each mutation strategy.

-Each population member is randomly assigned with one of the mutation strategy from the pool and the associated parameter values are chosen randomly from the corresponding pool of values.

while stopping criterion is not satisfied **do**

/*Generate a mutated vector $V_{i,G} = \{v^1_{i,G}, ..., v^D_{i,G}\}$ for each target vector $X_{i,G}$*/

for $i = 1 : NP$ **do**

Generate a mutated vector $V_{i,G} = \{v^1_{i,G}, ..., v^D_{i,G}\}$ corresponding to the target vector $X_{i,G}$ using the mutation strategy and parameters associated to the target vector.

/*Generate a trial vector $U_{i,G} = \{u^1_{i,G}, ..., u^D_{i,G}\}$ for each target vector $X_{i,G}$*/

/*Binomial crossover*/

for $i = 1 : NP$ **do**

$j_{rand} = [rand[0, 1).D]$

for $J = 1 : D$ **do**

$$u^j_{i,G} = \begin{cases} v^j_{i,G} & \text{if } (rand_j \leq CR)\ or j = j_{rand} j = 1, 2, .., D \\ x^j_{i,G} & \text{otherwise} \end{cases} \tag{1.19}$$

/*selection by competition between target(parent) and trial(offspring) vectors */

for $i = 1 : NP$ **do**

/*evaluate the trial vector $U_{i,G}$*/

if $f(U_{i,G}) \leq f(X_{i,G})$ **then**

$X_{i,G+1} = U_{i,G}, f(X_{i,G+1}) \leq f(X_{i,G})$

if $f(U_{i,G}) \leq f(X_{best,G})$ **then**

$X_{best,G} = U_{i,G}, f(X_{best,G}) \leq f(U_{i,G})$

/*$X_{best,G}$ is the best individual in generation G*/

else

$X_{i,G+1} = X_{i,G}$

for $i = 1 : NP$ **do**

if $f(U_{i,G}) > f(X_{i,G})$ **then**

Randomly select a new mutation strategy and parameter values from the pools or from the stored successful combinations

Des études expérimentaux ont montré que par rapport à d'autres variantes de DE, EPSDE est particulièrement efficace dans la résolution de certains problèmes très complexes (par exemple les fonctions de composition hybride)[22, 17].

Conclusion

En premier lieu, dans ce chapitre, nous avons introduit les problèmes d'optimisation et la manière dont ils peuvent être résolus de façon plus ou moins générique et efficiente par les métaheuristiques. Nous avons introduit aussi des généralités sur les algorithmes évolutionnaires et un rappel sur l'évolution de ces algorithmes. Ce type d'algorithme est appliqué sur des problèmes d'optimisation complexe dans de nombreux domaines, tels que l'économie, la fabrication, la médecine, etc.

Parmi les algorithmes évolutionnaires existants, nous avons étudié plus en détail les métaheuristiques à évolution différentielle (DE).

Nous avons distingué principe de fonctionnement des DEs et les différentes opérations de reproduction (mutation, croisement et élection) et les différents schémas de mutation. Nous avons présenté aussi l'algorithme à évolution différentielle EDEV qui est très performant ainsi que les trois variantes puissantes qui le composent (JADE, Code et EPSDE). EDEV donne d'excellents résultats, mais il est coûteux en temps de calcul. Un problème qui ne peut pas être ignoré par l'utilisateur qui cherche toujours les meilleurs résultats dans un temps réduit. Dans ce travail, on vise à paralléliser cet algorithme. Le calcul parallèle sera l'objectif du prochain chapitre.

Chapitre 2

Calcul Parallèle

Introduction

Le parallélisme est un concept de plus en plus populaire et contribue grandement à satisfaire la demande de performance croissante en informatique. Considérées à leurs débuts comme réservées à une élite, les machines parallèles sont, de nos jours, abordables et accessibles et la technologie pour les manipuler est devenue beaucoup plus conviviale et performante. Cette prolifération rapide de la technologie parallèle a atteint beaucoup de domaines parmi lesquels l'optimisation.

Les problèmes d'optimisation réels sont souvent complexes, leur modélisation continue d'évoluer en termes de contraintes et d'objectifs, et leur résolution est coûteuse en temps de calcul CPU. Bien que des algorithmes presque optimaux tels que les métaheuristiques permettent de réduire la complexité de leur résolution, elles restent insuffisantes pour résoudre des problèmes de grande taille en un temps raisonnable.

Dans ce chapitre, nous allons introduire les notions de base du parallélisme, la gestion de mémoire des ordinateurs parallèles et les modèles de programmation parallèle. Ces notions sont nécessaires dans le parallélisme de l'algorithme à évolution différentielle EDEV.

2.1 Motivations pour le Calcul Parallèle

Le développement rapide de la technologie dans la conception de processeurs (par exemple, processeurs multi -cœurs), les réseaux (par exemple, les réseaux locaux - LAN - tels que Myrinet et Infiniband, les réseaux étendus - WAN - tels que les réseaux optiques) et le stockage de données a fait de plus en plus recours au calcul parallèle [20]. De telles architectures représentent une stratégie efficace pour la conception et la mise en œuvre des métaheuristiques parallèles. De nos jours, même les ordinateurs portables et les postes de travail sont équipés de processeurs multi-cœurs. De plus, le rapport coût/performance est en constante diminution.
Le calcul parallèle peut être utilisé dans la conception et la mise en œuvre des métaheuristiques pour les raisons suivantes :

— L'un des principaux objectifs de la parallélisassions d'une métaheuristique est de réduire le temps de recherche. Cela aide à concevoir des méthodes d'optimisation en temps réel et interactives. C'est un aspect très important pour certaines classes de problèmes où il y a des exigences strictes sur le temps de recherche, comme dans les problèmes d'optimisation dynamique et les problèmes de contrôle critiques en temps tels que la planification «en temps réel».

— L'objectif principal d'une coopération parallèle entre métaheuristiques est d'améliorer la qualité des solutions. Une meilleure convergence et un temps de recherche réduit peuvent se produire. Notons qu'un modèle parallèle de métaheuristique peut être plus efficace qu'une métaheuristique séquentielle même sur un seul processeur.
— Une métaheuristique parallèle peut être plus robuste pour résoudre de manière efficace, différent problèmes d'optimisation.
— Les métaheuristiques parallèles permettent de résoudre des instances à grande échelle de problèmes d'optimisation complexes qui ne peuvent pas être résolues par une machine séquentielle.

Le calcul parallèle a influencé une variété de domaines allant des simulations informatiques pour les applications scientifiques et d'ingénierie aux applications commerciales dans l'extraction de données et le traitement transactionnel. Les avantages du parallélisme en termes de coûts associés aux exigences de performances des applications, présentent des arguments convaincants en faveur du calcul parallèle.[14]

2.2 Concepts et Terminologie

Il s'agit de la terminologie la plus importante du calcul parallèle.

2.2.1 Traitement Parallèle

La vieille vision du calcul parallèle est que le calcul parallèle n'est plus qu'une simple stratégie pour atteindre de hautes performances [8]. Or, le calcul parallèle concerne le matériel et les logiciels de calcul dans lesquels de nombreux calculs sont effectués simultanément. L'objectif principal du calcul parallèle est l'amélioration de la capacité de calcul [13]. Dans le sens le plus simple, le calcul parallèle est l'utilisation de plusieurs ressources de calcul pour résoudre les problèmes de calcul. Un problème est divisé en parties discrètes qui peuvent être résolues simultanément, où chaque partie est ensuite décomposée en une série d'instructions, chaque instruction de chaque partie s'exécute simultanément sur plusieurs processeurs comme La figure 2.1 montre.

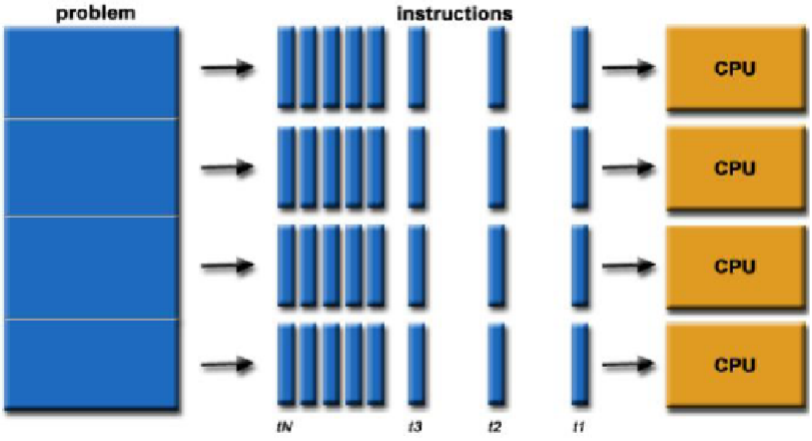


FIGURE 2.1 – Calcul Parallèle (de [13]).

2.2.2 Machine Parallèle

Les machines parallèles fournissent de grandes quantités de puissance de calcul, mais ils le font au prix d'une difficulté accrue de programmation et d'utilisation. Certes, un uni-

processeur suffisamment rapide serait plus simple à utiliser [8]. Aujourd'hui, pratiquement tous les ordinateurs autonomes sont parallèles d'un point de vue matériel :

- Plusieurs unités fonctionnelles (virgule flottante, entier, GPU, etc.),
- Plusieurs unités d'exécution / cœurs,
- Plusieurs threads matériels.

2.2.3 Architecture de Von Neumann

John Von Neumann était un mathématicien, physicien et informaticien. Il a apporté des contributions majeures à un certain nombre de domaines, notamment les mathématiques (fondements des mathématiques, analyse fonctionnelle, théorie de la représentation, algèbres d'opérateurs, géométrie, topologie et analyse numérique...) [21]. L'architecture dite architecture de Von Neumann est un modèle pour un ordinateur qui utilise une structure de stockage unique pour conserver à la fois les instructions et les données demandées ou produites par le calcul. De telles machines sont aussi connues sous le nom d'ordinateur à programme enregistré. La séparation entre le stockage et le processeur est implicite dans ce modèle.

2.2.4 Flux d'Instructions et Flux de Données

Le terme «flux» fait référence à une séquence d'instructions ou de données exploitées par l'ordinateur. Dans le cycle complet d'exécution des instructions, un flux d'instructions de la mémoire principale vers la CPU est établi. Dans le même temps, il y a un flux d'opérandes entre le processeur et la mémoire de manière bidirectionnelle [14]. Ce flux d'opérandes est appelé un flux de données.

2.2.5 Types de Classification

Une fois que tous les termes de base du traitement parallèle et du calcul ont été définis parallèle. Les ordinateurs parallèles peuvent être caractérisés sur la base des flux de données et d'instructions formant divers types d'organisations informatiques. Ils peuvent également être classés en fonction de la structure de l'ordinateur, par ex. plusieurs processeurs ayant une mémoire séparée ou une mémoire globale partagée. Des niveaux de traitement parallèles peuvent également être définis en fonction de la taille des instructions. Ainsi, les ordinateurs parallèles peuvent être classés en fonction de divers critères [14].

2.2.6 Classification de Flynn

La classification des architectures des ordinateurs la plus populaire a été définie par Flynn en 1966 [9]. Le système de classification de Flynn est basé sur la notion de flux d'informations. Deux types d'informations circulent dans un processeur : les instructions et les données. Le flux d'instructions est définie comme la séquence d'instructions exécutée par l'unité de traitement et le flux de données est définie comme le trafic de données échangé entre la mémoire et l'unité de traitement. Selon que l'un ou l'autre des flux d'instructions ou de données est unique ou multiple, l'architecture informatique dans la taxonomie de Flynn, peut être classée dans les quatre catégories distinctes suivantes 2.2.

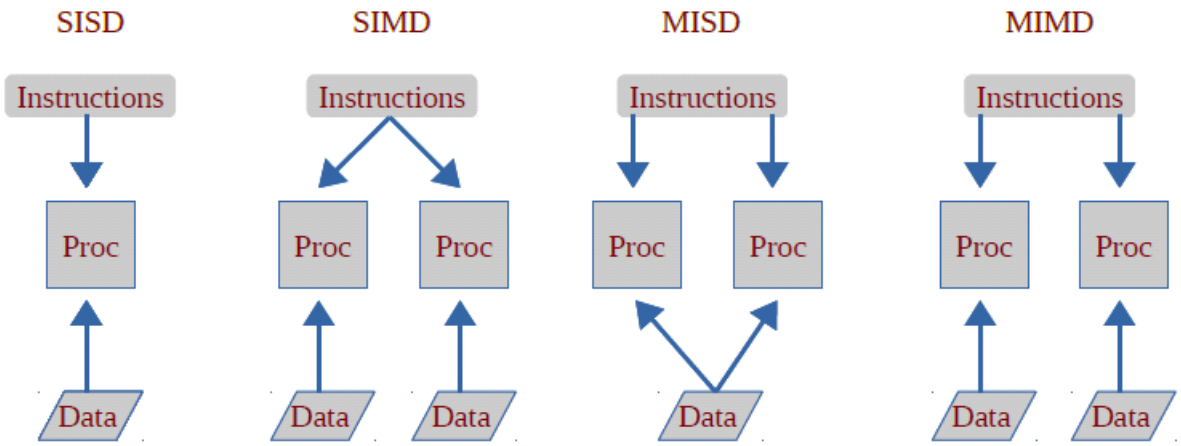


FIGURE 2.2 – Les classes d'architectures de la taxonomie de Flynn (de [13]).

— **Single Instruction and Single Data stream (SISD)**

Dans cette organisation, l'exécution séquentielle des instructions est effectuée par une CPU contenant un seul élément de traitement . Par conséquent, les machines SISD traitent qu'un seul flux d'instructions et un seul flux de données.

— **Single Instruction and Multiple Data stream (SIMD)**

Dans cette organisation, plusieurs éléments de traitement travaillent sous le contrôle d'une seule unité de contrôle. Il y a une instruction et plusieurs flux de données. Tous les éléments de traitement reçoivent la même instruction diffusée de l'UC. La mémoire principale peut également être divisée en modules pour générer plusieurs flux de données, agissant comme une mémoire distribuée. Par conséquent, tous les éléments de traitement exécutent simultanément la même instruction. Chaque processeur prend les données de sa propre mémoire et il dispose de flux de données distincts (Certains systèmes fournissent également une mémoire globale partagée pour les communications). Chaque processeur doit être autorisé à terminer son instruction avant que l'instruction suivante ne soit exécutée. Ainsi, l'exécution des instructions est synchrone.

— **Multiple Instruction and Single Data stream (MISD)**

Dans cette organisation, plusieurs éléments de traitement sont organisés sous le contrôle de plusieurs unités de contrôle. Où chaque unité de contrôle gère un flux d'instructions qui est traitée via son élément de traitement correspondant. Mais chaque élément de traitement ne traite qu'un seul flux de données à la fois. Par conséquent, pour gérer plusieurs flux d'instructions et un flux de données unique, plusieurs unités de contrôle et plusieurs éléments de traitement sont organisés dans cette classification. Tous les éléments de traitement interagissent avec la mémoire partagée commune pour l'organisation d'un flux de données unique [14].

— **Multiple Instruction and Multiple Data stream (MIMD)**

Dans cette organisation, plusieurs éléments de traitement et plusieurs unités de contrôle sont organisés comme dans MISD. Mais la différence est dans cette organisation, plusieurs flux d'instructions fonctionnent sur plusieurs flux de données. Par conséquent, pour gérer plusieurs flux d'instructions, plusieurs unités de commande et plusieurs

éléments de traitement sont organisés de telle sorte que plusieurs éléments de traitement traitent plusieurs flux de données de la mémoire principale. Les processeurs travaillent sur leurs propres données avec leurs propres instructions. Les tâches exécutées par différents processeurs peuvent démarrer ou se terminer à des moments différents. Ils ne sont pas verrouillés, comme dans les ordinateurs SIMD, mais fonctionnent de manière asynchrone. Cette classification reconnaît en fait l'ordinateur parallèle. Cela signifie que dans le vrai sens, l'organisation MIMD est dite être un ordinateur parallèle. Tous les systèmes multiprocesseurs relèvent de cette classification [14].

2.3 Gestion de Mémoire des Ordinateurs Parallèles

2.3.1 Mémoire Partagée

Les ordinateurs à mémoire partagée sont des ordinateurs parallèles qui varient considérablement, mais ont un facteur en commun qui est la capacité pour tous les processeurs d'accéder à toute la mémoire en tant qu'espace d'adressage global. Plusieurs processeurs peuvent fonctionner indépendamment mais ils partagent les mêmes ressources mémoire. Les changements dans un emplacement de mémoire effectués par un processeur sont également visibles par tous les autres processeurs. Historiquement, les machines à mémoire partagée ont été classées comme UMA et NUMA, en fonction des temps d'accès à la mémoire [14].

— **Accès Mémoire Uniforme (UMA) :**

Le plus souvent représentés aujourd'hui par les machines à multiprocesseurs symétriques (SMP), ils ont des processeurs identiques et des temps d'accès égaux à la mémoire. Si un processeur met à jour un emplacement dans la mémoire partagée, tous les autres processeurs connaissent la mise à jour. La cohérence du cache est réalisée au niveau matériel [14].

— **Accès Mémoire Non Uniforme (NUMA) :**

Souvent réalisé en reliant physiquement deux ou plusieurs SMP, où un SMP peut accéder directement à la mémoire d'un autre SMP. Tous les processeurs n'accèdent pas au même temps à toutes les mémoires. L'accès à la mémoire via une liaison est plus lent mais la cohérence du cache est maintenue.

1. **Avantages :**

Il y a deux avantages. Le premier est que l'espace d'adressage global offre une perspective de programmation conviviale à la mémoire. Deuxièmement, le partage de données entre les tâches est à la fois rapide et uniforme en raison de la proximité de la mémoire avec les processeurs.

2. **Inconvénient :**

Le principal inconvénient est le manque d'adaptation entre la mémoire et les processeurs. L'ajout de processeurs supplémentaires peut augmenter géométriquement le trafic sur le chemin mémoire-processeur partagé, et pour les systèmes cohérents en cache, augmenter géométriquement le trafic associé à la gestion du cache / mémoire. Ensuite, le programmeur est le responsable de la synchronisation pour garantir un accès "correct" à la mémoire globale.

2.3.2 Mémoire Distribuée

Comme les systèmes à mémoire partagée, les systèmes à mémoire distribuée varient considérablement mais partagent une caractéristique commune. Les systèmes de mémoire distribuée nécessitent un réseau de communication pour connecter la mémoire inter-processeur. Où les processeurs ont leur propre mémoire locale. L'adresses mémoire dans un processeur ne correspondent pas à un autre processeur, il n'y a donc pas de concept d'espace d'adressage global sur tous les processeurs. Chaque processeur fonctionne indépendamment car, il dispose d'une mémoire locale propre à lui. Les modifications qu'il apporte à sa mémoire locale n'ont aucun effet sur la mémoire des autres processeurs. Par conséquent, le concept de cohérence du cache ne s'applique pas. Lorsqu'un processeur a besoin d'accéder à des données dans une mémoire d'un autre processeur, c'est généralement au programmeur de définir explicitement comment et quand les données sont communiquées. La synchronisation entre les tâches est également le travail du programmeur [14].

2.3.3 Mémoire Hybride

Les ordinateurs les plus grands et les plus rapides du monde utilisent aujourd'hui des architectures de mémoire partagée et distribuée. Le composant de mémoire partagée peut être une machine à mémoire partagée et / ou un GPU. Où les processeurs sur un nœud de calcul partagent le même espace mémoire et nécessitent une communication pour échanger des données entre les nœuds de calcul.

2.4 Modèles de Programmation Parallèle

Un modèle de programmation parallèle spécifie la vue du programmeur sur un ordinateur parallèle en définissant comment le programmeur peut coder un algorithme. Cette vue est influencée par la conception architecturale et le langage, le compilateur ou les bibliothèques d'exécution et, par conséquent, il existe de nombreux modèles de programmation parallèle différents pour une même architecture. Nous décrivons dans cette partie quelques modèles de programmation parallèle les plus utilisés et les plus connus.

2.4.1 Modèle à Mémoire Partagée

Dans ce modèle de programmation, les processus / tâches partagent un espace d'adressage commun, dans lequel ils lisent et écrivent de manière asynchrone. Divers mécanismes tels que les verrous / sémaphores sont utilisés pour contrôler l'accès à la mémoire partagée, résoudre les conflits et prévenir les conditions de concurrence et les blocages. C'est peut-être le modèle de programmation parallèle le plus simple. Un avantage de ce modèle du point de vue du programmeur est que la notion de l'appropriation des données manque, il n'est donc pas nécessaire de spécifier explicitement la communication des données entre les tâches. Tous les processus ont un accès égal à la mémoire partagée. Le développement de programmes peut souvent être simplifié avec un inconvénient important en termes de performances, c'est qu'il devient plus difficile de comprendre et de gérer la localité des données.

2.4.2 Modèle de Threads

Ce modèle de programmation est un type de programmation à mémoire partagée. Dans le modèle de threads de programmation parallèle, un seul processus «lourd» peut avoir plusieurs chemins d'exécution simultanés «légers». Pour les applications où la charge de travail dépend des entrées de l'application qui peuvent varier considérablement, retarder la

décision sur le nombre de threads à utiliser jusqu'à l'exécution, lorsque les tailles d'entrée peuvent être examinées. Des exemples de paramètres d'entrée qui affectent le nombre de threads incluent des éléments tels que la taille de la matrice, la taille de la base de données, la taille et la résolution de l'image / vidéo, la profondeur / largeur / des structures arborescentes et la taille des structures basées sur des listes. De même, pour les applications conçues pour s'exécuter sur des systèmes où le nombre de processeurs peut varier considérablement, reporter la décision sur le nombre de threads à utiliser jusqu'à l'exécution de l'application lorsque la taille de la machine peut être examinée.

Pour les applications où la quantité de travail est imprévisible à partir des données d'entrée, envisagez d'utiliser une étape le calibrage pour comprendre la charge de travail et les caractéristiques du système afin de vous aider à choisir un nombre approprié de threads. Si l'étape de calibrage est coûteuse, les résultats de calibrage peuvent être stockés dans un endroit permanent comme le système de fichiers. Il est déconseillé de créer plus de threads que le nombre de processeurs sur le système, lorsque tous les threads peuvent être actifs simultanément ; cette situation fait que le système d'exploitation multiplexe les processeurs et produit généralement des performances sous-optimales.

Lors du développement d'une bibliothèque par opposition à une application entière, fournissez un mécanisme par lequel l'utilisateur de la bibliothèque peut sélectionner de manière pratique le nombre de threads utilisés par la bibliothèque, car il est possible que l'utilisateur ait un parallélisme de niveau supérieur qui rend le parallélisme dans la bibliothèque inutile [8].

2.4.3 Mémoire Distribuée/ Modèle de Transmission de Message

Le standard Message Passing Interface [6], annoncé en 1994 par le MPI Forum, est l'aboutissement des travaux d'un consortium de constructeurs, d'industriels et d'universitaires, réunis pour proposer une solution standard dédiée au calcul parallèle sur grappes et architectures parallèles. Cette interface de passage de messages définit un ensemble de fonctionnalités et de spécifications génériques, entièrement indépendantes des architectures. Elle s'articule autour de primitives de communications point-à-point et d'opérations collectives. MPI propose une interface de programmation définissant un ensemble de fonctions de communication ainsi que des recommandations pour leurs implantations. Plus précisément, MPI propose plusieurs méthodes de communication point à point telles que MPI_Send (envoi) et MPI_Recv (réception) ainsi que des communications collectives comme, par exemple, MPI_Barrier (barrière) et MPI_Broadcast (diffusion). En outre, afin d'accroître la portabilité des applications, MPI propose des types de données de base et donne la possibilité de créer des types complexes et structurés [18].

Les applications MPI créent plusieurs processus qui peuvent exécuter le même programme (modèle SPMD) ou un programme différent (modèle MPMD :Multiple Program Multiple Data) avec des données différentes. Au cours de l'exécution les processus s'échangent des messages suivant différents modes de communications : bloquant, non-bloquant, bufférisé, synchrone.

2.4.4 Modèle Parallèle de Données

Peut également être appelé modèle d'espace d'adressage global partitionné (PGAS). Le modèle parallèle de données démontre les caractéristiques qui sont :

— L'espace d'adressage est traité globalement,

— La plupart des travaux parallèles se concentrent sur l'exécution d'opérations sur un ensemble de données. L'ensemble de données est généralement organisé en une structure commune, telle qu'un tableau,
— Un ensemble de tâches travaille collectivement sur la même structure de données, cependant, chaque tâche fonctionne sur une partition différente de la même structure de données,
— Les tâches exécutent la même opération sur leur partition de travail, par exemple, «ajouter 4 à chaque élément du tableau».

Dans les architectures à mémoire partagée, toutes les tâches peuvent avoir accès à la structure de données via la mémoire globale. Dans les architectures à mémoire distribuée, la structure de données globale peut être divisée logiquement et / ou physiquement entre les tâches.

2.4.5 Modèle Hybride

Un modèle hybride combine plus d'un des modèles de programmation décrits précédemment. Actuellement, un exemple courant de modèle hybride est la combinaison du modèle de passage de message (MPI) avec le modèle de threads (OpenMP). Les threads exécutent des noyaux intensifs en calcul en utilisant des données locales sur les nœuds et les communications entre les processus sur différents nœuds se produisent sur le réseau à l'aide de MPI. Ce modèle hybride se prête bien à l'environnement matériel le plus populaire des machines multi / plusieurs cœurs en grappe. Un autre exemple similaire et de plus en plus populaire d'un modèle hybride est l'utilisation de MPI avec la programmation CPU-GPU (Graphics Processing Unit).
Les tâches MPI s'exécutent sur des processeurs utilisant la mémoire locale et communiquant entre eux sur un réseau. Un ensemble de tâches fonctionne collectivement sur la même structure de données, cependant, chaque tâche fonctionne sur une partition différente de la même structure de données. L'échange de données entre la mémoire locale du nœud et les GPU utilise CUDA (ou quelque chose d'équivalent).

2.4.6 SPMD et MPMD

SPMD est en fait un modèle de programmation «de haut niveau» qui peut être construit sur n'importe quelle combinaison des modèles de programmation parallèle mentionnés précédemment. PROGRAMME UNIQUE : Toutes les tâches exécutent leur copie du même programme simultanément. Ce programme peut être des threads, le passage de messages, des données parallèles ou hybrides. Les programmes SPMD ont généralement la logique nécessaire programmée en eux pour permettre à différentes tâches de se brancher ou d'exécuter conditionnellement uniquement les parties du programme qu'elles sont conçues pour exécuter. Autrement dit, les tâches ne doivent pas nécessairement exécuter le programme entier, peut-être seulement une partie de celui-ci. Le modèle SPMD, utilisant le passage de messages ou la programmation hybride, est probablement le modèle de programmation parallèle le plus couramment utilisé pour les grappes multi-nœuds [14].

Conclusion

Pour conclure ce chapitre, nous devons savoir qu'il n'est pas destiné à couvrir en profondeur la programmation parallèle, car cela demanderait beaucoup plus de temps. Ce chapitre a commencé par une discussion sur le calcul parallèle - ce que c'est et comment il est utilisé, suivi d'une discussion sur les concepts et la terminologie associés au calcul parallèle. Les thèmes des architectures de mémoire parallèle et des modèles de programmation sont ensuite explorés.

Dans le prochain chapitre, nous utiliserons l'une de ces techniques pour paralléliser l'algorithme à évolution différentielle « EDEV ».

Deuxième partie

Implémention et Parallélisation d'EDEV et Démonstration de son Efficacité

Chapitre 3

Analyse et Conception du Système

Chapitre 3

Analyse et Conception du Système

Introduction

Après avoir pris connaissance dans le premier chapitre sur les métaheuristiques et les algorithmes évolutionnaires. Nous avons étudié plus en détail les algorithmes à évolution différentielle. En particulier l'algorithme EDEV qui est très puissant et donne de très bon résultats, mais il est très coûteux en temps de calcul. EDEV consomment beaucoup de temps d'exécution car c'est un algorithme à évolution différentielle, en plus il est composé de trois variantes de DE (JADE, CoDE et EPSDE). Ces limites ont provoqué l'apparition de la version parallèle d'EDEV (PEDEV), qui est équivalente à la version séquentielle de l'algorithme EDEV en termes de résultats mais elle est beaucoup plus rapide que cette dernière.

Ce chapitre est divisé en cinq sections. La première section est consacrée à décrire la problématique de ce projet. La deuxième section explique le cycle du projet. La troisième section décrit la conception globale de l'application. Ensuite, les deux dernières sections sont dédiées à la conception détaillée des deux versions (version parallèle et version séquentielle), où chaque section contient l'analyse et la conception de chaque version.

3.1 Description du Problème

Cette thèse ne se limite pas à implémenter un algorithme évolutionnaire ou un algorithme à évolution différentielle. L'objectif principal est d'implémenter la version séquentielle de l'algorithme EDEV et de proposer une version parallèle de l'algorithme EDEV (PEDEV) et de comparer les résultats de ces deux implémentions. Cela nous fait penser que le principal problème abordé par cette thèse est comment paralléliser EDEV. Le problème est plutôt qu'un ensemble de plusieurs sous-problèmes, comment réaliser les deux versions de l'algorithme et comment former une application avec la possibilité de comparaison entre leurs résultats et aussi comment choisir le problème d'optimisation qui va être résolu par cet algorithme. La figure montre la problématique du projet 3.1.

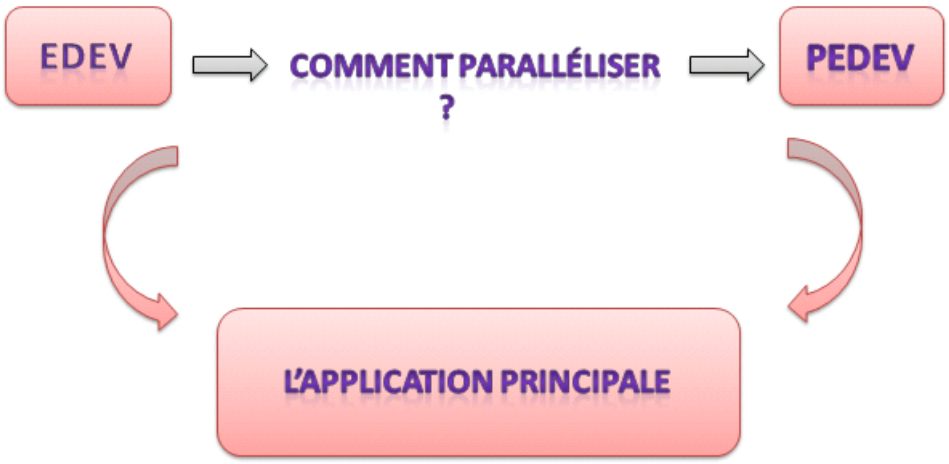


FIGURE 3.1 – Problématique du projet.

3.2 Cycle de Projet

Après avoir décrit la problématique de cette étude dans la dernière section, il est clair maintenant que notre projet sera divisé en trois parties qui sont :

- Implémenter la version séquentielle d'EDEV,
- Paralléliser EDEV et obtenir une version parallèle d'EDEV (c'est-à-dire PEDEV),
- Comparer les résultats des deux versions.

Toutes ces étapes sont décrites dans la figure 3.2.

FIGURE 3.2 – Cycle de projet.

3.3 Conception Globale de l'Application

La figure 3.3 montre l'architecture de l'application avec ses entrées et ses sorties.

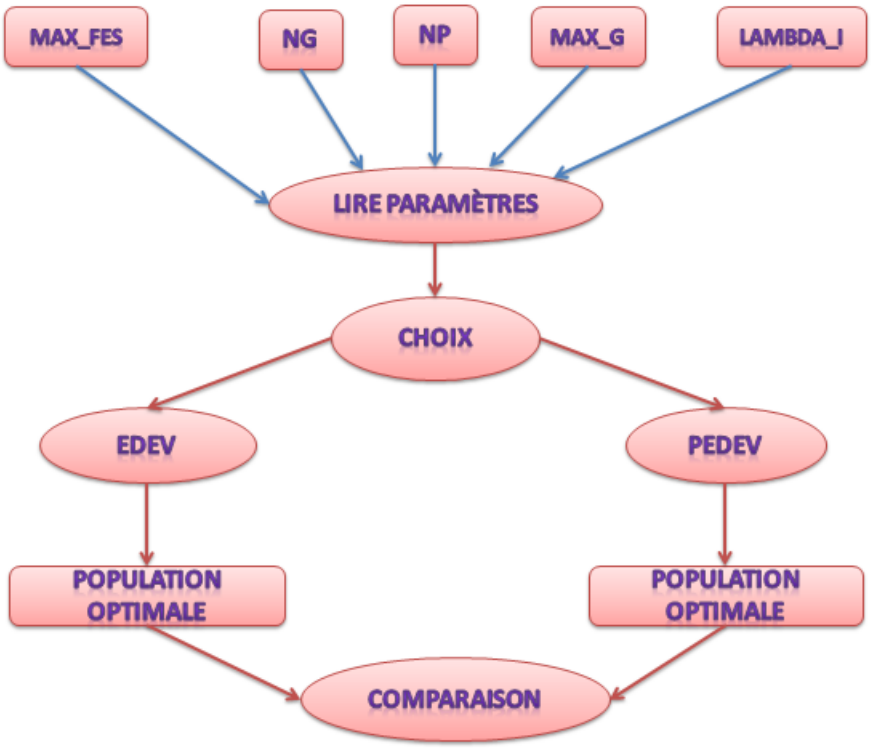


FIGURE 3.3 – Architecture globale de l'application.

Dans notre application, nous permettons à l'utilisateur d'entrer les données suivantes :

- MAX_FES représente le critère d'arrêt de l'algorithme CoDE,
- Ng représente la période dans laquelle se fait l'évaluation des performances de chaque sous-population,
- NP représente la taille de la population initiale,
- MAX_G représente le nombre de génération,
- Le nombre λ_i contrôle la taille de chaque sous population ($0 \leq \lambda_i \leq 1$).

3.3.1 Organigramme de l'Algorithme JADE

La figure (3.4) montre l'organigramme de l'algorithme JADE avec ses entrées et ses sorties.

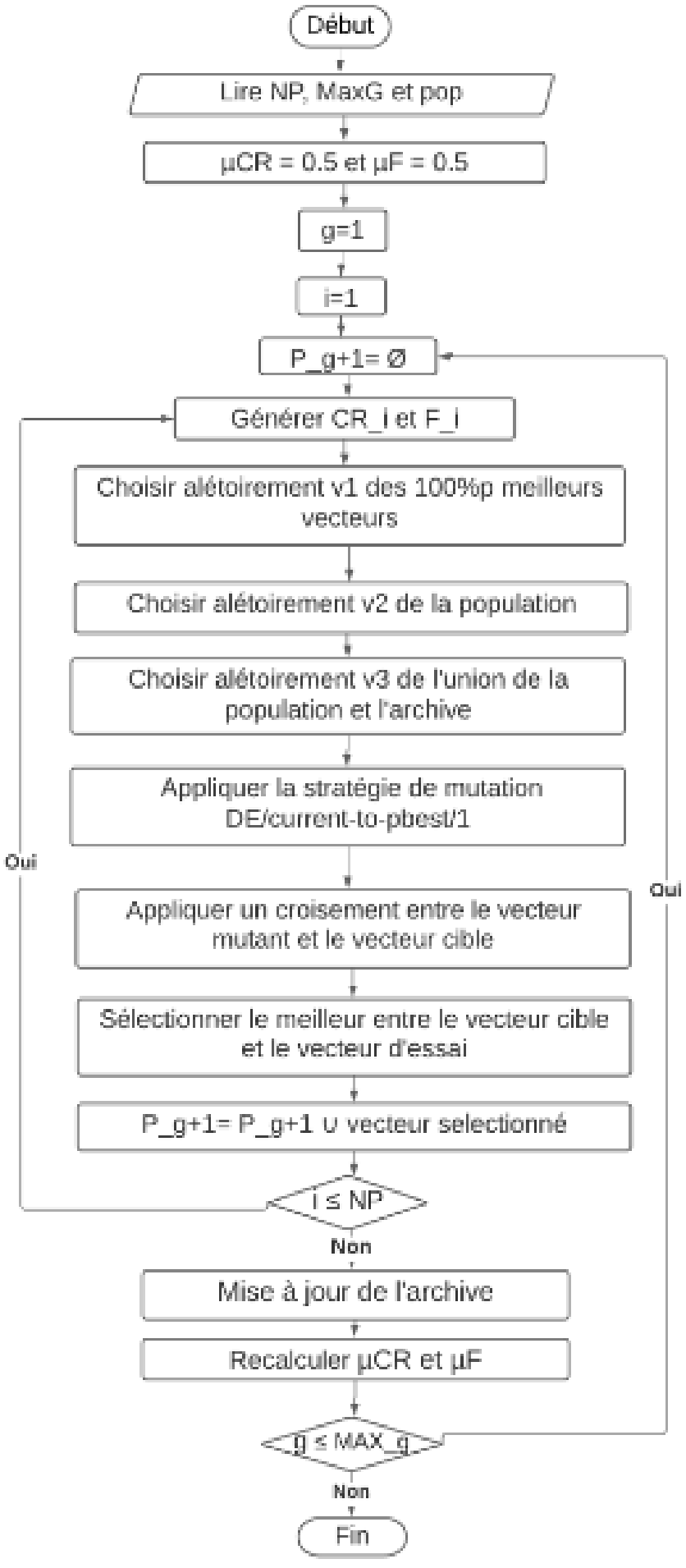


FIGURE 3.4 – Organigramme de l'algorithme JADE.

3.3.2 Organigramme de l'Algorithme CoDE

La figure (3.5) montre l'organigramme de l'algorithme CoDE avec ses entrées et ses sorties.

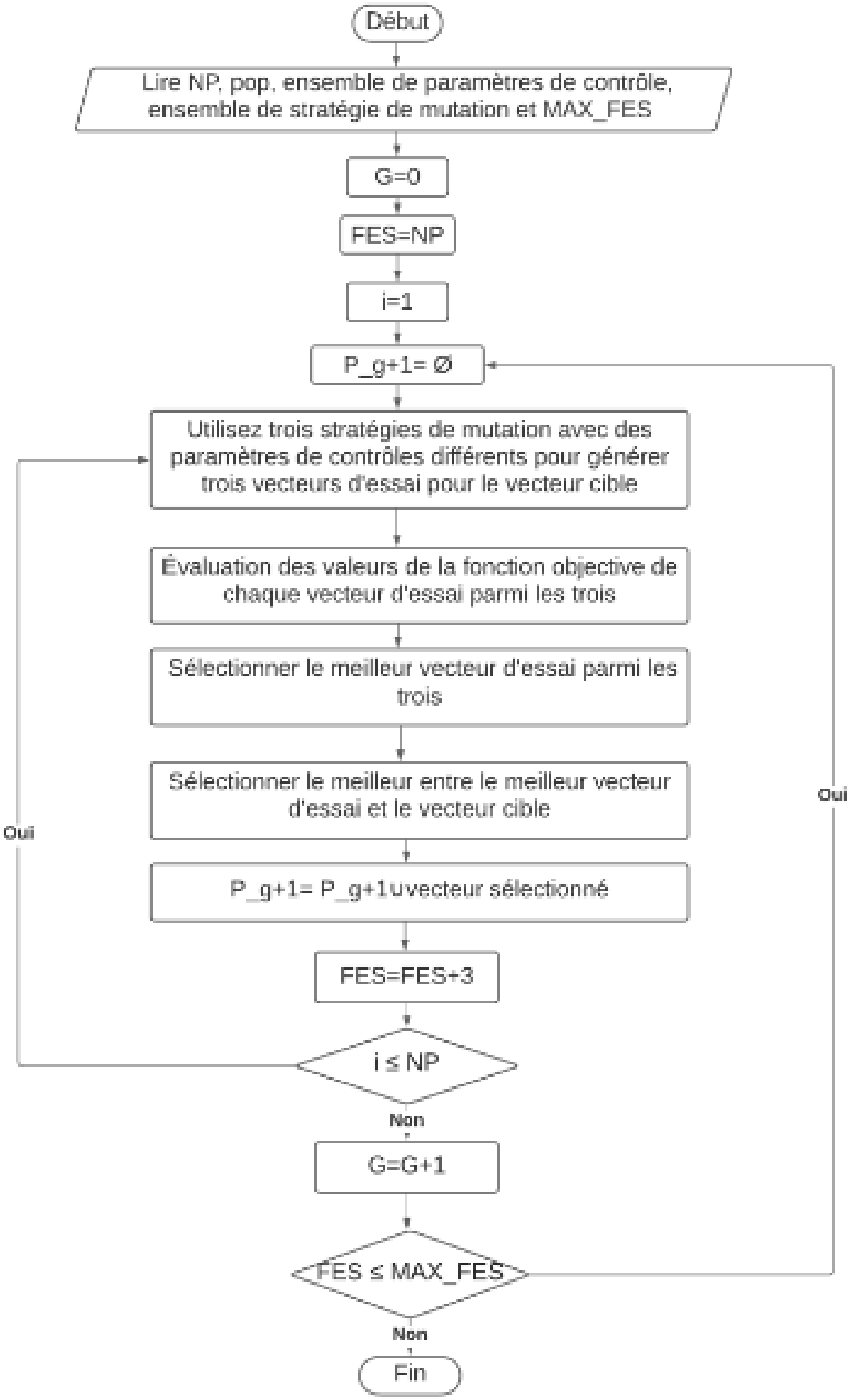


FIGURE 3.5 – Organigramme de l'algorithme CoDE.

3.3.3 Organigramme de l'Algorithme EPSDE

La figure (3.6) montre l'organigramme de l'algorithme EPSDE avec ses entrées et ses sorties.

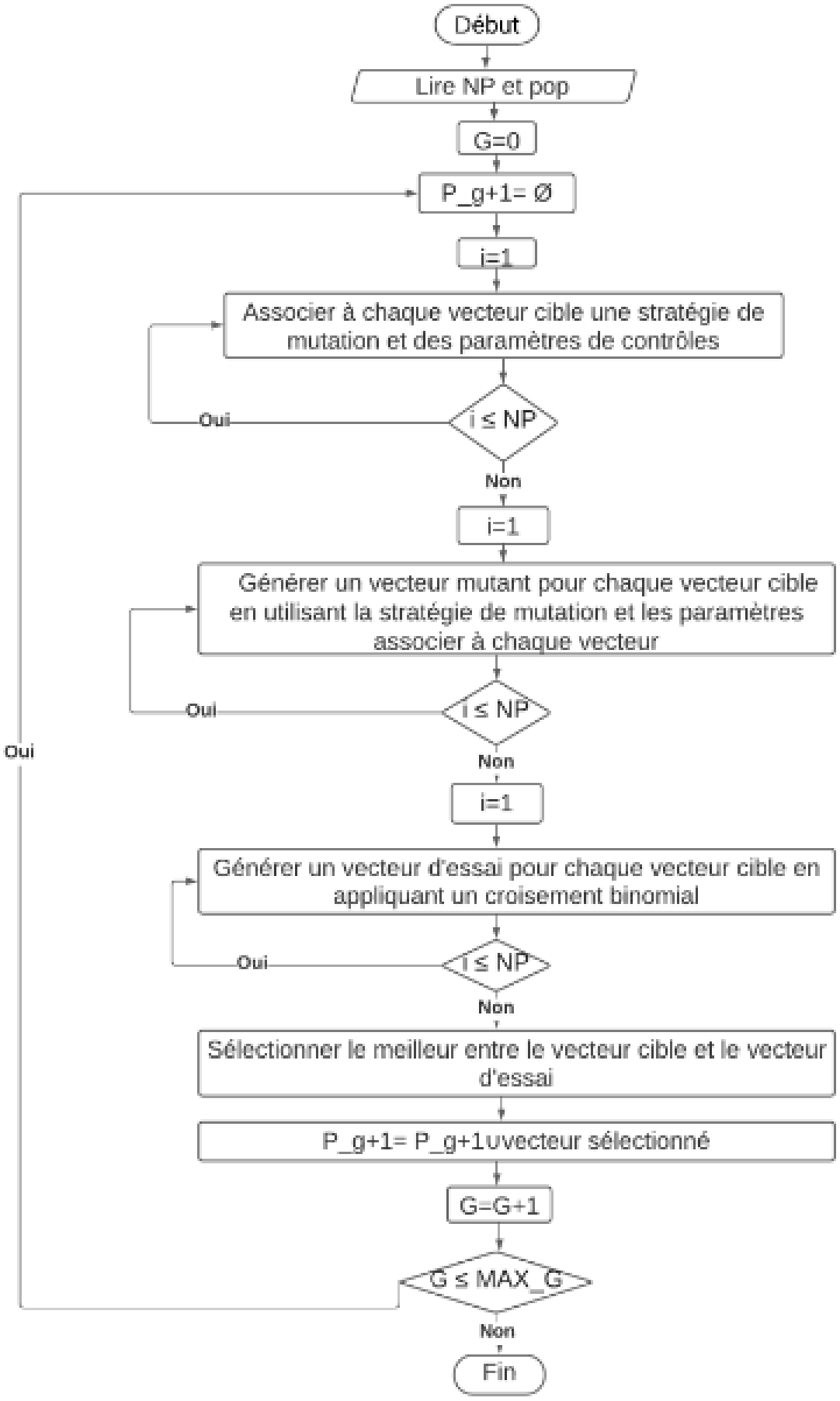


FIGURE 3.6 – Organigramme de l'algorithme EPSDE.

3.4 Version Séquentielle d'EDEV

L'implémentation de la version séquentielle d'EDEV est faite tel qu'il est décrit dans le papier [23]. EDEV est composé de trois algorithmes JADE [24], CoDE [22] et EPSDE [17] même ces trois algorithmes sont implémenté comme ils sont décrit dans leurs papiers originaux.
L'algorithme EDEV est divisé en trois fonctions principales. L'exécution est arrêté lorsqu'un critère d'arrêt est satisfait. Dans notre cas le critère d'arrêt est le nombre de génération. Dans cette section, nous détaillerons l'implémentation d'EDEV, ses fonctions, l'implémentation des trois algorithmes (JADE, CoDE et EPSDE).

3.5 Analyse

Les algorithmes à évolution différentielle visent à trouver la solution optimale ou quasi optimale en utilisant plusieurs stratégies, pour équilibrer la diversité de la population et garantir la convergence vers cette solution.
EDEV est un algorithme à évolution différentielle proposé en 2017 décrit dans [23]. Il permet d'obtenir d'excellents résultats dans la résolution des problèmes d'optimisation complexes. EDEV est composé de trois variantes puissantes de DE (JADE, CoDE et EPSDE), chaque variante à ces propres caractéristiques et utilise des différentes stratégies de reproduction. La combinaison de tous ces algorithmes et stratégies peut également rendre EDEV très robuste et performant.
EDEV se compose de trois fonctions principales et de trois algorithmes. La première fonction génère une population initiale distribuée aléatoirement dans l'espace solution. La taille de cette population égale à NP. La deuxièmement fonction divise la population entière en quatre sous-populations. Trois sous-populations indicatrices et une sous-population de récompense. La taille des trois sous population égale à $\lambda_i.NP$ tels que $\lambda_1 = \lambda_2 = \lambda_3$ et $\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1$. La dernière fonction principale applique l'algorithme JADE sur la première sous population, Code sur la deuxième et EPSDE sur la troisième. Ensuite, après chaque ng de générations, elle évalue les performances de chaque sous-population et affecte l'algorithme le plus performant à la quatrième sous-population. Enfin elle combine les quatre sous population pour obtenir une nouvelle population optimisée.

Algorithme JADE :

JADE [24] applique sur chaque individu de la première sous population une nouvelle stratégie de mutation, nommée DE/current-to-pbest/1 (3.2) avec archive pour produire un vecteur mutant. Ensuite, il applique un croisement entre le vecteur mutant et le vecteur cible, le résultat du croisement est un vecteur d'essai. En fin, il sélectionne le meilleur entre le vecteur cible et le vecteur d'essai, le vecteur sélectionné est introduit à la nouvelle génération. Le critère d'arrêt dans cet algorithme est le nombre de génération.

Algorithme CoDe :

CoDE [22] est appliqué sur la deuxième sous population. Ils génèrent pour chaque vecteur cible, trois vecteurs d'essais en utilisant trois stratégies de génération de vecteurs d'essai et trois paramètres de contrôle. Les stratégies de mutation et les paramètres de contrôle sont tiré aléatoirement de l'ensemble des stratégies et l'ensemble de paramètres de contrôles respectivement. Ensuite, il sélectionne le meilleur vecteur entre les trois vecteurs d'essais.

Si le vecteur sélectionné est meilleur que le vecteur cible, il remplace le vecteur cible par le vecteur d'essai dans la nouvelle génération. Le critère d'arrêt dans cet algorithme est le nombre Max_FES.

Algorithme EPSDE :

EPSDE affecte à chaque membre de la troisième sous population une stratégie de mutation aléatoire de l'ensemble des stratégies, et des valeurs de paramètres de contrôle tirées de l'ensemble de paramètres. Ensuite il produit un vecteur d'essai pour chaque vecteur cible, en utilisant la stratégie de mutation et les valeurs de paramètres attribuée à ce dernier. Si le vecteur cible est meilleur que le vecteur d'essai, il réinitialise le vecteur cible au hasard avec une nouvelle stratégie de mutation et des nouvelles valeurs de paramètres, à partir des ensembles respectifs ou à partir des combinaisons réussies stockées avec une probabilité égale. Si le vecteur d'essai est meilleur que le vecteur cible, la stratégie de mutation et les valeurs des paramètres associé au vecteur cible sont conservées avec le vecteur d'essai. Ce vecteur devient un vecteur cible dans la génération suivante s'il est meilleur que le vecteur optimale de la génération courante, sinon le vecteur cible ne sera pas remplacé. Le critère d'arrêt dans cet algorithme est le nombre génération.

3.5.1 Conception Globale

Après avoir expliqué EDEV dans la dernière section, l'architecture globale de l'application EDEV doit être identifiée avec l'ensemble des composants, et les relations entre eux de manière détaillée. La figure 3.7 montre l'architecture globale de EDEV.

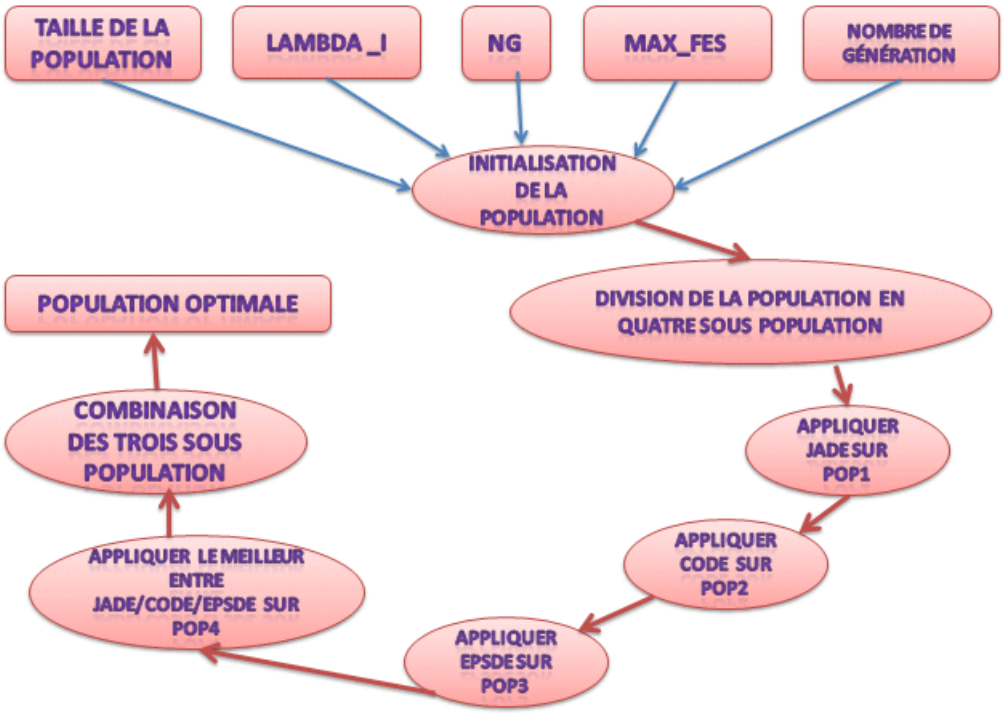


FIGURE 3.7 – Conception de la version séquentielle d'EDEV.

Dans cette partie de la conception, nous utilisons deux diagrammes UML pour expliquer les classes créées dans cette version séquentielle et également pour représenter la relation entre leurs composants à l'aide de deux diagrammes différents. La Figure 3.8 montre le diagramme de Classes de la version séquentielle d'EDEV.

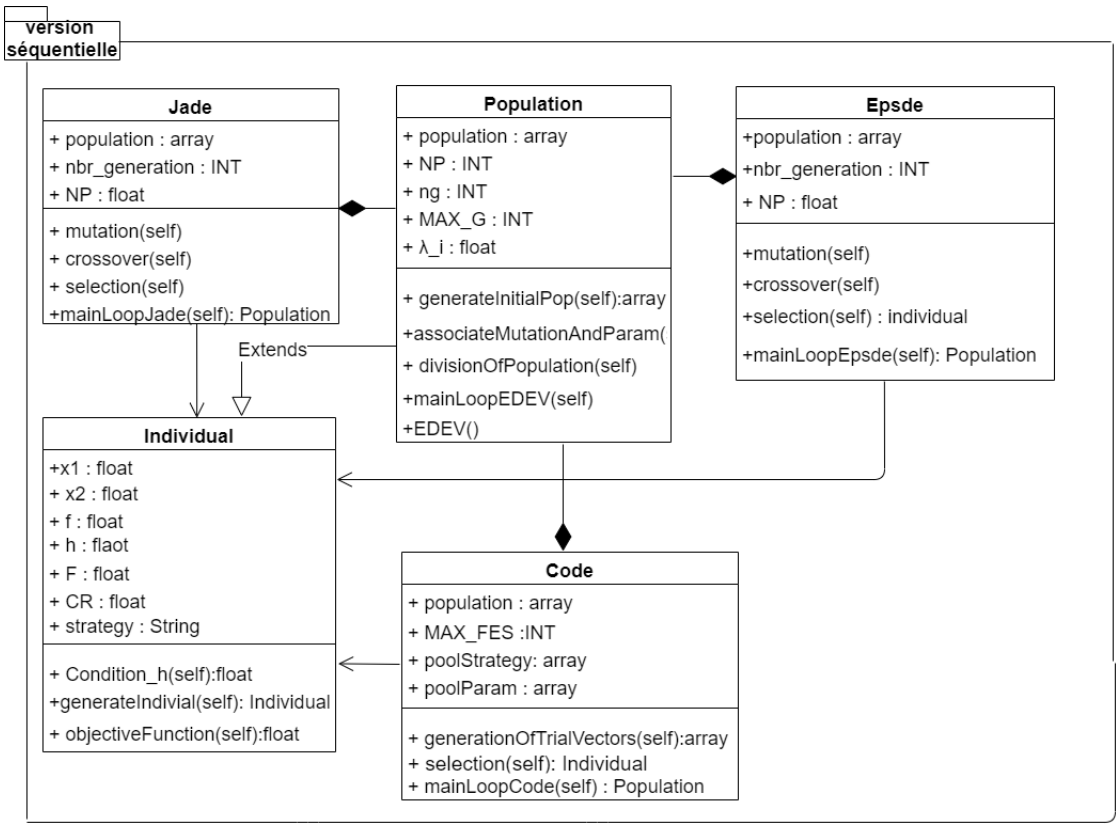


FIGURE 3.8 – Diagramme de classes de la version séquentielle.

La Figure 3.9 montre le diagramme de séquence de la version séquentielle d'EDEV.

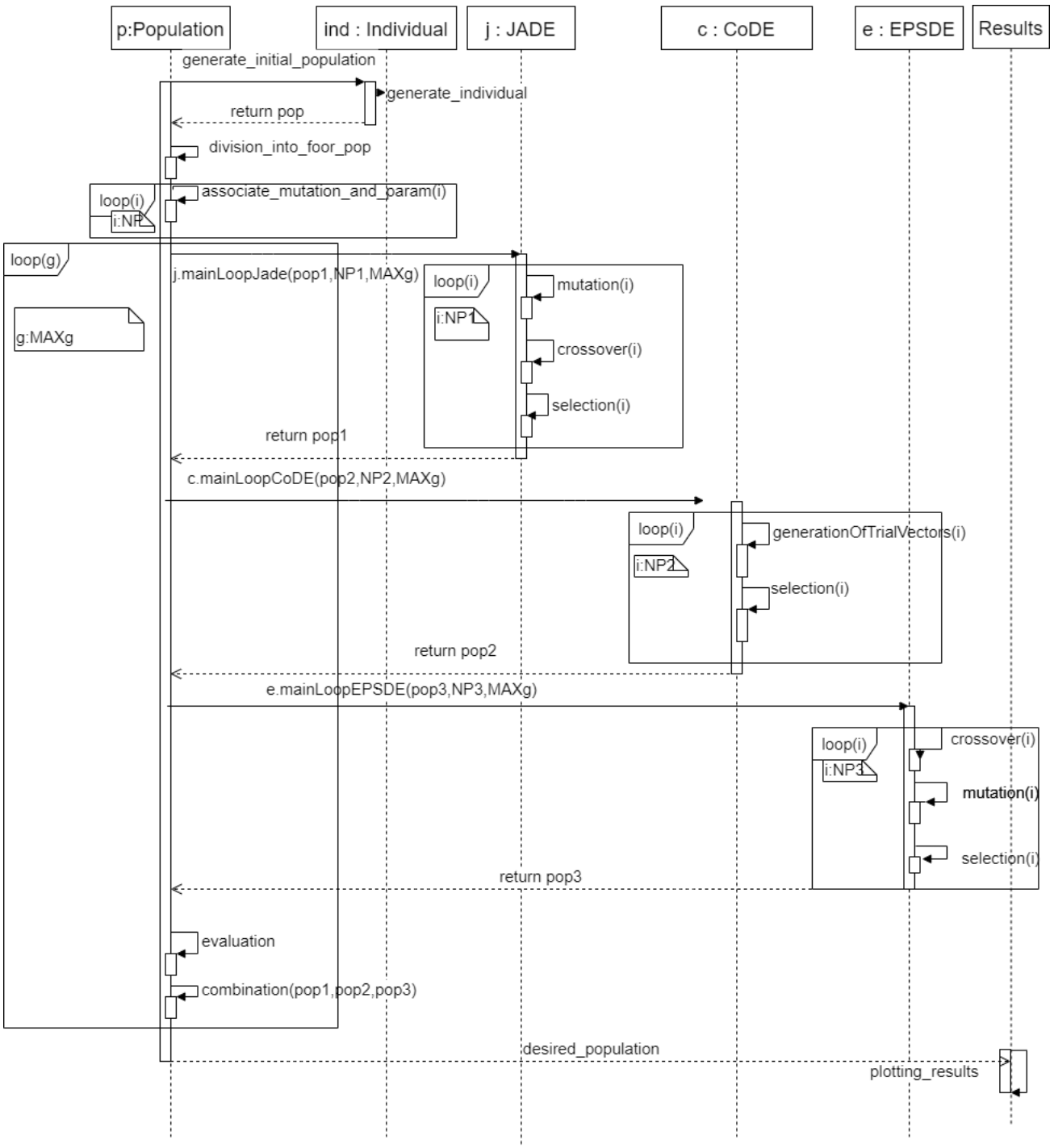


FIGURE 3.9 – Diagramme de séquence de la version séquentielle.

3.5.2 Conception Détaillée

Dans cette section, une description détaillée des fonctions sera discutée en commençant par exposer les classes et en détaillant chaque attribut et méthode dans chaque classe.

Classe Individu :
Cette classe est dédiée à la création d'un objet Individu qui est la plus petite unité de cette conception. Elle a sept attributs. Les deux premiers sont x_1 et x_2 qui représentent les gènes de l'individu. Les valeurs de x_1 et x_2 sont tirée aléatoirement entre 1 et -1. Puis il ya l'attribut f qui représente la valeur de l'évaluation de l'individu par la fonction objective. Ensuite, Individu a également les attributs F, CR et strategy. F et CR sont des paramètres

de contrôles tiré de l'ensemble des paramètres de contrôle et strategy est une stratégie de mutation tiré aléatoirement de l'ensemble des stratégies de mutation.

Generate individual
C'est la méthode génératrice de l'individu. Elle génère les variables de décision aléatoires x_1 et x_2. Puis elle initialise tous les attributs de la classe Individu.

Objective functions
Cette méthode calcule la valeur de la fonction objective de l'objet Individu. Dans notre cas, la fonction objective est une formule mathématique définie dans [16].

Classe Code :
La classe Code possède cinq attributs. Le premier c'est population qui représente la population initiale. Le deuxième c'est *NP* qui représente la taille de la population. Le troisième c'est MAX_FES qui représente le critère d'arrêt. Le troisième c'est poolParam qui représente l'ensemble des paramètres de contrôles qui sont les suivants : [F=1.0, C_r=0.1], [F=1.0, C_r=0.9] and [F=0.8, C_r=0.2] [22]. Le quatrième c'est poolStrategy qui représente l'ensemble des stratégies de mutation qui sont les suivants : rand/1/bin, rand/2/bin et current/to/rand/1 [22]. Passant maintenant aux fonctions, la classe Code possède deux fonctions generationOfTrialVectors et Selection et une méthode MainLoopCode qui invoque les deux méthodes précédentes qui sont décrites ci-dessous.

Generation of trial vectors
Cette fonction génère trois vecteurs d'essai pour chaque vecteur cible, en utilisant trois stratégies de mutation tiré de l'ensemble des stratégies de mutation, chacune avec des paramètres sélectionnés au hasard de l'ensemble des paramètres de contrôle. L'algorithme 8 montre les détails de cette fonction.

Algorithm 8 Generation of trial vectors

- select a control parameter P_1 from [[F=1.0, C_r=0.1], [F=1.0, C_r=0.9], [F=0.8, C_r=0.2]] ;
- use rand/1/bin and P_1 to generate trial vector $\vec{u}_{i_1,G}$ for the target vector $\vec{x}_{i,G}$;
- select a control parameter P_2 from [[F=1.0, C_r=0.1],
- use rand/2/bin and P_2 to generate trial vector $\vec{u}_{i_2,G}$ for the target vector $\vec{x}_{i,G}$;
- select a control parameter P_3 from [[F=1.0, C_r=0.1],
- use current/to/rand/1 and P_3 to generate trial vector $\vec{u}_{i_3,G}$ for the target vector $\vec{x}_{i,G}$;

Selection
Cette fonction sélectionne le meilleur vecteur entre les trois vecteurs d'essais généré. Si le vecteur sélectionné est meilleur que le vecteur cible, elle remplace le vecteur cible par le vecteur d'essai dans la nouvelle génération. L'algorithme 9 montre les détails de cette fonction.

Algorithm 9 Selection function of CoDE

- choose the best trial vector (denoted as $\vec{u}^*_{i,G}$) from the three trial vectors $\vec{u}_{i_1,G}, \vec{u}_{i_2,G}, and \vec{u}_{i_3,G}$; select the best between $\vec{u}^*_{i,G}$ and $\vec{x}_{i,G}$

CoDE Main loop
Cette fonction contient la boucle principale de l'algorithme, à chaque itération, elle invoque les deux méthodes citées au dessus. L'algorithme 10 montre les détails de cette fonction.

Algorithm 10 CoDE Main loop

$FES = NP$
while $FES < MAX\_FES$ **do**
  $P_{g+1} = \emptyset$
  for $i = 1 : NP$ **do**
    invoke GenerationOftrialVectors ;
    invoke Selection ;
    $P_{g+1} = P_{g+1} \cup$ ResultOfSelection ;
    $FES = FES + 3$
  $G = G + 1$

Classe Epsde :
La classe Epsde possède trois attributs. Le premier c'est population qui représente la population initiale. Le deuxième c'est NP qui représente la taille de la population. Le troisième c'est MAX_G qui représente le nombre de génération. La classe Epsde possède quatre fonctions mutation, crossover, selection et une méthode MainLoopEpsde qui invoque les trois méthodes précédentes qui sont décrites ci-dessous.

Mutation
Cette fonction génère un vecteur mutant pour chaque vecteur cible en utilisant la stratégie de mutation et les valeurs des paramètres associé à ce dernier. L'algorithme 11 montre les détails de cette fonction.

Algorithm 11 Mutation function of EPSDE

/*Generate a mutated vector $V_{i,G} = \{v^1_{i,G}, ..., v^D_{i,G}\}$ for each target vector $X_{i,G}$*/
for $i = 1 : NP$ **do**
  Generate a mutated vector $V_{i,G} = \{v^1_{i,G}, ..., v^D_{i,G}\}$ corresponding to the target vector $X_{i,G}$ using the mutation strategy and parameters associated to the target vector.

Crossover
Cette fonction génère un vecteur d'essai pour chaque vecteur cible en effectuent un croisement binomial entre le vecteur d'essai et le vecteur mutant. L'algorithme 12 montre les détails de cette fonction.

Algorithm 12 Crossover function of EPSDE

/*Generate a trial vector $U_{i,G} = \{u^1_{i,G}, ..., u^D_{i,G}\}$ for each target vector $X_{i,G}$*/
/*Binomial crossover*/
for $i = 1 : NP$ **do**
  $j_{rand} = [rand[0, 1).D]$
  for $J = 1 : D$ **do**

$$u^j_{i,G} = \begin{cases} v^j_{i,G} & \text{if } (rand_j \leq CR) \; or j = j_{rand} j = 1, 2, .., D \\ x^j_{i,G} & \text{otherwise} \end{cases} \tag{3.1}$$

Selection
Cette fonction sélectionne le meilleur vecteur entre le vecteur d'essai produit est le vecteur cible. Si le vecteur d'essai est meilleur que le vecteur cible, la stratégie de mutation et les valeurs des paramètres sont conservées avec le vecteur d'essai, et la combinaison de ces deux derniers est stockée et il devient un vecteur cible dans la génération suivante s'il est meilleur que le vecteur optimale de la génération courante. Si le vecteur cible est meilleur que le vecteur d'essai, alors le vecteur cible est réinitialisé au hasard avec une nouvelle stratégie de

mutation et des valeurs de paramètres associées à partir des ensembles respectifs ou à partir des combinaisons réussies stockées. L'algorithme 13 montre les détails de cette fonction.

Algorithm 13 Selection function of EPSDE

```
/*selection by competition between target(parent) and trial(offspring) vectors */
for i = 1 : NP do
    /*evaluate the trial vector U_{i,G}*/
    if f(U_{i,G}) ≤ f(X_{i,G}) then
        X_{i,G+1} = U_{i,G}, f(X_{i,G+1}) ≤ f(X_{i,G})
        if f(U_{i,G}) ≤ f(X_{best,G}) then
            X_{best,G} = U_{i,G}, f(X_{best,G}) ≤ f(U_{i,G})
            /*X_{best,G} is the best individual in generation G*/
        else
            X_{i,G+1} = X_{i,G}
for i = 1 : NP do
    if f(U_{i,G}) > f(X_{i,G}) then
        Randomly select a new mutation strategy and parameter values from the pools or from the stored
successful combinations
```

Classe Jade :

La classe Jade possède trois attributs. Le premier c'est population qui représente la population initiale. Le deuxième c'est NP qui représente la taille de la population. Le troisième c'est MAX_G qui représente le nombre de génération. La classe Jade possède quatre fonctions mutation, crossover et selection qui sont décrites ci-dessous.

Mutation

Cette fonction applique la stratégie de mutation "current-to-pbest/1" avec archive qui est décrite comme suit :

$$v_{i,G} = x_{i,G} + F_i(x_{pbest,G} - x_{i,G}) + F_i(x_{r1,G} - \tilde{x}_{r2,G}) \tag{3.2}$$

où $x_{i,g}$ est le vecteur cible, $x_{r1,g}$ est sélectionné aléatoirement de la population, $x^p_{best,g}$ est sélectionné aléatoirement parmi les 100% meilleurs individus de la population en cours(P) avec p $\in$ (0, 1]. Tandis que $\tilde{x}_{r2,G}$ est choisi aléatoirement de l'union $P \cup A$, de la population en cours et de l'archive. F_i est le facteur de mutation associé à X_i, tirée aléatoirement à chaque génération. Initialement, l'archive A est vide. Ensuite, après chaque génération, les solutions parentes qui échouent dans le processus de sélection sont ajoutées à l'archive. L'algorithme 14 montre les détails de cette fonction.

Algorithm 14 Mutation function of JADE

```
for i = 1 : NP do
    Generate CR_i = randn_i(μ_CR, 0.1),  F_i = randc_i(μ_F, 0.1)
    Randomly choose x^p_{best,g} as one of the 100P% best vectors
    Randomly choose x_{r1,g i,g} from current population P
    Randomly choose x̃_{r2,g} ≠ x_{r1,g} ≠ x_{i,g} from P ∪ A
    v_{i,g} = x_{i,g} + F_i.(x^p_{best,g} − x_{i,g}) + F_i.(x_{r1,g} − x̃_{r2,g})
```

Crossover

Cette fonction applique un croisement sur le vecteur mutant et le vecteur cible. L'algorithme 15 montre les détails de cette fonction.

Algorithm 15 Crossover function of JADE

for $i = 1 : NP$ **do**
  Generate j_{rand} =$randint[1, D]$
  for $j = 1 : D$ **do**
    if $j = j_{rand}$ or rand(0,1)< CR **then**
      $u_{j,i,g} = v_{j,i,g}$
    else
      $u_{j,i,g} = x_{j,i,g}$

Selection

Cette fonction sélectionne le meilleur vecteur entre le vecteur cible et le vecteur d'essai. L'algorithme 16 montre les détails de cette fonction.

Algorithm 16 Selection function of JADE

if $f(x_{i,g}) \leq f(u_{i,g})$ **then**
  $x_{i,g+1} = x_{i,g}$
else
  $x_{i,g+1} = u_{i,g}$; $x_{i,g} \longrightarrow A$; $CR_i \longrightarrow S_{CR}, F_i \longrightarrow S_F$

Jade Main loop

Cette fonction contient la boucle principale de l'algorithme, à chaque itération, elle invoque les trois fonctions citées au dessus, fait la mise à jour de l'archive (la taille de l'archive ne doit pas dépasser NP), et recalcule μ_{CR} et μ_F. L'algorithme 17 montre les détails de cette fonction.

Algorithm 17 Jade Main loop

for $g = 1 : G$ **do**
  $S_F = \emptyset$; $S_{CR} = \emptyset$;
  for $i = 1 : NP$ **do**
    invoke Mutation ;
    invoke Crossover ;
    invoke Selection
    Randomly remove solutions from A so that $|A| \leq NP$
    $\mu_{CR} = (1 - c)$. $\mu_{CR} + c$. $mean_A(S_{CR})$
    $\mu_F = (1 - c)$. $\mu_F + c$. $mean_L(S_F)$

Classe Population :

La classe Population possède cinq attributs. Dans cette section, nous discuterons chaque attribut. En fait, ces attributs seront les entrées de l'application principale plus tard, comme nous l'avons mentionné dans la figure 3.7 dans la conception globale. Le premier attribut c'est NP qui représente la taille de la population. Le deuxième c'est MAX_G qui représente le nombre de génération c'est-à-dire le nombre d'itérations de l'algorithme.. Le troisiéme c'est λ_i représente la taille des trois sous population égale à $\lambda_i.NP$ tels que $\lambda_1 = \lambda_2 = \lambda_3$ et $\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1$. Le quatrième c'est ng qui représente une période. Après ng nombre de générations, la quatrième sous-population est affectée à la variante de DE la plus performante. En fin, le cinquième c'est MAX_FES qui représente le critère d'arrêt de l'algorithme CoDE. Après avoir défini les attributs de la classe d'EDEV, nous passerons aux méthodes. Comme nous l'avons dit précédemment, cet algorithme à quatre fonctions principales qui sont GenerationOfInialPopulation, DivisionIntoFourSubpopulations, AssociateStrategyOfMutationAndParameters et MainLoopEDEV qui contient la boucle principale. Dans cette section, nous discuterons toutes ces fonctions en utilisant des pseudo codes pour expliquer l'implémentation de ces fonctions.

Generation of initial population

Cette méthode génère une population aléatoirement distribué dans l'espace de recherche. La taille de la population égale NP. La méthode lance une boucle de 1 jusqu'à NP. A chaque itération, elle invoque la classe Individu pour crée un objet individu en utilisant la méthode «Generate individual ». L'algorithme 18 montre les détails de cette fonction.

Algorithm 18 Generate initial population

for $i = 1 : NP$ **do**
/* create object individual
individual.generate_Individual
/* add individual to the population
=0

Division into four sub populations

Cette méthode divise la population entière en quatre sous-populations. Trois sous-populations indicatrices et une sous-population de récompense. Les trois sous-populations indicateurs sont désignées par *pop1*, *pop2* et *pop3* et la sous-population de récompense est représentée par *pop4*. la taille des trois sous population égale à $\lambda_i.NP$ tels que $\lambda_1 = \lambda_2 = \lambda_3$ et $\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1$. L'algorithme 19 montre les détails de cette fonction.

Algorithm 19 Division into four sub populations

Set $NP_i = \lambda_i.NP$ for i=1,2,3 ; /*size of the three subpopulations*/
Set $NP_4 = NP - (NP_i.3)$
Randomly divide the population into pop 1 , pop 2 , pop 3 and pop 4 with respect to their sizes ;

Associate strategy of mutation and parameters

Cette fonction affecte une stratégie de mutation aléatoire et des valeurs de paramètres tirées de l'ensemble des paramètres de contrôle à chaque membre de la population initiale. L'ensemble des stratégies de mutation sont : "DE/best/2/bin", "DE/rand/1/bin" et "DE/current-to-rand/1/ bin" [17]. Les valeurs de *CR* sont prise dans l'intevalle [0,1 - 0,9] par pas de 0,1. Les valeurs de F sont comprises entre 0,4 et 0,9 par pas de 0,1. [17]. L'algorithme 20 montre les détails de cette fonction.

Algorithm 20 Associate strategy of mutation and parameters

for $i = 1 : NP$ **do**
select randomly a mutation strategy from the pool of mutation strategy
select randomly *CR* and F from the pool of parameter
associate strategy and parameter to the target vector $\vec{x}_{i,G}$

Main loop of EDEV

C'est la fonction principale d'EDEV elle applique JADE, CoDE et EPSDE sur *pop1*, *pop2* et *pop3* respectivement. Premièrement, elle criée une instance de l'objet JADE en lui passant sur *pop1* et NP_1. Ensuite, elle invoque la méthode MainLoopJADE a partir de l'objet JADE, cette méthode fait la mise à jour de *pop1* et calcule f_1. Deuxièmement, elle criée une instance de l'objet CoDE en lui passant sur *pop2* et NP_2. Ensuite, elle invoque la méthode MainLoopCoDE a partir de l'objet CoDE, cette méthode fait la mise à jour de *pop2* et calcule f_2. Troisièmement, elle criée une instance de l'objet EPSDE en lui passant sur *pop3* et NP_3. Ensuite, elle invoque la méthode MainLoopEPSDE a partir de l'objet EPSDE, cette méthode fait la mise à jour de *pop3* et calcule f_3. Elle combine les quatre nouvelles sous population dans pop. Après ng nombre de générations, elle fait l'évaluation des trois sous

population. Ensuite, elle partitionne aléatoirement pop en *pop1*,*pop2*, *pop3* et *pop4* et applique le meilleur algorithme sur *pop4*. Ce processus est répéter à chaque génération et il s'arête lorsque le nombre de génération atteint MAX_G. L'algorithme 21 montre les détails de cette fonction.

Algorithm 21 Main loop of EDEV

while $g \leq MaxG$ **do**
- 10 : $g = g + 1$;
- 11 : Execute JADE on pop_1 , update pop_1 and calculate Δf_1;
- 12 : Execute CoDE on pop_2 , update pop_2 and calculate Δf_2;
- 13 : Execute EPSDE on pop_3 , update pop_3 and calculate Δf_3;
- 14 : Combine updated pop_1, pop_2 and pop_3 into $pop, i.e., pop = \bigcup_{i=1,2,3} pop_i$;
- **if** $mod(g, ng) == 0$ **then**
  - 16 : $K = arg(max_{i=1,2,3}(\frac{\Delta f_i}{ng.NP_i}))$;
- 18 : Randomly partition pop into pop_1, pop_2, pop_3 and pop_4 ;
- 19 : Let $pop_k = pop_k \bigcup pop_4, K \in$ 1,2,3 ;

3.6 Version Parallèle d'EDEV

3.6.1 Analyse

PEDEV est un algorithme à évolution différentielle proposé dans cette thèse pour obtenir des résultats satisfaisantes avec moins de temps d'exécution. Cet algorithme a les mêmes fonctions principales qu'EDEV mais la seule différence est que PEDEV attribue aux trois processus les sous population pop1, pop2 et pop3. Le premier processus applique l'algorithme JADE sur pop1, le deuxième processus applique l'algorithme CoDE sur pop2 et le troisième processus applique l'algorithme EPSDE sur pop3.

PEDEV utilise le principe "maître / esclave" [1] où le processus principal est le maître et les n-processus sont les esclaves.

3.6.2 Conception Globale

La figure 3.10 montre l'architecture globale de PEDEV.

1. Le maître / esclave est un modèle de communication dans lequel un appareil ou un processus a un contrôle unidirectionnel est le maître, les autres appareils jouent le rôle d'esclaves. Le maître donne des ordres aux esclaves qui les exécute.[5].

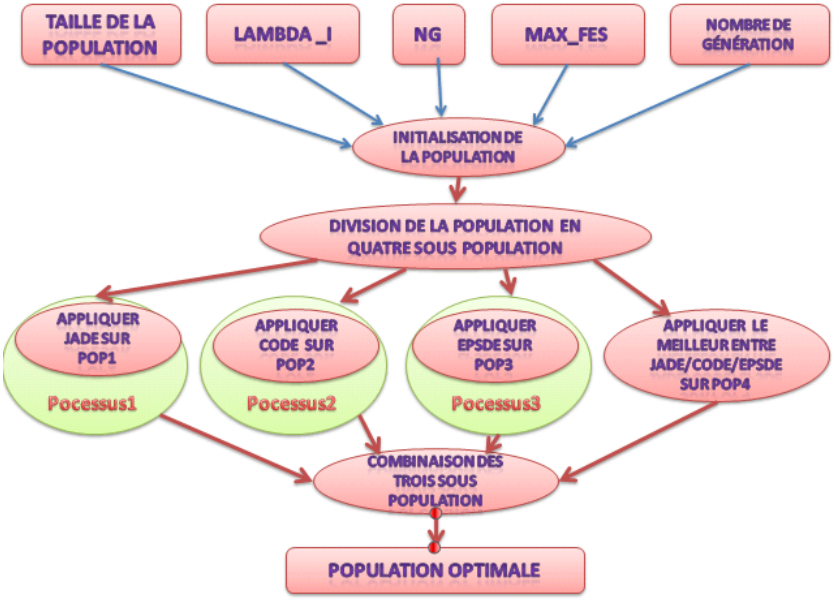


FIGURE 3.10 – Conception de la version parallèle.

Dans cette partie, nous utilisons deux diagrammes UML pour expliquer les classes créées dans cette version parallèle et également pour représenter la relation entre leurs composants.

La figure 3.11 montre le diagramme de classes de cette version parallèle.

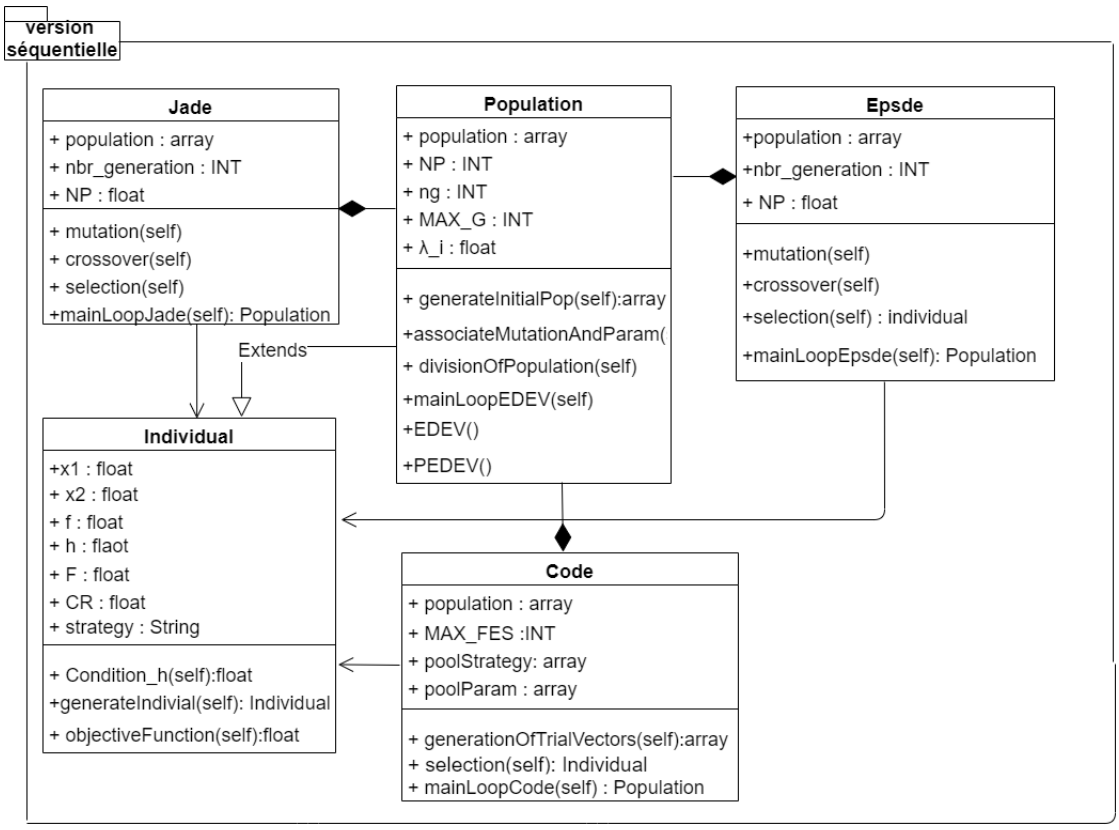


FIGURE 3.11 – Diagramme de classe de la version parallèle.

La figure 3.12 montre le diagramme de séquence de cette version parallèle.

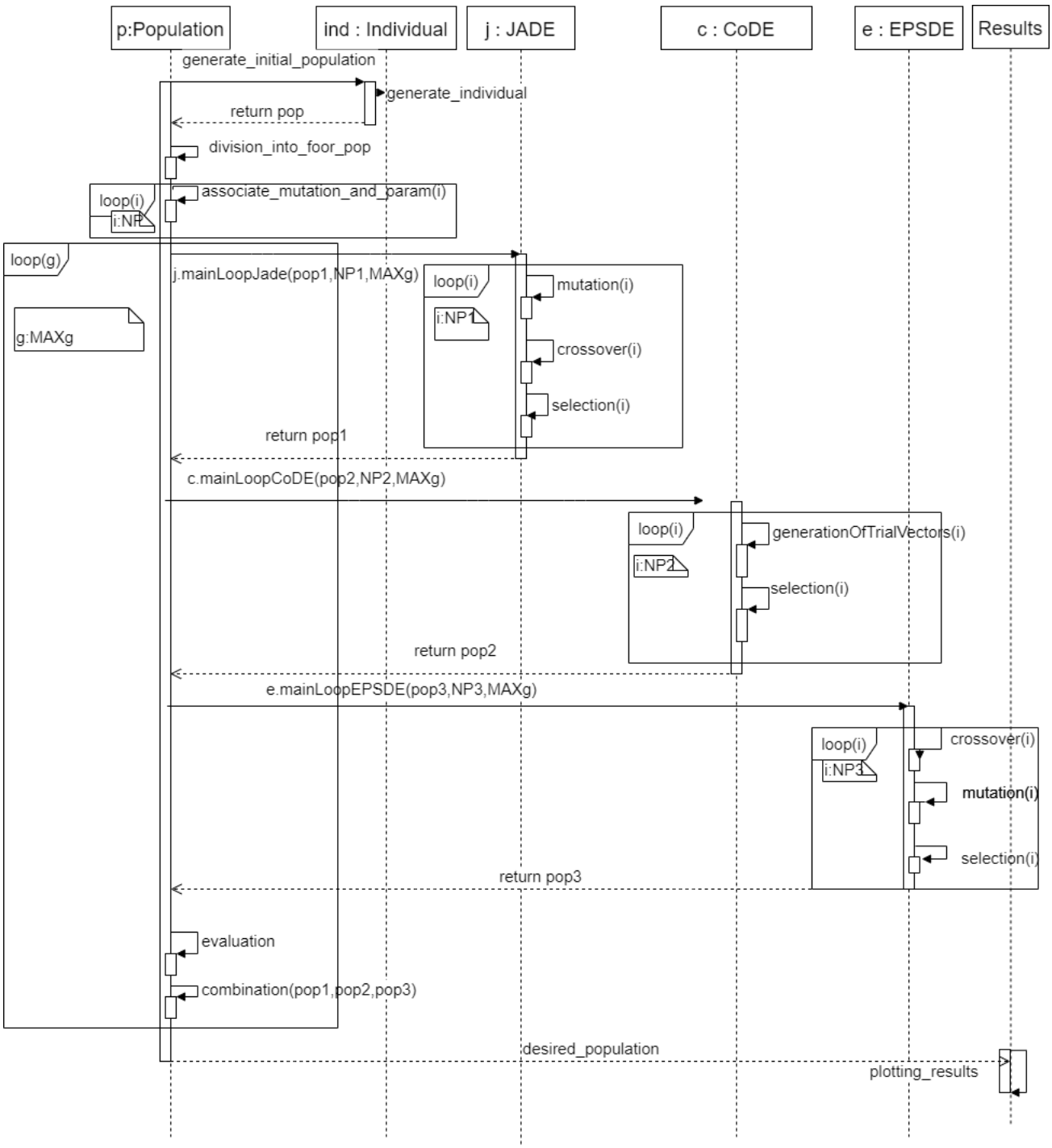


FIGURE 3.12 – Diagramme de séquence de la version parallèle.

3.6.3 Conception Détaillée

Dans la version parallèle, une nouvelle méthode est ajoutée qui est PEDEV les autres méthodes sont similaires. La méthode PEDEV attribue aux trois processus les sous population pop1, pop2 et pop3. Le premier processus applique l'algorithme JADE sur pop1, le deuxième processus applique l'algorithme CoDE sur pop2 et le troisième processus applique l'algorithme EPSDE sur pop3. Le pseudo code de la méthode PEDEV est décrit dans l'algorithme 22 sans ré-expliquer les autres méthodes.

Algorithm 22 PEDEV

while $g \leq MaxG$ **do**
$g = g + 1$;
q1 = mp.Queue()
p1 = mp.Process
p1 execute JADE on pop_1 , update pop_1 and calculate Δf_1;
p1.start()
q1.get()
p1.join()
q2 = mp.Queue()
p2 = mp.Process
p2 execute CoDE on pop_2 , update pop_2 and calculate Δf_2;
p2.start()
q2.get()
p2.join()
q3 = mp.Queue()
p3 = mp.Process
p3 execute EPSDE on pop_3 , update pop_3 and calculate Δf_3;
p3.start()
q3.get()
p3.join()
Combine updated pop_1, pop_2 and pop_3 into $pop, i.e., pop = \bigcup_{i=1,2,3} pop_i$;
if $mod(g, ng) == 0$ **then**
16 : $K = arg(max_{i=1,2,3}(\frac{\Delta f_i}{ng.NP_i}))$;
18 : Randomly partition pop into pop_1, pop_2, pop_3 and pop_4 ;
19 : Let $pop_k = pop_k \bigcup pop_4, K \in 1,2,3$;

Conclusion

Dans ce chapitre, nous avons présenté notre contributions. Ce chapitre est divisé en cinq sections. Les deux dernières sections sont considérées comme les sections principales, où nous présentons dans chaque section d'entre elles la conception et l'analyse concernant les deux versions (c'est-à-dire la version séquentielle et la version parallèle) en utilisant des diagrammes UML.

Le chapitre suivant est dédié à l'implémentation de l'ensemble de l'application qui regroupe les deux versions (c'est-à-dire EDEV et PEDEV).

Chapitre 4

Implémentation et Étude Expérimentale

Introduction

Après l'analyse et la conception de chaque version (c'est-à-dire séquentielle et parallèle) avec une description détaillée pour tout le cycle du projet. La prochaine étape est la mise en œuvre d'EDEV et PEDEV. Ce chapitre vise à coder les deux versions (c'est-à-dire, EDEV et PEDEV) et a implémenter une application globale regroupe les deux versions, évaluer l'application et comparer les résultats de ces deux algorithmes pour conclure la meilleure version.

Ce chapitre est divisé en quatre parties. La première section présente brièvement les outils de développement et les langages que nous avons exploités dans la réalisation de notre projet. La deuxième section décrit les principaux résultats de mise en œuvre de notre application finale. La troisième section est dédiée à l'évaluation des deux versions (c'est-à-dire, EDEV et PEDEV) en utilisant un problème d'optimisation définit dans l'article [16]. La dernière section présente une étude comparative pour prouver l'efficacité du PEDEV contre EDEV.

4.1 Langages Et Outils de Développement

Dans cette section, nous présentons différents outils et langages, qui nous ont aidés lors de la réalisation de notre projet au niveau de programmation et au niveau théorique.

4.1.1 Langage de Programmation Python

Python est un langage de programmation intelligent que nous avons utilisé dans la mise en œuvre de notre application. Il est flexible et sa syntaxe est facile à apprendre. Un programme Python est court par rapport aux programmes des autres langages, en raison de la disponibilité de nombreuses fonctions implémentées. Python est un langage de programmation open source et non typé. Il est disponible pour tous les systèmes d'exploitation (Windows, LINUX, Mac OS).

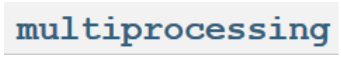
4.1.2 Éditeur PyCharm

PyCharm est un environnement de développement intégré (IDE) open source, utilisé pour la programmation python. C'est un assistant de codage puissant, il peut mettre en évidence les erreurs et introduit des corrections rapides basés sur un débogueur Python intégré. C'est un éditeur approprié pour écrire et tester de nombreuses lignes de code et de classes, car il offre une vue structurelle du projet et une navigation rapide dans les fichiers.

4.1.3 Bibliothèque "matplotlib"

matplotlib [1] est une bibliothèque Python capable de produire des graphes de qualité. matplotlib essai de rendre les tâches simples et de rendre possible les choses compliqués. Vous pouvez générer des graphes, histogrammes, des spectres de puissance (lié à la transformée de Fourier), des graphiques à barres, des graphiques d'erreur, des nuages de dispersion, etc... en quelques lignes de code. Puisque dans notre projet nous avons beaucoup de résultats qui doivent être tracés, nous avons donc choisi *matplotlib* pour les tracer.

4.1.4 Interface de "threading" Basée sur les Processus



multiprocessing [3] est un paquet qui permet l'instanciation de processus via la même API que le module threading. Le paquet multiprocessing offre à la fois des possibilités de programmation concurrente locale ou à distance, contournant les problèmes du Global Interpreter Lock en utilisant des processus plutôt que des fils d'exécution. Ainsi, le module multiprocessing permet au développeur de bénéficier entièrement des multiples processeurs sur une machine. Il tourne à la fois sur les systèmes Unix et Windows.[3].

4.1.5 Système de Composition de Documents LaTeX

LaTeX [15] est un système de composition puissant et flexible pour la production d'articles techniques et scientifiques de haute qualité. Il est basé sur la langue des balises. Il suit la philosophie de conception de séparer la présentation du contenu, ainsi les auteurs se concentrent sur ce qu'ils écrivent, pas sur ce qui est affiché, car l'apparence est gérée par LaTeX . L'apparence comprend de nombreux aspects, la structure du document (partie, chapitre, section, ..etc), des figures, des références et des bibliographies. Il est plus familier à un programmeur informatique, car il suit le cycle code-compilation-exécution.

4.1.6 Application de Création de Diagrammes "Draw.io"

draw.io [2] est une application de création de diagrammes entièrement gratuite et autonome conçue par les leaders technologiques de la création de diagrammes Web. Aucune inscription, aucune limitation.

4.2 Implémentation

Les algorithmes étudiés sont implémentés en utilisant le paradigme de programmation orientée objet (POO) et un ensemble de logiciels et de matériel qui sont résumés dans le

tableau 4.1.

Logiciel/Matériel	Version
OS	Microsoft Windows 10 Professional, 64bits, version 10.0.17134
CPU	Intel(R) Core(TM) i3-5005U CPU @2.00GHz 2.00GHz
RAM	4.00Go
Interpréteur Python	3.7.0
PyCharm	2016.3.1
matplotlib	2.0.0

TABLE 4.1 – Versions Logiciel/Matériel

4.3 Test et Étude Expérimentale

Nous avons divisé ce test et cette étude expérimentale en deux parties. La première partie est consacrée à la discussion des résultats de la version séquentielle (EDEV) en utilisant un problème d'optimisation mono-objectif fournis dans l'article [16] et elle se concentre sur le test de la crédibilité des résultats de l'application. La deuxième partie se concentre sur la preuve de l'efficacité de la version parallèle (PEDEV).

Nous allons présenter les résultats expérimentaux de l'application, l'objectif ici est d'étudier l'efficacité d'EDEV, JADE, Code et EPSDE en utilisant une fonction de test proposée dans [16]. Pour la fonction de test donnée $f(\vec{x}) = x_1^2+(x_2-1)^2$, la solution optimale calculée dans [16] est $\vec{x}^* = (0.707036070037170616, 0.500000004333606807)$ et $f(\vec{x}^*) = 0.7499$ tels que $-1 \leq x_1 \leq 1$, $-1 \leq x_2 \leq 1$
Pour tester la fiabilité d'EDEV on a choisi différentes valeurs des paramètres d'entrée pour chaque algorithme (EDEV, JADE, CoDE et EPSDE).

4.3.1 Résultats Expérimentaux d'EDEV (version séquentielle)

Variation de la Taille de la Population Initiale

On a choisi cinq tailles de population initiale : 100, 200, 300, 400 et 500 pour tester EDEV et on a fixé les autres paramètres d'entrée.
La figure 4.1 montre les résultats lorsque la taille initiale de la population est 100.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

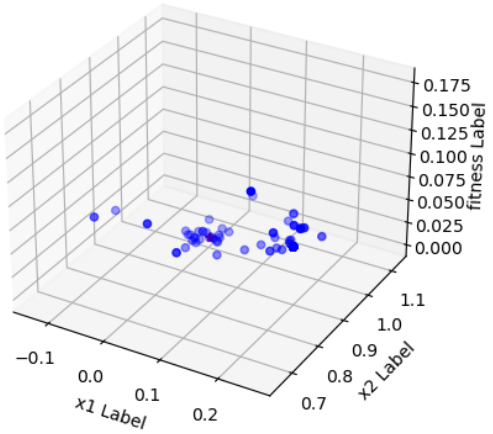


FIGURE 4.1 – Résultats avec une taille de la population initiale = 100

La figure 4.2 montre les résultats lorsque la taille initiale de la population est 200.

- Taille population initiale = 200,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

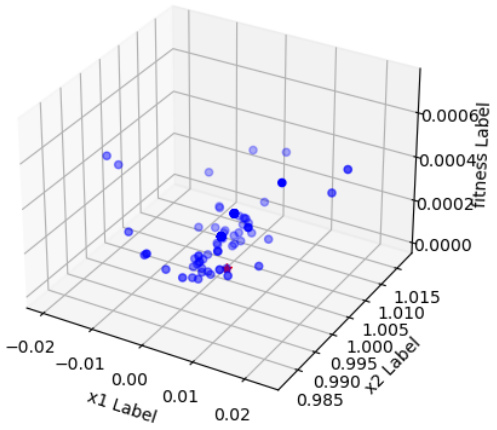


FIGURE 4.2 – Résultats avec une taille de la population initiale = 200

La figure 4.3 montre les résultats lorsque la taille initiale de la population est 300.

- Taille population initiale = 300,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

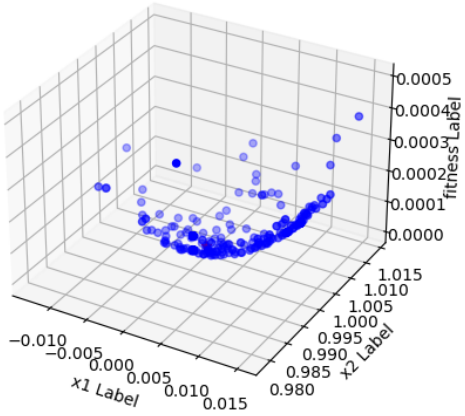


FIGURE 4.3 – Résultats avec une taille de la population initiale = 300

La figure 4.4 montre les résultats lorsque la taille initiale de la population est 400.

- Taille population initiale = 400,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

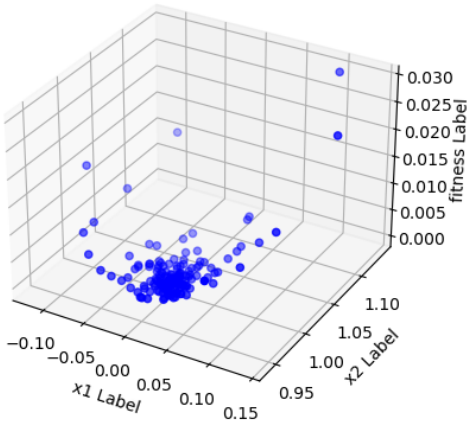


FIGURE 4.4 – Résultats avec une taille de la population initiale = 400

La figure 4.5 montre les résultats lorsque la taille initiale de la population est 500.

- Taille population initiale = 500,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

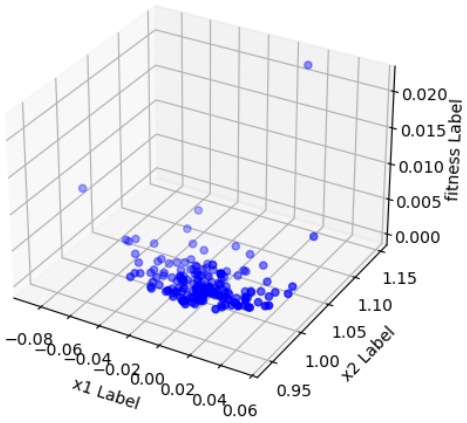


FIGURE 4.5 – Résultats avec une taille de la population initiale = 500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque taille de population.
Le tableau 4.2 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

NP	x_1	x_2	f	$\bar{f}$	σ
100	0.003316	0.995902	2.779*10^{-}5	1.30786	2.339182
200	0.00201	1.000004	4.073*10^{-}6	2.27949	2.93488
300	9.682*10^{-}5	0.99945	3.09*10^{-}7	1.16169*10^{-}6	2.1208*10^{-}6
400	0.0004866	0.99917	9.243*10^{-}7	1.254*10^{-}8	2.354*10^{-}8
500	-0.00149	1.000578	2.574*10^{-}7	2.589*10^{-}9	2.897*10^{-}9

TABLE 4.2 – Résultat expérimentaux de la variation de la taille de population.

En examinant les résultats sur le tableau 4.2, on remarque que la taille de la population a un effet sur le processus d'optimisation.

Variation du Nombre de Génération

On a varié le nombre de génération entre 10 et 50 par pas de 10 pour tester EDEV et on a fixé les autres paramètres d'entrée.
La figure 4.6 montre les résultats lorsque le nombre de génération égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

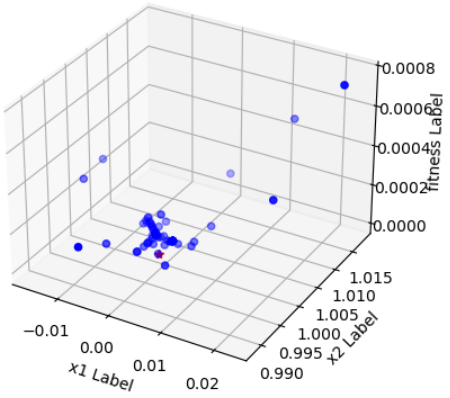


FIGURE 4.6 – Résultats avec nombre de génération = 10

La figure 4.7 montre les résultats lorsque le nombre de génération égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 20,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

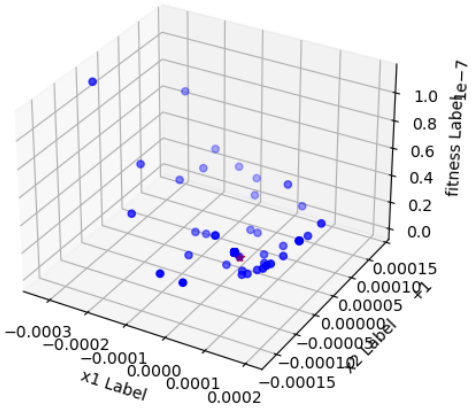


FIGURE 4.7 – Résultats avec nombre de génération = 20

La figure 4.8 montre les résultats lorsque le nombre de génération égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 30,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

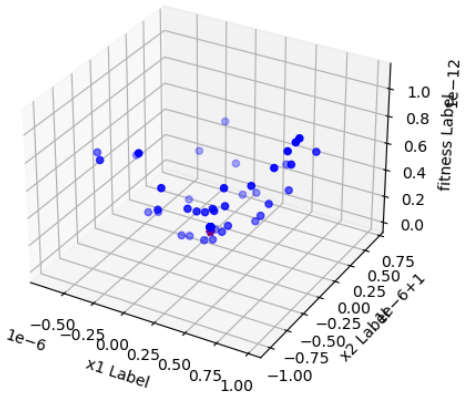


FIGURE 4.8 – Résultats avec nombre de génération = 30

La figure 4.6 montre les résultats lorsque le nombre de génération égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 40,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

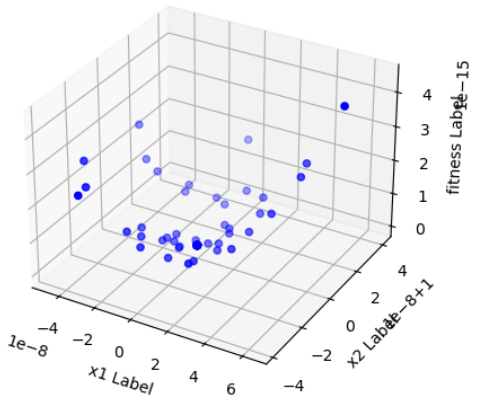


FIGURE 4.9 – Résultats avec nombre de génération = 40

La figure 4.6 montre les résultats lorsque le nombre de génération égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 50,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

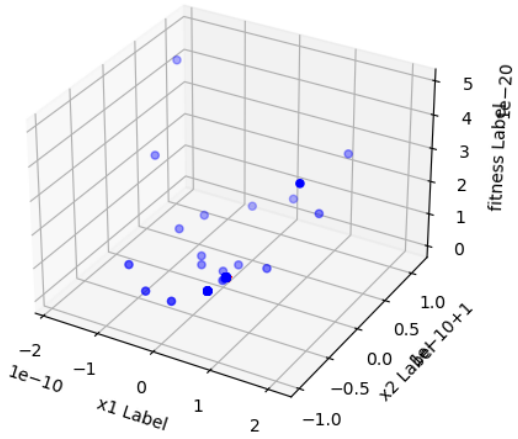


FIGURE 4.10 – Résultats avec nombre de génération = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de génération.
Le tableau 4.3 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

MAX_G	x_1	x_2	f	$\bar{f}$	σ
10	-0.001604	0.9989	3.764*10⁻6	1.30786	2.33918
20	-3.629*10⁻7	0.9999	2.177*10⁻13	1.303*10⁻9	2.484*10⁻9
30	5.345*10⁻8	1	3.116*10⁻15	1.634*10⁻13	3.192*10⁻13
40	-1.558*10⁻9	1	4.151*10⁻18	2.212*10⁻16	8.545*10⁻16
50	6.604*10⁻12	1	4.512*10⁻23	1.456*10⁻18	9.278*10⁻18

TABLE 4.3 – Résultat expérimentaux de la variation du nombre de génération.

En examinant les résultats sur le tableau 4.3, on remarque que le nombre de génération a un effet sur le processus d'optimisation.

Variation du Nombre ng

On a varié le nombre ng entre 10 et 50 par pas de 10 pour tester EDEV et on a fixé les autres paramètres d'entrée.
La figure 4.11 montre les résultats lorsque ng égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

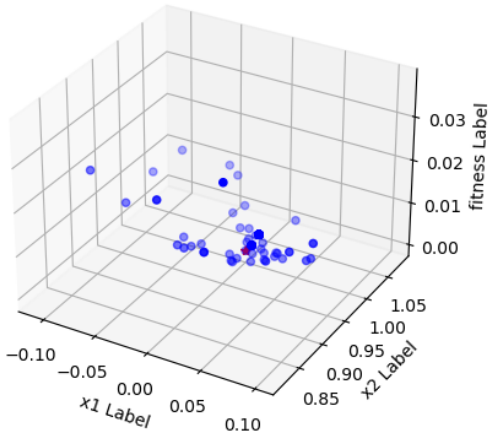


FIGURE 4.11 – Résultats avec ng = 10

La figure 4.12 montre les résultats lorsque ng égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 20,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

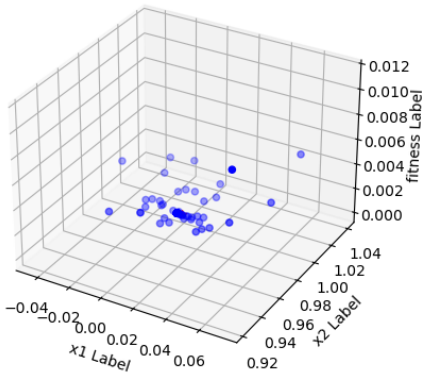


FIGURE 4.12 – Résultats avec ng = 20

La figure 4.13 montre les résultats lorsque ng égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 30,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

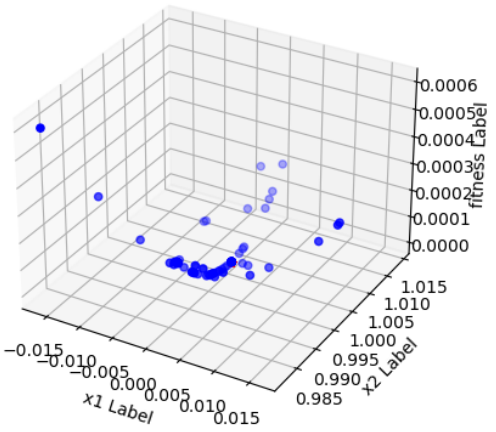


FIGURE 4.13 – Résultats avec ng = 30

La figure 4.14 montre les résultats lorsque ng égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 40,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

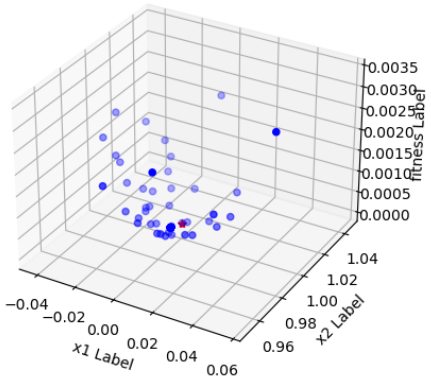


FIGURE 4.14 – Résultats avec ng = 40

La figure 4.15 montre les résultats lorsque ng égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 50,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

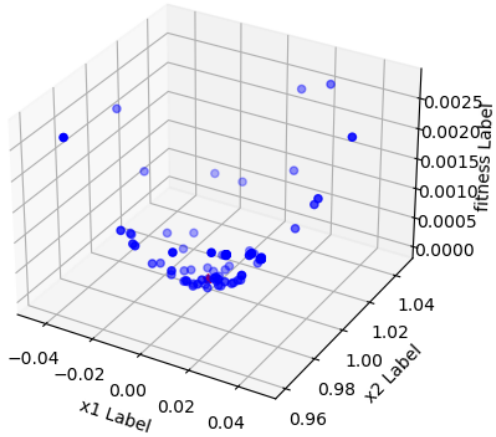


FIGURE 4.15 – Résultats avec ng = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre ng.
Le tableau 4.4 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

ng	x_1	x_2	f	$\bar{f}$	σ
10	-0.000187	1.000325	1.412*10$^-$7	0000	00000
20	-0.000441	0.99823	3.324*10$^-$6	0000	00000
30	0.000396	1.000169	1.856*10$^-$7	00000	00000
40	-0.001935	0.00424	2.176*10$^-$5	00000	00000
50	-0.000434	0.99876	1.707*10$^-$6	0000	00000

TABLE 4.4 – Résultat expérimentaux de la variation du nombre ng.

En examinant les résultats sur le tableau 4.4, on remarque que le nombre ng n'a pas un effet sur le processus d'optimisation.

4.3.2 Résultats Expérimentaux de JADE

Variation de la Taille de la Population Initiale

On a choisi cinq tailles de population initiale : 100, 200, 300, 400 et 500 pour tester JADE et on a fixé les autres paramètres d'entrée.
La figure 4.16 montre les résultats lorsque la taille initiale de la population est 100.

- Taille population initiale = 100,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

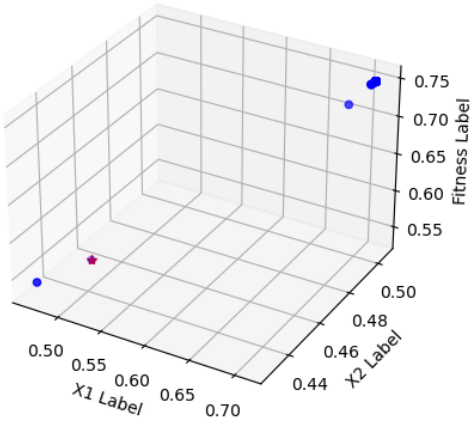


FIGURE 4.16 – Résultats avec une taille de la population initiale = 100

La figure 4.17 montre les résultats lorsque la taille initiale de la population est 200.

- Taille population initiale = 200,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

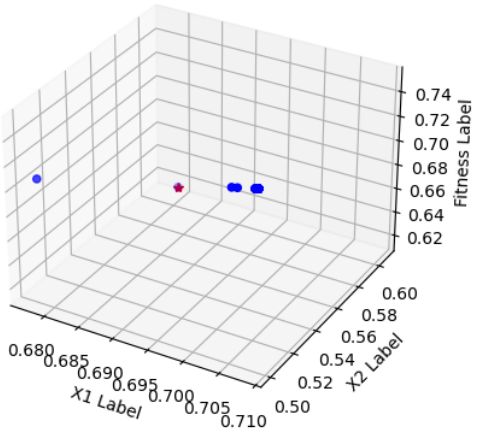


FIGURE 4.17 – Résultats avec une taille de la population initiale = 200

La figure 4.18 montre les résultats lorsque la taille initiale de la population est 300.

- Taille population initiale = 100,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

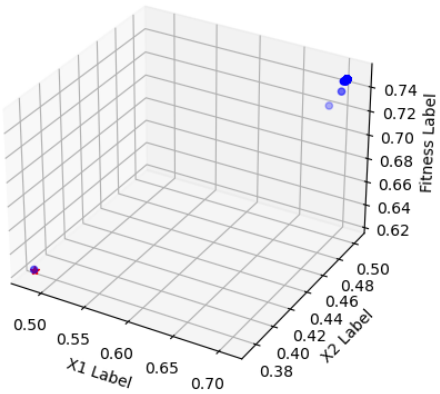


FIGURE 4.18 – Résultats avec une taille de la population initiale = 300

La figure 4.19 montre les résultats lorsque la taille initiale de la population est 400.

- Taille population initiale = 400,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

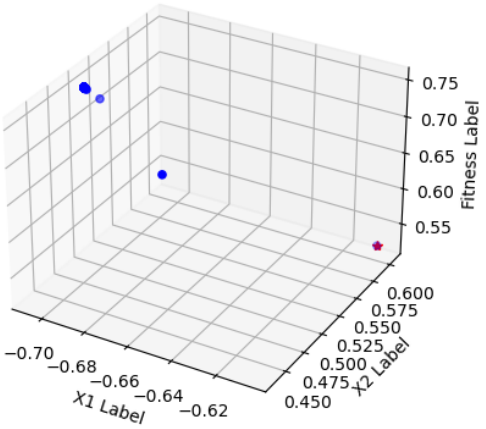


FIGURE 4.19 – Résultats avec une taille de la population initiale = 400

La figure 4.20 montre les résultats lorsque la taille initiale de la population est 500.

- Taille population initiale = 500,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

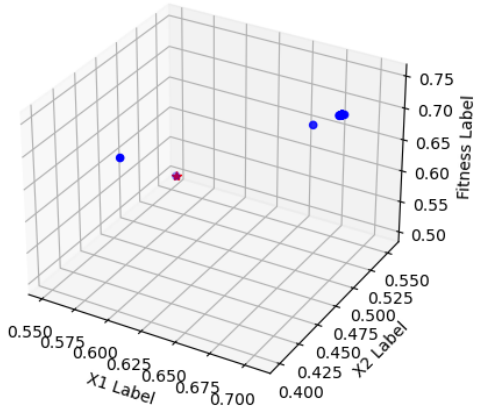


FIGURE 4.20 – Résultats avec une taille de la population initiale = 500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque taille de population.
Le tableau 4.5 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

NP	x_1	x_2	f	$\bar{f}$	σ
100	0.48309	0.450579	0.53524	0.167567	0.12492
200	0.679552	0.606894	0.61632	0.176524	0.18893
300	0.481526	0.373625	0.62421	0.165515	0.14413
400	-0.605573	0.604132	0.52343	0.79943	0.159351
500	0.551039	0.555153	0.501533	0.095549	0.13654

TABLE 4.5 – Résultat expérimentaux de la variation de la taille de population.

En examinant les résultats sur le tableau 4.5, on remarque que la taille de population n'a pas un effet sur le processus d'optimisation.

Variation du Nombre de Génération

On a varié le nombre de génération entre 10 et 50 par pas de 10 pour tester JADE et on a fixé les autres paramètres d'entrée.
La figure 4.21=10 montre les résultats lorsque le nombre de génération égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

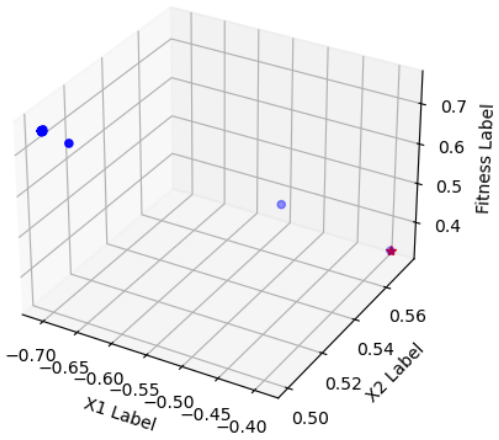


FIGURE 4.21 – Résultats avec nombre de génération = 10

La figure 4.22 montre les résultats lorsque le nombre de génération égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 20,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

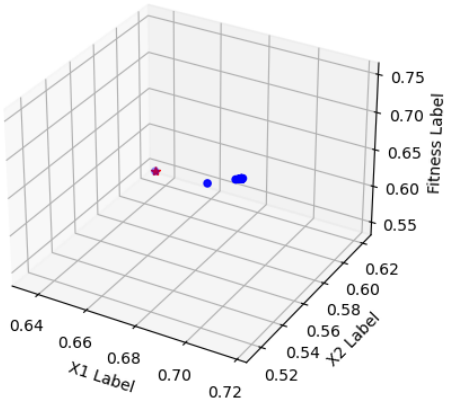


FIGURE 4.22 – Résultats avec nombre de génération = 20

La figure 4.23 montre les résultats lorsque le nombre de génération égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 30,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

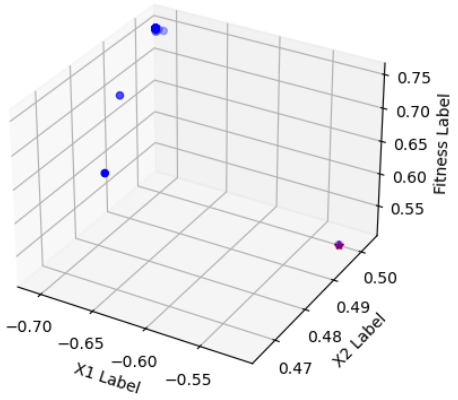


FIGURE 4.23 – Résultats avec nombre de génération = 30

La figure 4.24 montre les résultats lorsque le nombre de génération égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 40,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

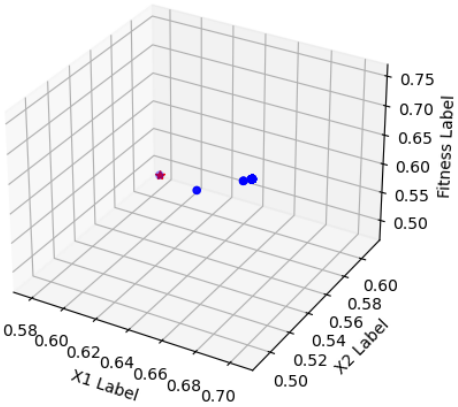


FIGURE 4.24 – Résultats avec nombre de génération = 40

La figure 4.25 montre les résultats lorsque le nombre de génération égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

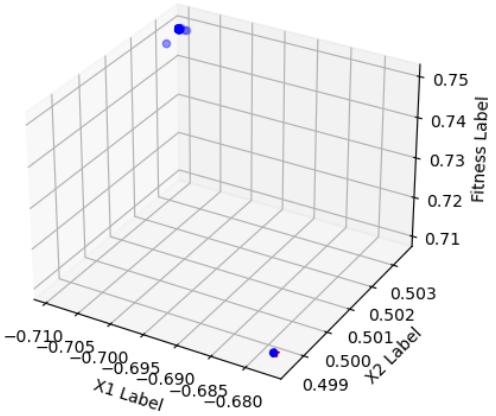


FIGURE 4.25 – Résultats avec nombre de génération = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de génération.
Le tableau 4.6 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

MAX_G	x_1	x_2	f	$\bar{f}$	σ
10	-0.390991	0.573219	0.335016	0.167567	0.12492
20	0.635025	0.620917	0.54696	0.119828	0.150071
30	-0.51681	0.497149	0.519951	0.205406	0.143302
40	0.57774	0.612552	0.4839	0.29745	0.02153
50	-0.677268	0.49843	0.71026	0.20796	0.18712

TABLE 4.6 – Résultat expérimentaux de la variation du nombre de génération.

En examinant les résultats sur le tableau 4.6, on remarque que le nombre de génération n'a pas un effet sur le processus d'optimisation.

Variation du Taux de Croisement

On a varié le taux de croisement μ_{CR} entre 0.5 et 4 pour tester JADE et on a fixé les autres paramètres d'entrée.

La figure 4.26 montre les résultats lorsque μ_{CR} égale à 0.5.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

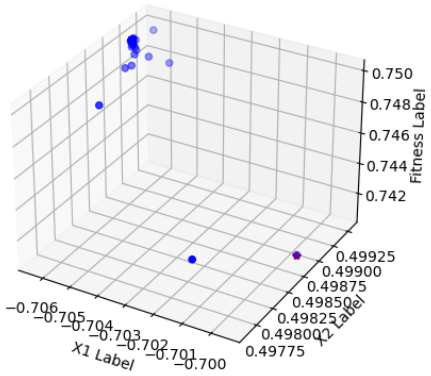


FIGURE 4.26 – Résultats avec $\mu_{CR} = 0.5$

La figure 4.27 montre les résultats lorsque μ_{CR} égale à 1.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 1$,
- $\mu_F = 0.5$.

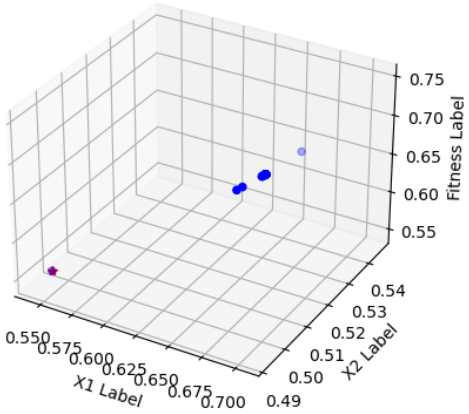


FIGURE 4.27 – Résultats avec $\mu_{CR} = 1$

La figure 4.28 montre les résultats lorsque μ_{CR} égale à 2.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 2$,
- $\mu_F = 0.5$.

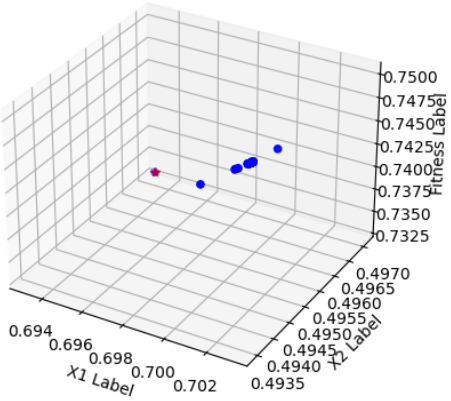


FIGURE 4.28 – Résultats avec $\mu_{CR} = 2$

La figure 4.29 montre les résultats lorsque μ_{CR} égale à 3.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 3$,
- $\mu_F = 0.5$.

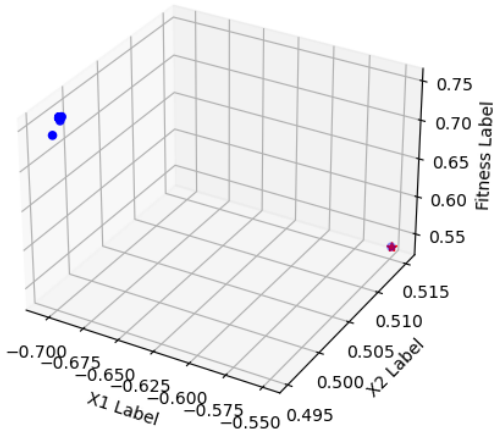


FIGURE 4.29 – Résultats avec $\mu_{CR} = 3$

La figure 4.30 montre les résultats lorsque μ_{CR} égale à 4.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 4$,
- $\mu_F = 0.5$.

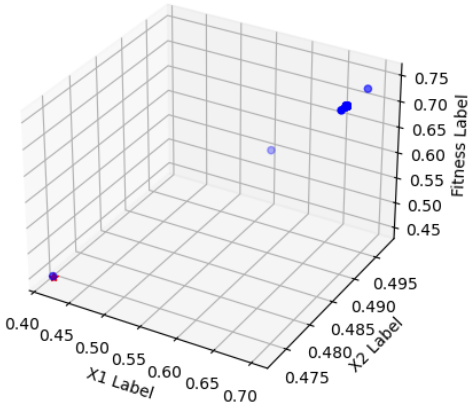


FIGURE 4.30 – Résultats avec $\mu_{CR} = 4$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de taux de croisement μ_{CR}.
Le tableau 4.7 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

μ_{CR}	x_1	x_2	f	$\bar{f}$	σ
0.5	-0.69953	0.498614	0.740734	0.167567	0.12492
1	0.541376	0.497626	0.545467	0.1409	0.12492
2	0.693102	0.497084	0.733315	0.06439	0.08302
3	-0.549785	0.515156	0.537337	0.037645	0.043259
4	0.414712	0.47293	0.449291	0.11894	0.14163

TABLE 4.7 – Résultat expérimentaux de la variation du taux de croisement.

En examinant les résultats sur le tableau 4.7, on remarque que le taux de croisement CR n'a pas un effet sur le processus d'optimisation.

Variation du Facteur de Mutation

On a varié le facteur de mutation μ_F entre 0.5 et 4 pour tester JADE et on a fixé les autres paramètres d'entrée.

La figure 4.31 montre les résultats lorsque μ_F égale à 0.5.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 0.5$.

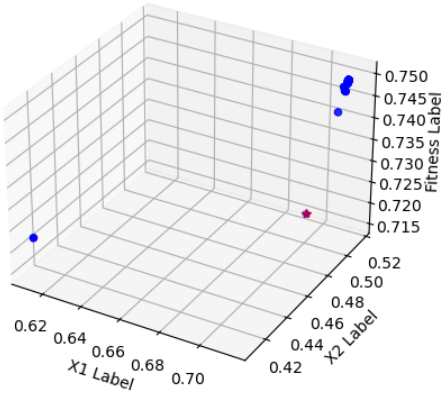


FIGURE 4.31 – Résultats avec $\mu_F = 0.5$

La figure 4.32 montre les résultats lorsque μ_F égale à 1.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 1$.

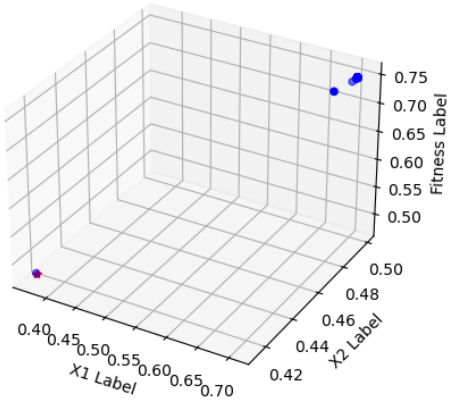


FIGURE 4.32 – Résultats avec $\mu_F = 1$

La figure 4.35 montre les résultats lorsque μ_F égale à 2.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 2$.

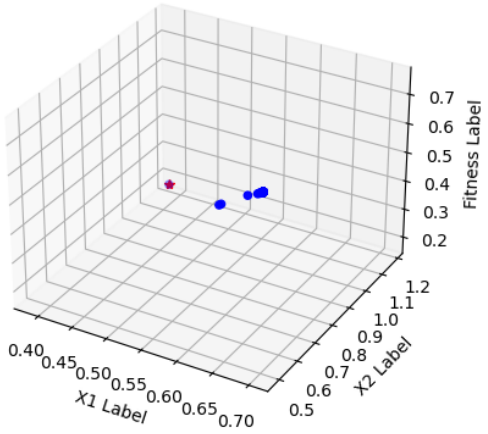


FIGURE 4.33 – Résultats avec $\mu_F = 2$

La figure 4.34 montre les résultats lorsque μ_F égale à 3.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 3$.

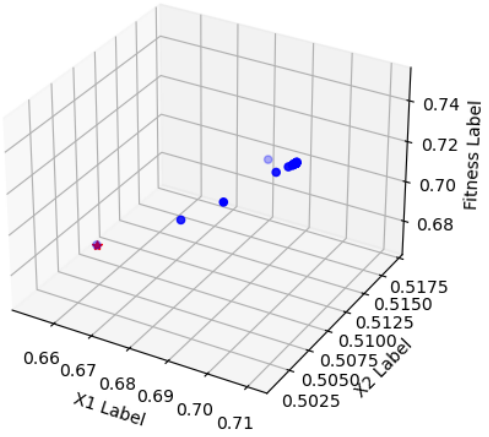


FIGURE 4.34 – Résultats avec $\mu_F = 3$

La figure 4.31 montre les résultats lorsque μ_F égale à 2.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 2$.

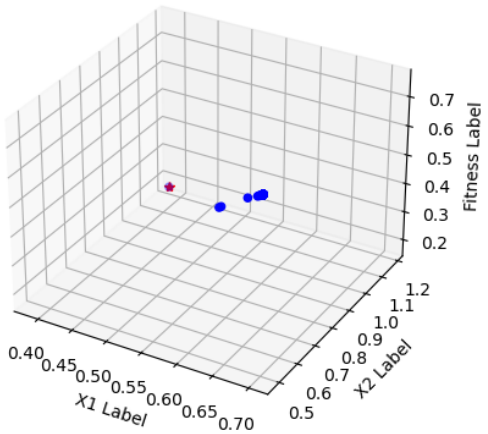


FIGURE 4.35 – Résultats avec $\mu_F = 2$

La figure 4.36 montre les résultats lorsque μ_F égale à 4.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $\mu_{CR} = 0.5$,
- $\mu_F = 4$.

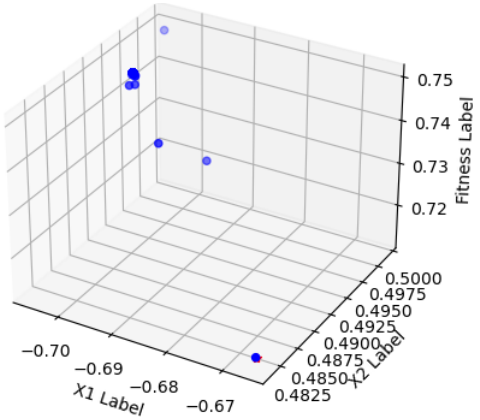


FIGURE 4.36 – Résultats avec $\mu_F = 4$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de facteur de mutation μ_F.
Le tableau 4.8 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

μ_F	x_1	x_2	f	$\bar{f}$	σ
0.5	0.69313	0.51547	0.7512	0.167567	0.12492
1	0.367869	0.413829	0.47892	0.088645	0.120422
2	0.387343	1.179673	0.18231	0.089644	0.178804
3	0.652962	0.507834	0.668586	0.139535	0.142433
4	0.665785	0.481307	0.71231	0.18988	0.091221

TABLE 4.8 – Résultat expérimentaux de la variation du facteur de mutation.

En examinant les résultats sur le tableau 4.8, on remarque que le facteur de mutation F n'a pas un effet sur le processus d'optimisation.

4.3.3 Résultats Expérimentaux de EPSDE

Variation de la Taille de la Population Initiale

On a choisi cinq tailles de population initiale : 100, 200, 300, 400 et 500 pour tester EPSDE et on a fixé les autres paramètres d'entrée.
La figure 4.37 montre les résultats lorsque taille de population initiale égale à 100.

- Taille population initiale = 100,
- Nombre de génération = 10,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$.

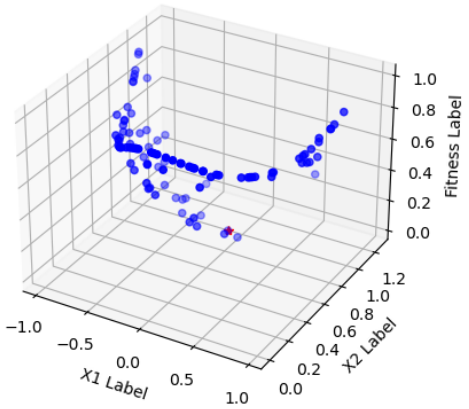


FIGURE 4.37 – Résultats avec une taille de la population initiale = 100

La figure 4.38 montre les résultats lorsque taille de population initiale égale à 200.

- Taille population initiale = 200,
- Nombre de génération = 10,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$.

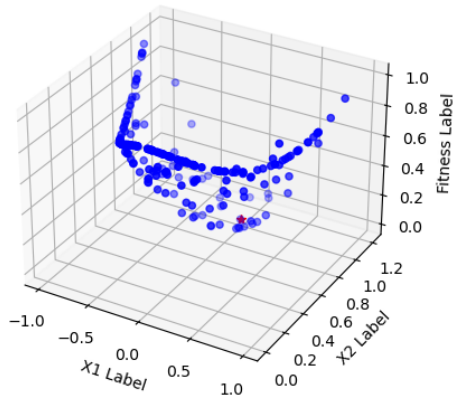


FIGURE 4.38 – Résultats avec une taille de la population initiale = 200

La figure 4.39 montre les résultats lorsque taille de population initiale égale à 300.

- Taille population initiale = 300,
- Nombre de génération = 10,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$.

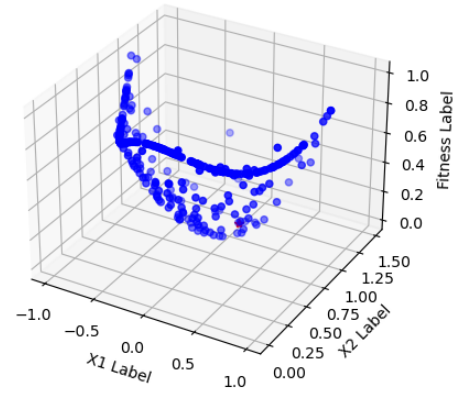


FIGURE 4.39 – Résultats avec une taille de la population initiale = 300

La figure 4.40 montre les résultats lorsque taille de population initiale égale à 400.

- Taille population initiale = 400,
- Nombre de génération = 10,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

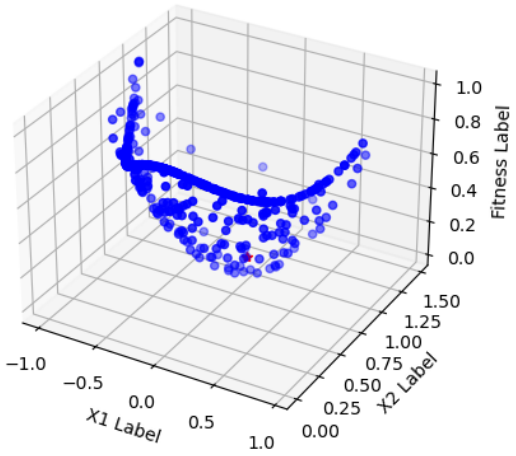


FIGURE 4.40 – Résultats avec une taille de la population initiale = 400

La figure 4.41 montre les résultats lorsque taille de population initiale égale à 500.

- Taille population initiale = 500,
- Nombre de génération = 10,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

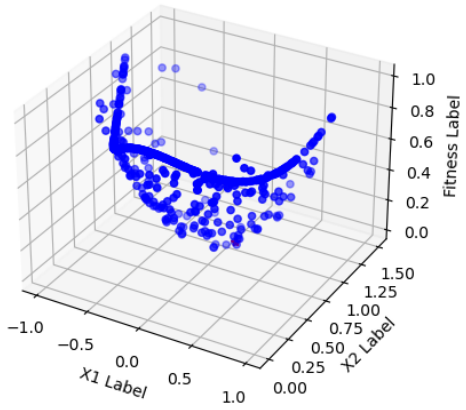


FIGURE 4.41 – Résultats avec une taille de la population initiale = 500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque taille de population.
Le tableau 4.9 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

NP	x_1	x_2	f	$\bar{f}$	σ
100	-0.053648	0.8980617	0.013268	0.001752	0.0041001
200	0.013923	0.96169	0.001661	0.0009097	0.004727
300	-0.01059	1.229367	0.05272	0.0009097	0.00275
400	-0.085406	1.025336	0.007936	0.0004266	0.002216
500	0.101105	0.967455	0.011281	0.0006969	0.002797

TABLE 4.9 – Résultat expérimentaux de la variation de la taille de population.

En examinant les résultats sur le tableau 4.9, on remarque que la taille de population n'a pas un effet sur le processus d'optimisation.

Variation du Nombre de Génération

On a varié le nombre de génération entre 10 et 50 par pas de 10 pour tester EPSDE et on a fixé les autres paramètres d'entrée.
La figure 4.42 montre les résultats lorsque le nombre de génération égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

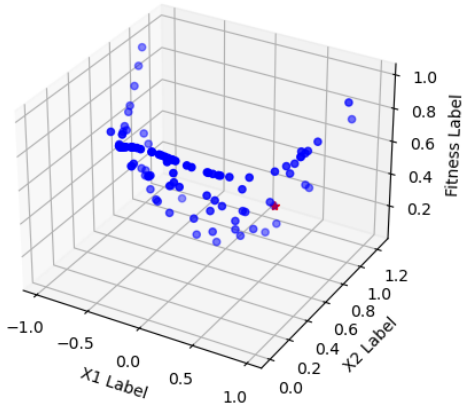


FIGURE 4.42 – Résultats avec nombre de génération = 10

La figure 4.43 montre les résultats lorsque le nombre de génération égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 20,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

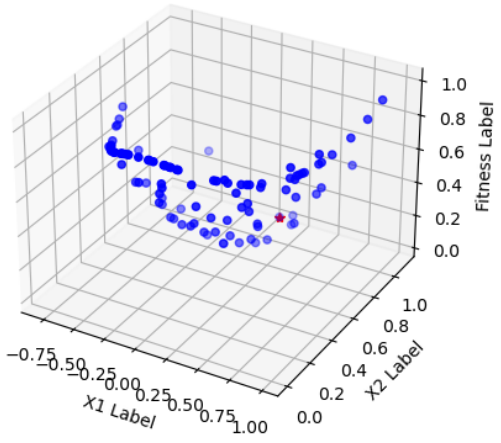


FIGURE 4.43 – Résultats avec nombre de génération = 20

La figure 4.44 montre les résultats lorsque le nombre de génération égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 30,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

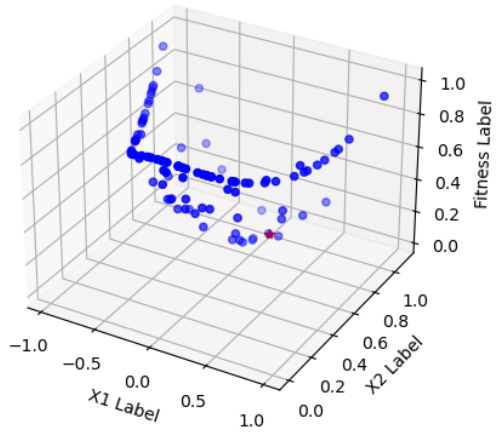


FIGURE 4.44 – Résultats avec nombre de génération = 30

La figure 4.42 montre les résultats lorsque le nombre de génération égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 40,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

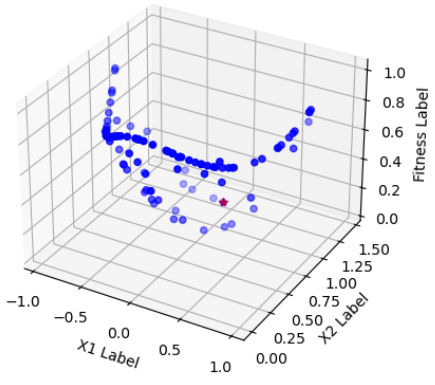


FIGURE 4.45 – Résultats avec nombre de génération = 40

La figure 4.46 montre les résultats lorsque le nombre de génération égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

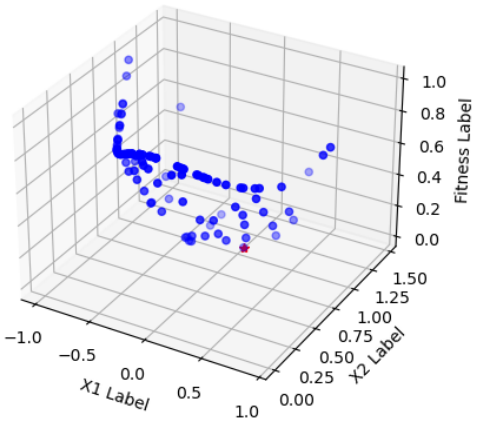


FIGURE 4.46 – Résultats avec nombre de génération = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de génération.
Le tableau 4.10 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

MAX_G	x_1	x_2	f	$\bar{f}$	σ
10	0.97361	1.226652	0.06085	0.001752	0.0041001
20	0.076363	1.110427	0.018025	0.002066	0.005707
30	0.0515508	0.92098	0.008901	0.0003418	0.0010797
40	-0.042835	1.187572	0.03701	0.0006369	0.0014672
50	0.075674	0.97193	0.0065143	0.003121	0.007517

TABLE 4.10 – Résultat expérimentaux de la variation du nombre de génération.

En examinant les résultats sur le tableau 4.10, on remarque que le nombre de génération n'a pas un effet sur le processus d'optimisation.

Variation du Taux de Croisement

On a varié le taux de croisement CR pour tester EPSDE et on a fixé les autres paramètres d'entrée.

La figure 4.47 montre les résultats lorsque $CR \in [0.1\ ,\ 0.9]$,

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

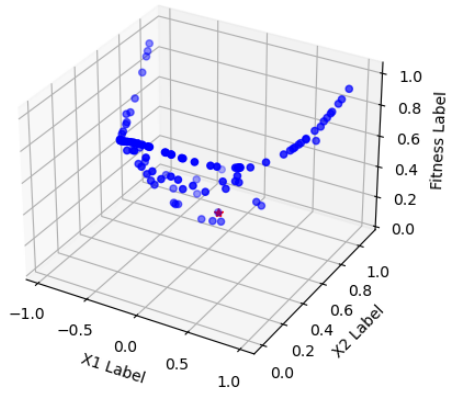


FIGURE 4.47 – Résultats avec $CR \in [0.1\ ,\ 0.9]$

La figure 4.48 montre les résultats lorsque $CR \in [0.1\ ,\ 2]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 2]$,
- $F \in [0.4\ ,\ 0.9]$.

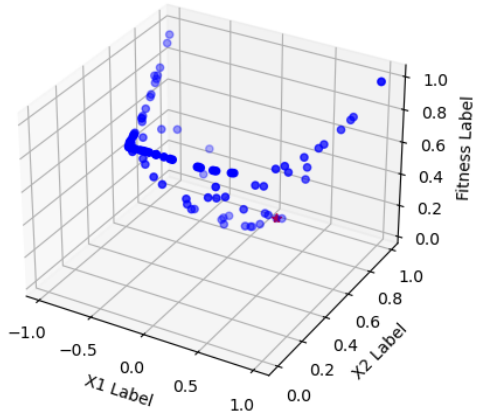


FIGURE 4.48 – Résultats avec $CR \in [0.1\ ,\ 2]$

La figure 4.49 montre les résultats lorsque $CR \in [0.1\ ,\ 3]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 3]$,
- $F \in [0.4\ ,\ 0.9]$.

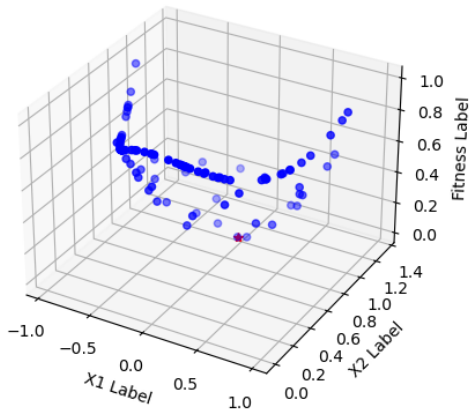


FIGURE 4.49 – Résultats avec $CR \in [0.1\ ,\ 3]$

La figure 4.50 montre les résultats lorsque $CR \in [0.1\ ,\ 4]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 4]$,
- $F \in [0.4\ ,\ 0.9]$.

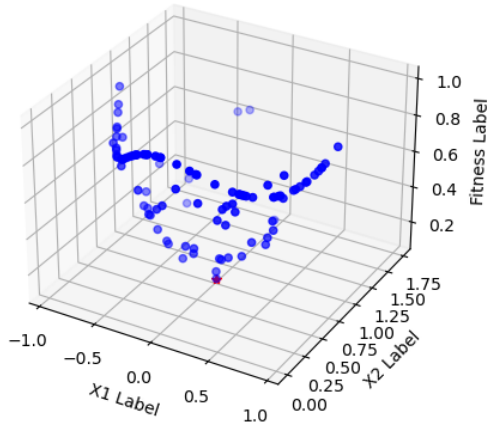


FIGURE 4.50 – Résultats avec $CR \in [0.1\ ,\ 4]$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de taux de croisement CR.
Le tableau 4.11 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

CR	x_1	x_2	f	$\bar{f}$	σ
$CR \in [0.1\ ,\ 0.9]$	-0.151628	0.843261	0.04755	0.003591	0.139721
$CR \in [0.1\ ,\ 1]$	0.541376	0.497626	0.545467	0.0006679	0.0014841
$CR \in [0.1\ ,\ 2]$	0.1348925	0.904049	0.027402	0.0022841	0.007072
$CR \in [0.1\ ,\ 3]$	0.019347	0.929413	0.005356	0.00331567	0.009417
$CR \in [0.1\ ,\ 4]$	-0.012362	0.883871	0.013638	0.0060942	0.0176004

TABLE 4.11 – Résultat expérimentaux de la variation du taux de croisement.

En examinant les résultats sur le tableau 4.11, on remarque que le taux de croisement n'a pas un effet sur le processus d'optimisation.

Variation du Facteur de Mutation

On a varié le facteur de mutation F pour tester EPSDE et on a fixé les autres paramètres d'entrée.

La figure 4.51 montre les résultats lorsque $F \in [0.4\ ,\ 0.9]$. .

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

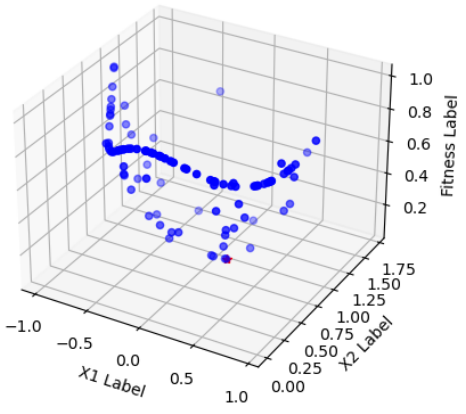


FIGURE 4.51 – Résultats avec $F \in [0.4\ ,\ 0.9]$

La figure 4.52 montre les résultats lorsque $F \in [0.4\ ,\ 1]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 1]$.

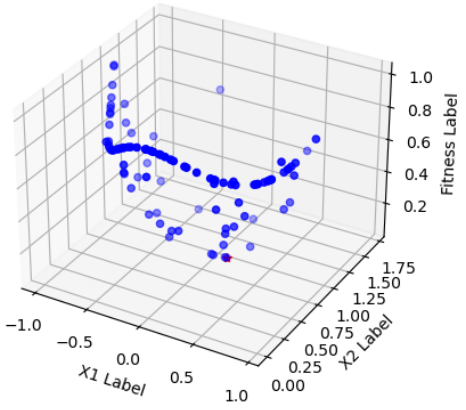


FIGURE 4.52 – Résultats avec $F \in [0.4\ ,\ 1]$

La figure 4.53 montre les résultats lorsque $F \in [0.4\ ,\ 2]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 2]$.

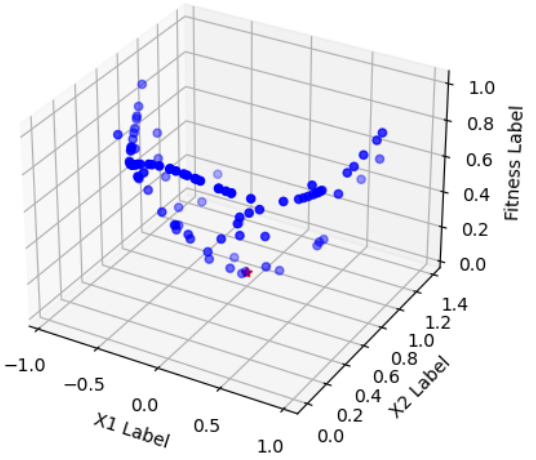


FIGURE 4.53 – Résultats avec $F \in [0.4\ ,\ 2]$

La figure 4.54 montre les résultats lorsque $F \in [0.4\ ,\ 3]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 3]$.

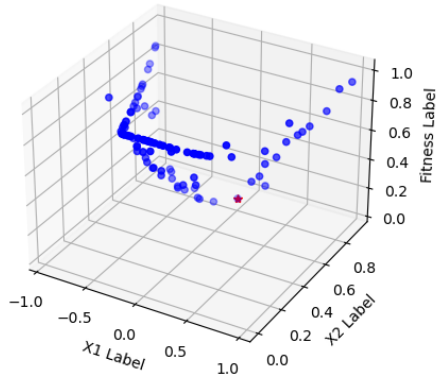


FIGURE 4.54 – Résultats avec $F \in [0.4\ ,\ 3]$

La figure 4.55 montre les résultats lorsque $F \in [0.4\ ,\ 4]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 4]$.

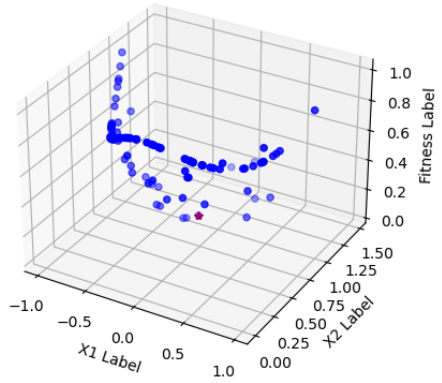


FIGURE 4.55 – Résultats avec $F \in [0.4\ ,\ 4]$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de facteur de mutation F.
Le tableau 4.12 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

F	x_1	x_2	f	f	σ
$F \in [0.4\ ,\ 0.9]$	0.17317	0.848411	0.052967	0.0020011	0.0055015
$F \in [0.4\ ,\ 1]$	-0.025116	0.820477	0.032859	0.0041025	0.01325
$F \in [0.4\ ,\ 2]$	-0.14371	0.978903	0.035887	0.002626	0.006849
$F \in [0.4\ ,\ 3]$	-0.1882606	0.978903	0.035887	0.0011128	0.005243
$F \in [0.4\ ,\ 4]$	0.1754271	1.106735	0.0421672	0.0014121	0.00419

TABLE 4.12 – Résultat expérimentaux de la variation du facteur de mutation.

En examinant les résultats sur le tableau 4.12, on remarque que le facteur de mutation F n'a pas un effet sur le processus d'optimisation.

4.3.4 Résultats Expérimentaux de CoDE

Variation de la Taille de la Population Initiale

On a choisi cinq tailles de population initiale : 100, 200, 300, 400 et 500 pour tester CoDE et on a fixé les autres paramètres d'entrée.
La figure 4.56 montre les résultats lorsque taille de population initiale égale à 100.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 1500.

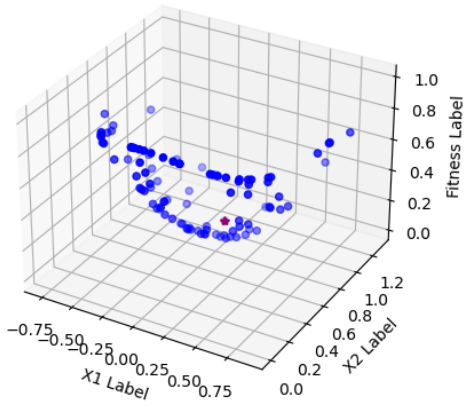


FIGURE 4.56 – Résultats avec une taille de la population initiale = 100

La figure 4.57 montre les résultats lorsque taille de population initiale égale à 200.

- Taille population initiale = 200,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 1500.

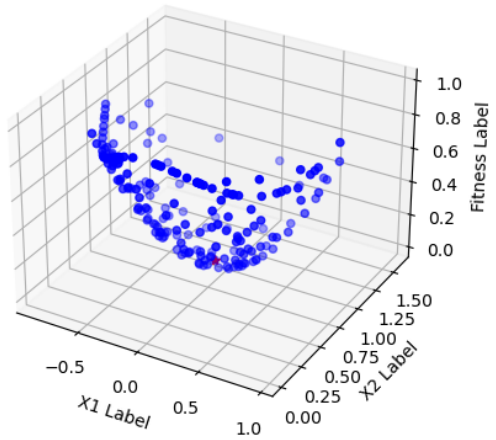


FIGURE 4.57 – Résultats avec une taille de la population initiale = 200

La figure 4.58 montre les résultats lorsque taille de population initiale égale à 300.

- Taille population initiale = 300,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$,
- MAX_FES = 1500.

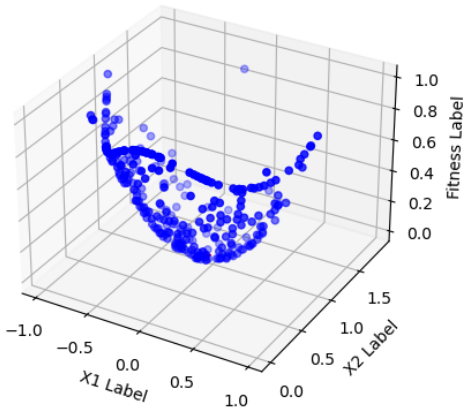


FIGURE 4.58 – Résultats avec une taille de la population initiale = 300

La figure 4.59 montre les résultats lorsque taille de population initiale égale à 400.

- Taille population initiale = 400,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$,
- MAX_FES = 1500.

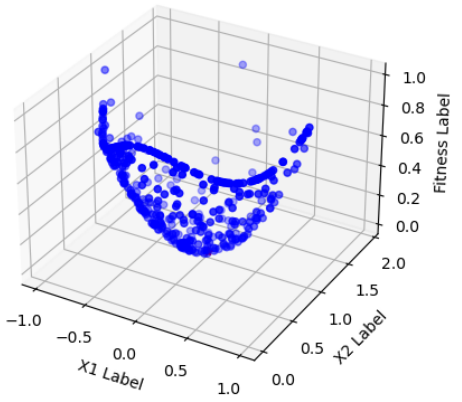


FIGURE 4.59 – Résultats avec une taille de la population initiale = 400

La figure 4.60 montre les résultats lorsque taille de population initiale égale à 500.

- Taille population initiale = 500,
- $CR \in [0.1 , 0.9]$,
- $F \in [0.4 , 0.9]$,
- MAX_FES = 1500.

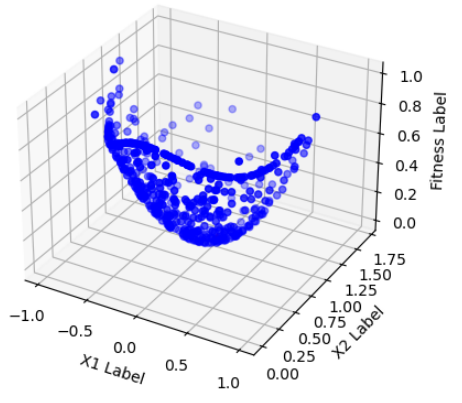


FIGURE 4.60 – Résultats avec une taille de la population initiale = 500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective

pour chaque taille de population.
Le tableau 4.13 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

NP	x_1	x_2	f	$\bar{f}$	σ
100	0.04219	0.99409	0.001814	0.0002986	0.001019
200	-0.02232	1.00888	0.000577	0.0002035	0.00065802
300	0.018541	1.0325	0.0014003	5.06*10⁻5	0.0001118
400	0.013812	1.00997	0.0002903	5.21*10⁻5	0.0001881
500	-0.007157	0.966249	0.00119	0.00016648	0.0007453

TABLE 4.13 – Résultat expérimentaux de la variation de la taille de population.

En examinant les résultats sur le tableau 4.13, on remarque que la taille de population a un effet sur le processus d'optimisation.

Variation du Nombre MAX_FES

On a varié le nombre MAX_FES qui représente le nombre d'évaluation de la fonction objective et le critère d'arrêt de l'algorithme pour tester CoDE et on a fixé les autres paramètres d'entrée.
La figure 4.63 montre les résultats lorsque MAX_FES égale à 500.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 500.

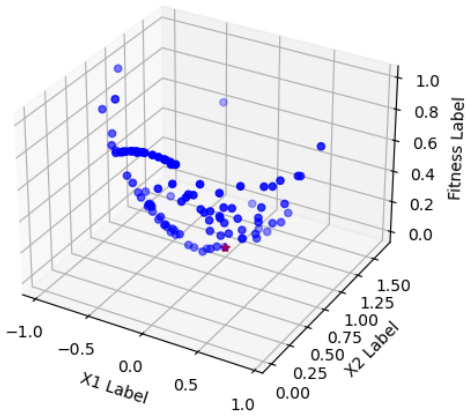


FIGURE 4.61 – Résultats avec MAX_FES = 500

La figure 4.62 montre les résultats lorsque MAX_FES égale à 1000.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 1000.

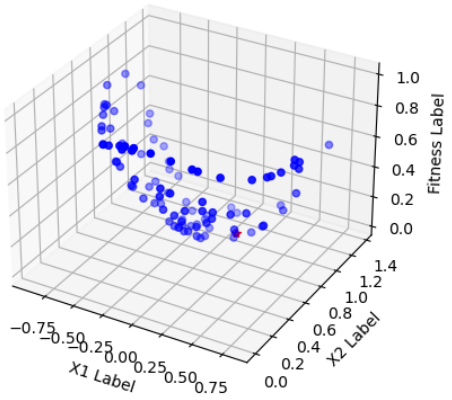


FIGURE 4.62 – Résultats avec MAX_FES = 1000

La figure 4.62 montre les résultats lorsque MAX_FES égale à 1500.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 1500.

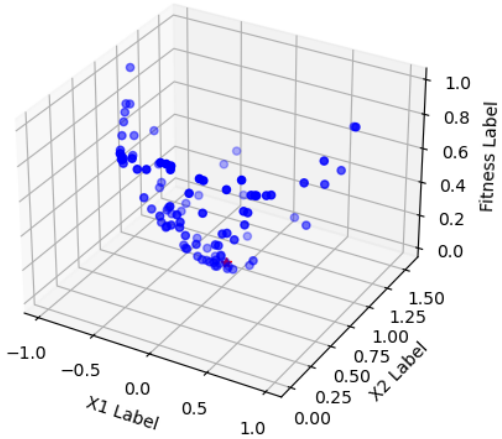


FIGURE 4.63 – Résultats avec MAX_FES = 1500

La figure 4.64 montre les résultats lorsque MAX_FES égale à 2000.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 2000.

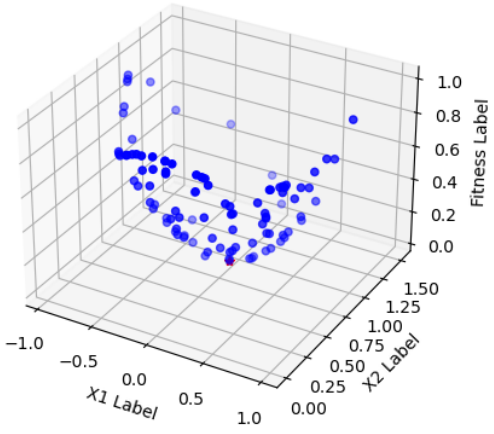


FIGURE 4.64 – Résultats avec MAX_FES = 2000

La figure 4.65 montre les résultats lorsque MAX_FES égale à 2500.

- Taille population initiale = 100,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$,
- MAX_FES = 2500.

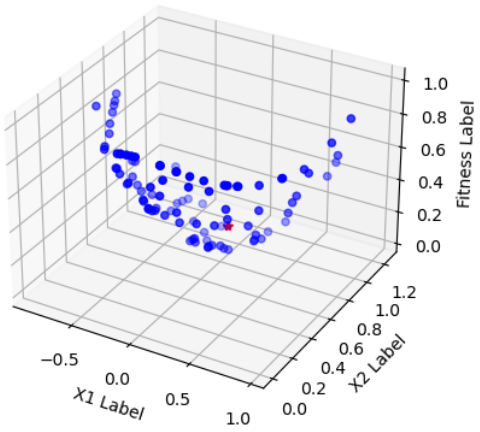


FIGURE 4.65 – Résultats avec MAX_FES = 2500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque valeur du nombre MAX_FES.
Le tableau

MAX_FES	x_1	x_2	f	$\bar{f}$	σ
500	0.0003913	0.984713	0.0002338	4.81*10^{-}5	0.000103
1000	0.1049508	0.9658721	0.0121794	0.00108	0.00343
1500	-0.0723209	0.895295	0.0161933	0.0002986	0.001019
2000	0.027284	0.841742	0.025789	0.0007189	0.002502
2500	-0.123128	1.033914	0.0186081	0.000349	0.00126

TABLE 4.14 – Résultat expérimentaux de la variation du nombre MAX_FES.

En examinant les résultats sur le tableau

Variation du Taux de Croisement

On a varié le taux de croisement CR pour tester CODE et on a fixé les autres paramètres d'entrée.

La figure 4.66 montre les résultats lorsque $CR \in [0.1\ ,\ 0.9]$,

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

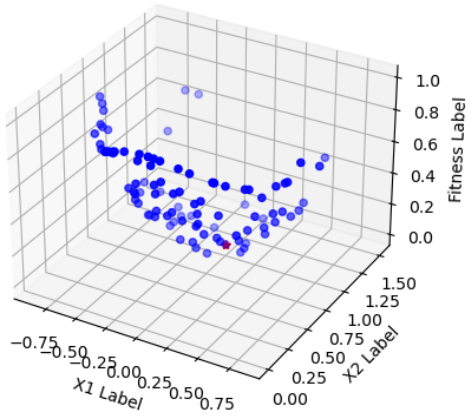


FIGURE 4.66 – Résultats avec $CR \in [0.1\ ,\ 0.9]$

La figure 4.67 montre les résultats lorsque $CR \in [0.1\ ,\ 2]$,

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 2]$,
- $F \in [0.4\ ,\ 0.9]$.

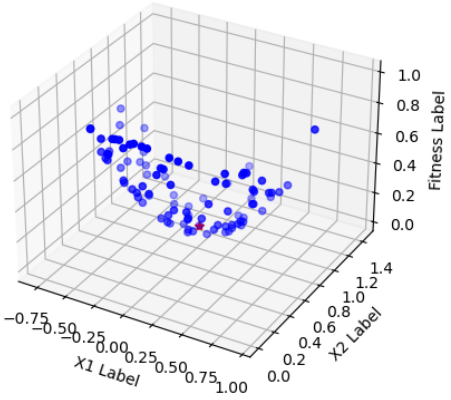


FIGURE 4.67 – Résultats avec $CR \in [0.1\ ,\ 2]$

La figure 4.68 montre les résultats lorsque $CR \in [0.1\ ,\ 3]$,

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 3]$,
- $F \in [0.4\ ,\ 0.9]$.

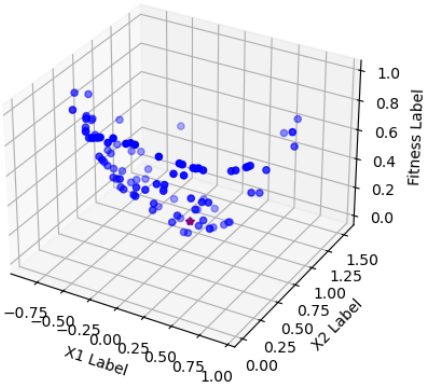


FIGURE 4.68 – Résultats avec $CR \in [0.1\ ,\ 3]$

La figure 4.69 montre les résultats lorsque $CR \in [0.1\ ,\ 4]$,

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 4]$,
- $F \in [0.4\ ,\ 0.9]$.

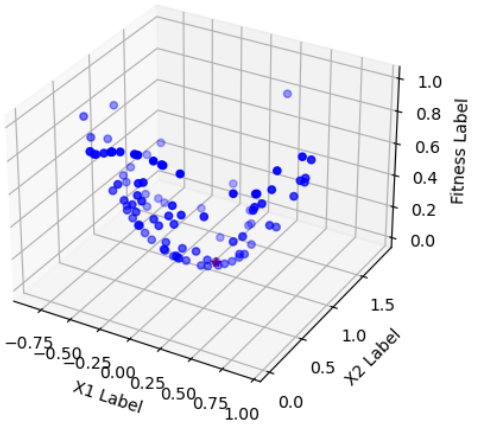


FIGURE 4.69 – Résultats avec $CR \in [0.1\ ,\ 4]$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de taux de croisement CR.
Le tableau 4.15 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

CR	x_1	x_2	f	$\bar{f}$	σ
$CR \in [0.1\ ,\ 0.9]$	-0.0004084	1.003471	1.18438	0.0005183	0.001617
$CR \in [0.1\ ,\ 1]$	-0.06976	0.935401	0.009039	0.001106	0.005361
$CR \in [0.1\ ,\ 2]$	-0.07411	0.993091	0.005541	0.0002788	0.000939
$CR \in [0.1\ ,\ 3]$	0.08965	0.98209	0.008358	0.0006787	0.002352
$CR \in [0.1\ ,\ 4]$	-0.06789	1.084509	0.01175	0.0001074	0.0002789

TABLE 4.15 – Résultat expérimentaux de la variation du taux de croisement.

En examinant les résultats sur le tableau 4.15, on remarque que le taux de croisement CR n'a pas un effet sur le processus d'optimisation.

Variation du Facteur de Mutation

On a varié le facteur de mutation F pour tester CoDE et on a fixé les autres paramètres d'entrée.

La figure 4.70 montre les résultats lorsque $F \in [0.4\ ,\ 0.9]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 0.9]$.

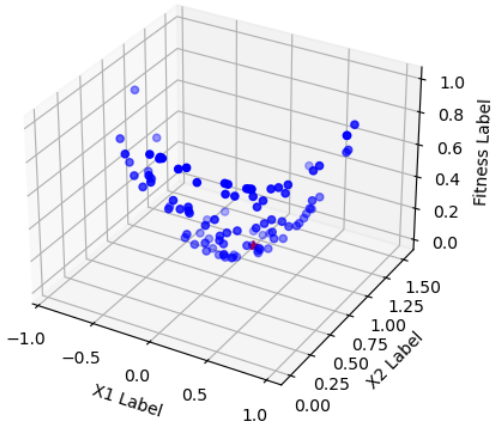


FIGURE 4.70 – Résultats avec $F \in [0.4\ ,\ 0.9]$

La figure **??** montre les résultats lorsque $F \in [0.4\ ,\ 2]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 2]$.

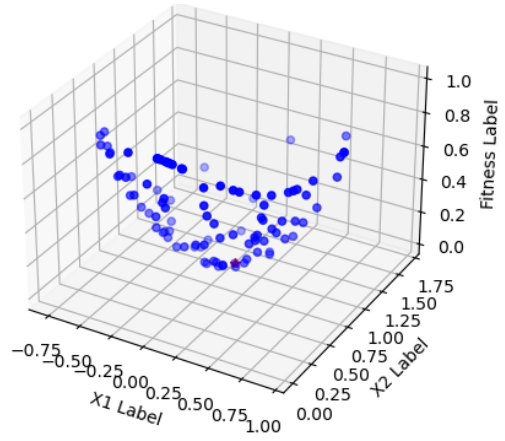


FIGURE 4.71 – Résultats avec $F \in [0.4\ ,\ 2]$

La figure 4.72 montre les résultats lorsque $F \in [0.4\ ,\ 3]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 3]$.

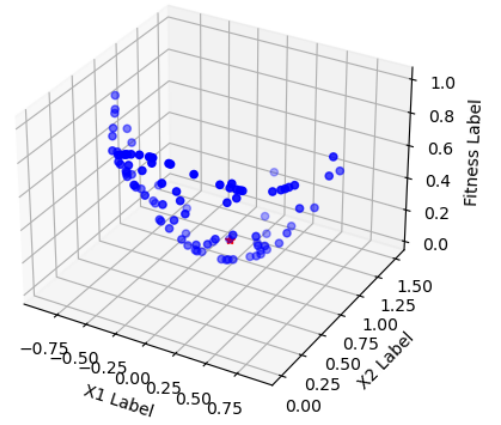


FIGURE 4.72 – Résultats avec $F \in [0.4\ ,\ 3]$

La figure 4.73 montre les résultats lorsque $F \in [0.4\ ,\ 4]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 4]$.

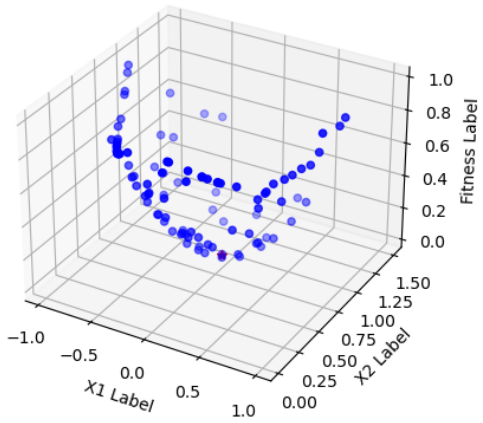


FIGURE 4.73 – Résultats avec $F \in [0.4\ ,\ 4]$

La figure 4.74 montre les résultats lorsque $F \in [0.4\ ,\ 5]$.

- Taille population initiale = 100,
- Nombre de génération = 50,
- $CR \in [0.1\ ,\ 0.9]$,
- $F \in [0.4\ ,\ 5]$.

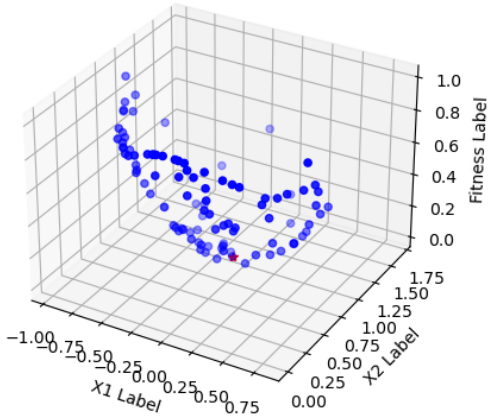


FIGURE 4.74 – Résultats avec $F \in [0.4\ ,\ 5]$

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de facteur de mutation F.
Le tableau 4.16 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

F	x_1	x_2	f	$\bar{f}$	σ
$F \in [0.4\ ,\ 0.9]$	0.066245	1.05398	0.007303	0.0002897	0.0007353
$F \in [0.4\ ,\ 1]$	-0.025116	0.820477	0.032859	0.000516	0.001577
$F \in [0.4\ ,\ 2]$	-0.14371	0.978903	0.035887	4.58*10^{-}5	0.000111
$F \in [0.4\ ,\ 3]$	-0.1882606	0.978903	0.035887	0.000965	0.00378
$F \in [0.4\ ,\ 4]$	0.1754271	1.106735	0.0421672	1.802*10^{-}5	3.127*10^{-}5

TABLE 4.16 – Résultat expérimentaux de la variation du facteur de mutation.

En examinant les résultats sur le tableau 4.8, on remarque que le facteur de mutation F a un effet sur le processus d'optimisation.

4.3.5 Résultats Expérimentaux de PEDEV (version parallèle)

Maintenant, nous allons tester la version parallèle d'EDEV (PEDEV) en utilisant le même problème précédent définie dans [16] et on compare les résultats. Pour la fonction de test donnée $f(\vec{x}) = x_1^2 + (x_2 - 1)^2$, la solution optimale calculée dans [16] est $\vec{x}^* = (0.707036070037170616, 0.500000004333606807)$ et $f(\vec{x}^*) = 0.7499$ tels que $-1 \leq x_1 \leq 1$, $-1 \leq x_2 \leq 1$.

Variation de la Taille de la Population Initiale

On a choisi cinq tailles de population initiale : 100, 200, 300, 400 et 500 pour tester PEDEV et on a fixé les autres paramètres d'entrée.
La figure 4.75 montre les résultats lorsque la taille initiale de la population est 100.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

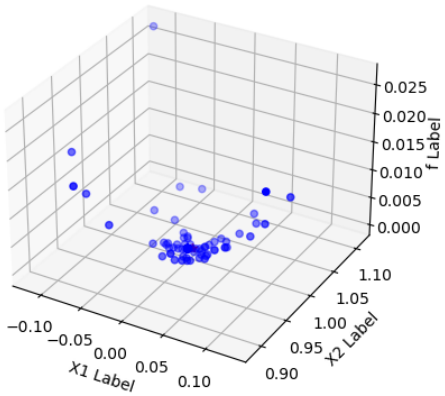


FIGURE 4.75 – Résultats avec une taille de la population initiale = 100

La figure 4.76 montre les résultats lorsque la taille initiale de la population est 200.

- Taille population initiale = 200,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

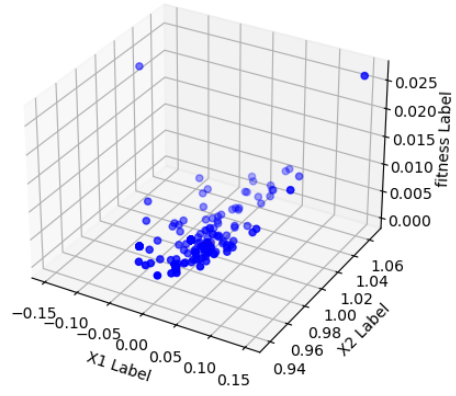


FIGURE 4.76 – Résultats avec une taille de la population initiale = 200

La figure 4.77 montre les résultats lorsque la taille initiale de la population est 300.

- Taille population initiale = 300,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

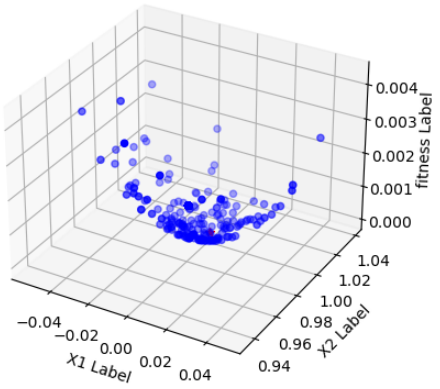


FIGURE 4.77 – Résultats avec une taille de la population initiale = 300

La figure 4.78 montre les résultats lorsque la taille initiale de la population est 400.

- Taille population initiale = 400,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

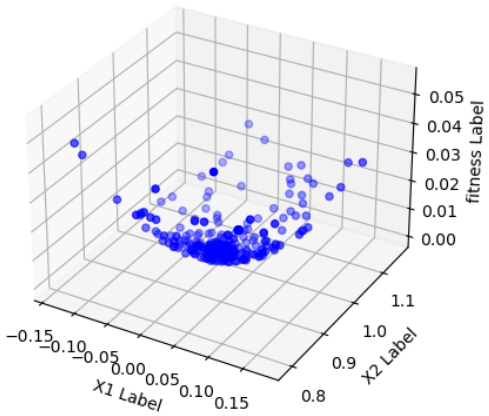


FIGURE 4.78 – Résultats avec une taille de la population initiale = 400

La figure 4.79 montre les résultats lorsque la taille initiale de la population est 500.

- Taille population initiale = 500,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

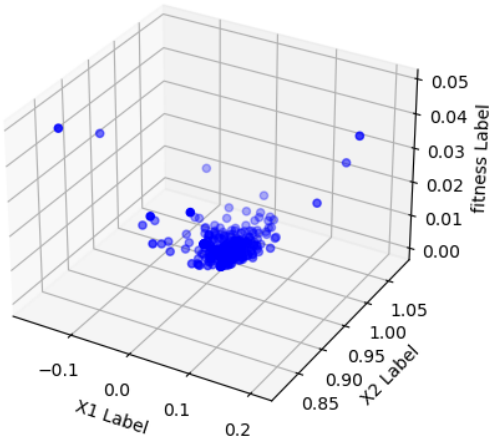


FIGURE 4.79 – Résultats avec une taille de la population initiale = 500

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque taille de population.
Le tableau 4.17 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

NP	x_1	x_2	f	$\bar{f}$	σ
100	0.0007173	0.99393	3.726*10−5	4.779*10−6	8.579*10−6
200	0.002564	1.002232	1.155*10−5	3.248*10−6	7.779*10−6
300	0.001385	1.000097	1.925*10−6	1.816*10−6	3.293*10−6
400	0.0008912	1.00306	1.018*10−5	1.252*10−6	4.027*10−6
500	-0.000456	0.99975	2.683*10−7	1.345*10−6	4.312*10−6

TABLE 4.17 – Résultat expérimentaux de la variation de la taille de population.

En examinant les résultats sur le tableau 4.17, on remarque que la taille de la population a un effet sur le processus d'optimisation.

Variation du Nombre de Génération

On a varié le nombre de génération entre 10 et 50 par pas de 10 pour tester PEDEV et on a fixé les autres paramètres d'entrée.
La figure 4.80 montre les résultats lorsque le nombre de génération égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

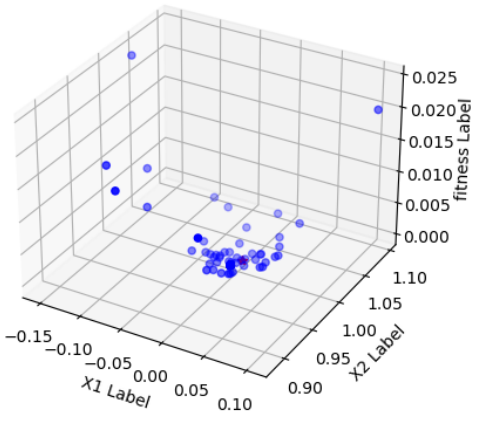


FIGURE 4.80 – Résultats avec nombre de génération = 10

La figure 4.81 montre les résultats lorsque le nombre de génération égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 20,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

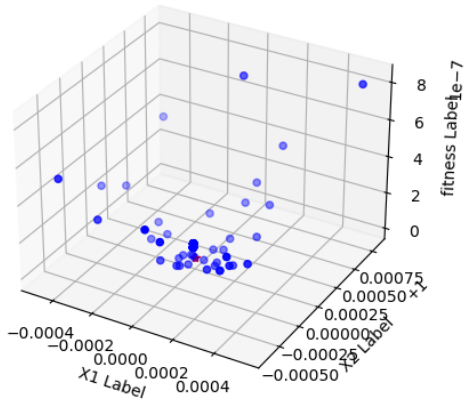


FIGURE 4.81 – Résultats avec nombre de génération = 20

La figure 4.82 montre les résultats lorsque le nombre de génération égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 30,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

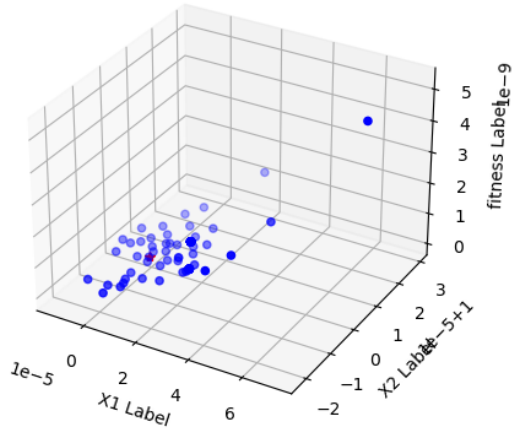


FIGURE 4.82 – Résultats avec nombre de génération = 30

La figure 4.83 montre les résultats lorsque le nombre de génération égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 40,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

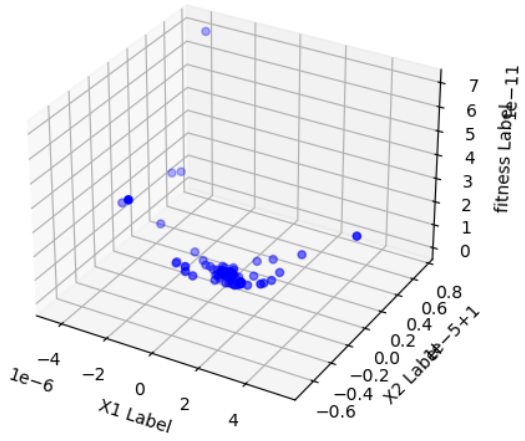


FIGURE 4.83 – Résultats avec nombre de génération = 40

La figure 4.84 montre les résultats lorsque le nombre de génération égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 50,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

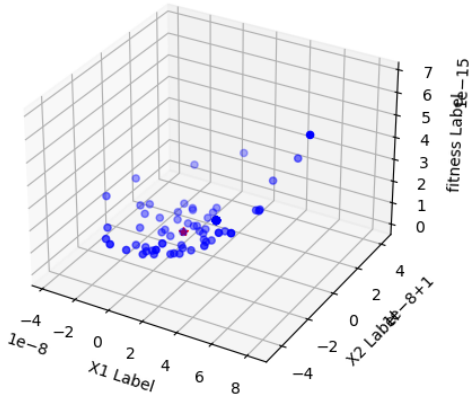


FIGURE 4.84 – Résultats avec nombre de génération = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre de génération.
Le tableau 4.18 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

MAX_G	x_1	x_2	f	$\bar{f}$	σ
10	0.0007173	0.9939	3.726*10⁻5	4.779*10⁻6	8.579*10⁻6
20	-9.774*10⁻5	1.0004	1.241*10⁻8	0000	00000
30	1.136*10⁻6	0.9999	1.356*10⁻12	0000	00000
40	-7.055*10⁻8	1	7.179*10⁻15	0000	00000
50	-8.528*10⁻10	1	3.577*10⁻18	0000	00000

TABLE 4.18 – Résultat expérimentaux de la variation du nombre de génération.

En examinant les résultats sur le tableau 4.18, on remarque que le nombre de génération a un effet sur le processus d'optimisation.

Variation du Nombre ng

On a varié le nombre ng entre 10 et 50 par pas de 10 pour tester PEDEV et on a fixé les autres paramètres d'entrée.
La figure 4.85 montre les résultats lorsque ng égale à 10.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 10,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

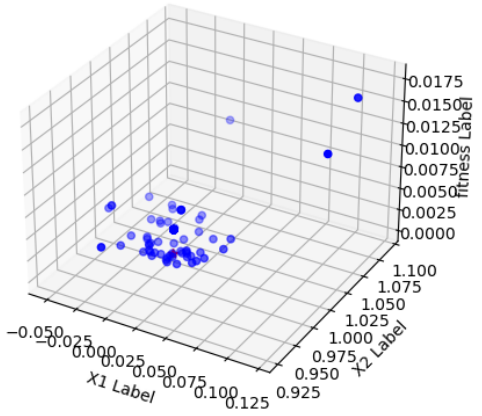


FIGURE 4.85 – Résultats avec ng = 10

La figure 4.86 montre les résultats lorsque ng égale à 20.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 20,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

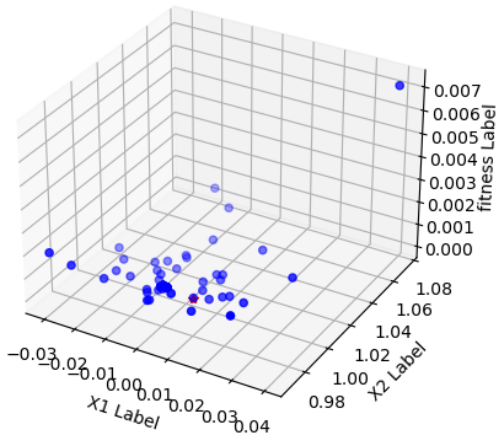


FIGURE 4.86 – Résultats avec ng = 20

La figure 4.87 montre les résultats lorsque ng égale à 30.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 30,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

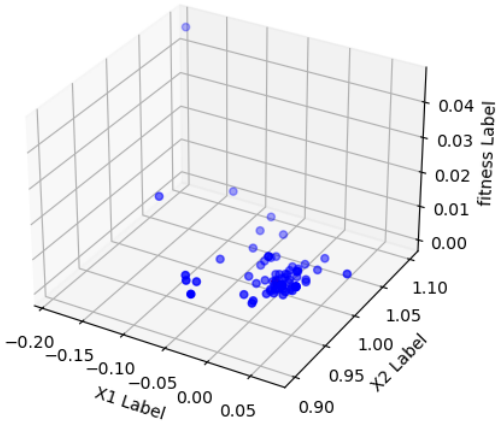


FIGURE 4.87 – Résultats avec ng = 30

La figure 4.88 montre les résultats lorsque ng égale à 40.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 40,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

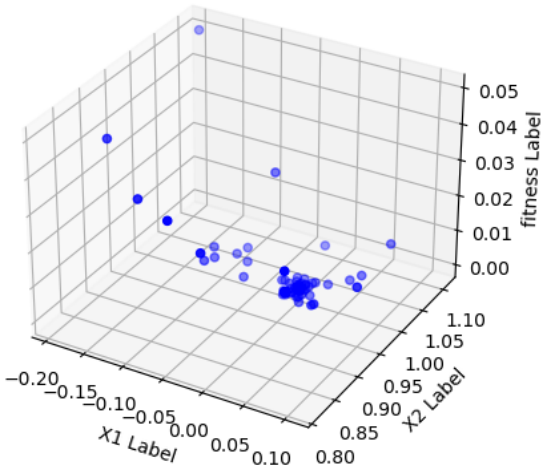


FIGURE 4.88 – Résultats avec ng = 40

La figure 4.89 montre les résultats lorsque ng égale à 50.

- Taille population initiale = 100,
- Nombre de génération = 10,
- ng= 50,
- MAX_FES= 1500,
- $\lambda_i = 0.2$.

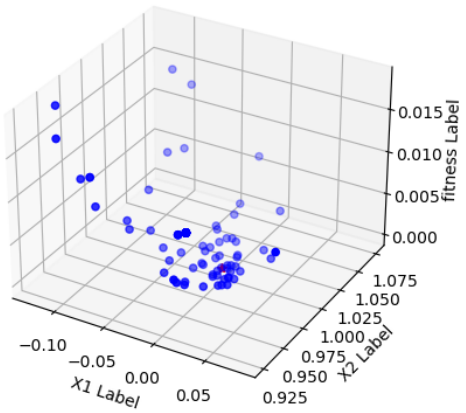


FIGURE 4.89 – Résultats avec ng = 50

Pour avoir des résultats plus crédibles, on a calculé 30 fois la valeur de la fonction objective pour chaque nombre ng.
Le tableau 4.19 résume les résultat expérimentaux trouvés tels que :

- x_1 et x_2 représentent les coordonnées de la solution optimale $\vec{x}^*$,
- f représente la valeur de la fonction objective de $\vec{x}^*$,
- $\bar{f}$ représente la moyenne de l'ensemble de 30 valeurs de la fonction objective,
- σ représente l'écart type de l'ensemble de 30 valeurs de la fonction objective.

ng	x_1	x_2	f	$\bar{f}$	σ
10	-0.00012	1.00123	1764*10⁻6	0.782*10⁻6	1.432*10⁻6
20	0.0011468	1.0001	1.315*10⁻6	4.779*10⁻6	8.579*10⁻6
30	-0.002872	0.99753	1.432*10⁻6	4.754*10⁻6	7.145*10⁻6
40	-0.002097	0.99851	2.59*10⁻6	4.324*10⁻6	7.823*10⁻6
50	-0.001512	0.99935	2.71*10⁻6	3.112*10⁻6	8.457*10⁻6

TABLE 4.19 – Résultat expérimentaux de la variation du nombre ng.

En examinant les résultats sur le tableau 4.19, on remarque que le nombre ng a un effet sur le processus d'optimisation.

Comparaison entre EDEV et PEDEV

Dans le processus de comparaison entre EDEV et PEDEV on a utilisé le problème que nous avions vu auparavant [16] avec les mêmes entrées.

Le tableau 4.20 montre les résultat expérimentaux du temps d'exécution trouvés d'EDEV et de PEDEV en variant la taille de population.

NP	**Temps d'exécution d'EDEV**	**Temps d'exécution de PEDEV**
100	10.55 s	91.63 s
200	60.93 s	177.37 s
300	113.08 s	285.46 s
400	541.7 s	463.7 s
500	932.8 s	714.5 s

TABLE 4.20 – Résultat expérimentaux de la comparaison entre EDEV et PEDEV (temps d'exécution).

En examinant les résultats sur le tableau 4.20, on remarque que le temps d'exécution de PEDEV est inférieur à celui d'EDEV lorsque la taille de la population initiale est supérieur ou égale à 400.

Le tableau 4.21 montre les résultat expérimentaux de la moyenne et l'écart type de l'ensemble de 30 valeurs de la fonction objective trouvés d'EDEV et de PEDEV en variant la taille de population.

NP	$\bar{f}$ **d'EDEV**	σ **d'EDEV**	$\bar{f}$ **de PEDEV**	σ **de PEDEV**
100	1.30786	2.3391	4.779*10⁻6	8.579*10⁻6
200	2.2794	2.9438	3.248*10⁻6	7.779*10⁻6
300	1.616*10⁻6	2.1208*10⁻6	1.816*10⁻6	3.293*10⁻6
400	1.254*10⁻8	2.354*10⁻8	1.252*10⁻6	4.027*10⁻6
500	2.589*10⁻9	2.879*10⁻9	1.345*10⁻6	4.321*10⁻6

TABLE 4.21 – Résultat expérimentaux de la comparaison entre EDEV et PEDEV (espace objectif).

En examinant les résultats sur le tableau 4.20, on remarque que l'espace objectif est amélioré dans PEDEV par rapport à EDEV.

Conclusion

Dans ce chapitre, nous avons présenté les outils que nous avons utilisés pour réaliser notre application, puis les résultats de notre implémentation sous la forme de courbes. Dans la dernière section nous avons discuté les résultats de l'exécution de l'algorithme implémenté (EDEV) avec divers paramètres et on a prouvé l'efficacité de la version parallèle d'EDEV (c'est-à-dire PEDEV).

Conclusion Générale

Conclusion Générale

Dans ce chapitre, nous avons présenté les outils que nous avons utilisés pour réaliser notre application, puis les résultats de notre implémentation sous la forme de courbe. Dans la dernière section nous avons discuté les résultats de l'exécution de l'algorithme implémenté (EDEV) avec divers paramètres et on a prouvé l'efficacité de la version parallélisé d'EDEV (c'est-à-dire PEDEV).

Les algorithmes à évolution différentielle (DE) sont assez exigeants en temps de calcul car ils cherchent itérativement une solution optimale ou une solution quasi optimale. Pour cette raison, les DEs nécessitent une parallélisation pour minimiser le temps d'exécution. La parallélisation est faite par l'attribution de différentes parties de l'espace de recherche à différents processeurs. Pour la satisfaction de l'utilisateur qui recherche toujours des résultats efficaces dans un temps réduit, nous avons créé la version parallèle (PEDEV) de l'algorithme à évolution différentielle EDEV pour minimiser le temps de calcul de ce dernier.
Lors de la réalisation du projet, nous avons acquis des connaissances sur :

- Problèmes d'optimisation,
- Métaheuristiques,
- Algorithmes évolutionnaire (EAs),
- Algorithmes à évolution différentielle,
- Calcul parallèle, les architectures parallèles et les modèles de programmation parallèle,

Nous avons implémenté l'algorithme à évolution différentielle EDEV et les trois algorithmes JADE, CoDE, EPSDE tels qu'ils sont décrit dans leurs papiers originaux. L'objectif principal de cette étude n'est pas d'implémenter la version séquentielle d'EDEV, mais de rechercher des résultats efficaces en minimisant le temps de calcul d'EDEV. Pour cette raison, nous avons proposé la version parallèle d'EDEV (PEDEV).

Nous avons implémenté ces algorithmes en utilisant le langage de programmation Python. Notre application est utile pour toute fonction de test. Il est important de mentionner dans cette thèse que le PEDEV proposé représente la première version parallèle d'EDEV dans la littérature. Le temps d'exécution de PEDEV est inférieur à celui d'EDEV et le temps de calcul est un critère exigé par l'utilisateur.
Dans nos recherches futures, nous avons l'intention de nous concentrer sur d'autres questions qui restent à résoudre, dont certaines sont :

- Utilisez une étude de cas sélectionnée du monde réel,
- Transformez le problème en problème d'optimisation multi-objectif.
- Mettre en œuvre d'autres algorithmes à évolution différentielle (leurs versions séquentielles et parallèles) et les comparer avec EDEV et PEDEV pour prouver l'efficacité du PEDEV proposé.

Bibliographie

[1] matplotlib. `http://matplotlib.org/`, 2017. accessed on May 18, 2017.

[2] Flowchart maker & online diagram software, jun 2019.

[3] multiprocessing process based threading interface python documentation, may 2019.

[4] Métaheuristiques, mai 2020.

[5] Wikipedia, June 21, 2019.

[6] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[7] Charles Darwin and William F Bynum. *The origin of species by means of natural selection : or, the preservation of favored races in the struggle for life.* AL Burt New York, 2009.

[8] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of parallel computing*, volume 3003. Morgan Kaufmann Publishers San Francisco^ eCA CA, 2003.

[9] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.

[10] Xavier Gandibleux, Marc Sevaux, Kenneth Sörensen, and Vincent T'kindt. *Metaheuristics for multiobjective optimisation*, volume 535. Springer Science & Business Media, 2004.

[11] Fred Glover. Future paths for integer programming and links to ar tifi cial intelli g en ce. *Computers operations research*, 13(5) :533–549, 1986.

[12] Giovanni Iacca, Fabio Caraffini, and Ferrante Neri. Continuous parameter pools in ensemble differential evolution. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1529–1536. IEEE, 2015.

[13] Yoram Koren, Xi Gu, and Weihong Guo. Reconfigurable manufacturing systems : Principles, design, and future trends. *Frontiers of Mechanical Engineering*, 13(2) :121–136, 2018.

[14] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. Introduction to parallel computing. 2nd, 2002.

[15] Leslie Lamport. LaTeX. `https://en.wikipedia.org/wiki/LaTeX`, 2017. accessed on May 18, 2017.

[16] JJ Liang, Thomas Philip Runarsson, Efren Mezura-Montes, Maurice Clerc, Ponnuthurai Nagaratnam Suganthan, CA Coello Coello, and Kalyanmoy Deb. Problem definitions and evaluation criteria for the cec 2006 special session on constrained real-parameter optimization. *Journal of Applied Mechanics*, 41(8) :8–31, 2006.

[17] Rammohan Mallipeddi, Ponnuthurai N Suganthan, Quan-Ke Pan, and Mehmet Fatih Tasgetiren. Differential evolution algorithm with ensemble of parameters and mutation strategies. *Applied soft computing*, 11(2) :1679–1696, 2011.

[18] Maxime Martinasso. *Analyse et modélisation des communications concurrentes dans les réseaux haute performance*. PhD thesis, 2007.

[19] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4) :341–359, 1997.

[20] El-Ghazali Talbi. *Metaheuristics : from design to implementation*, volume 74. John Wiley & Sons, 2009.

[21] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4) :27–75, 1993.

[22] Yong Wang, Zixing Cai, and Qingfu Zhang. Differential evolution with composite trial vector generation strategies and control parameters. *IEEE transactions on evolutionary computation*, 15(1) :55–66, 2011.

[23] Guohua Wu, Xin Shen, Haifeng Li, Huangke Chen, Anping Lin, and Ponnuthurai N Suganthan. Ensemble of differential evolution variants. *Information Sciences*, 423 :172–186, 2018.

[24] Jingqiao Zhang and Arthur C Sanderson. Jade : adaptive differential evolution with optional external archive. *IEEE Transactions on evolutionary computation*, 13(5) :945–958, 2009.