SEP **TNM**

INSTITUTO TECNOLÓGICO DE TIJUANA

División de Estudios de Posgrado e Investigación

Optimización de Controladores Difusos Aplicando Algoritmos Bioinspirados en la Nube

Trabajo de tesis

Presentado por:

M.C. Alejandra Mancilla Soto

Para obtener el grado de:

Doctor en Ciencias en Computación

Director:

Dr. Oscar Castillo López

Co-Director:

Dr. José Mario García Valdéz

Tijuana, B.C. Mayo de 2024.

Tijuana, Baja California, 30/abril/2024
Oficio No. 070/DEPI/2024
Asunto: Autorización de Impresión de Tesis

MARÍA MAGDALENA SERRANO ORTEGA
JEFA DEL DEPARTAMENTO DE SERVICIOS ESCOLARES
PRESENTE

En lo referente al trabajo de tesis, "Optimización de controladores difusos aplicando algoritmos bioinspirados en la nube". Presentado por C. **Alejandra Mancilla Soto**, alumna del Doctorado en Ciencias en Computación con número de control **D20210003;** informo a usted que a solicitud del comité de tutorial, tengo a bien **Autorizar la Impresión de Tesis**, atendiendo las disposiciones de los Lineamientos para la Operación de Estudios de Posgrado del Tecnológico Nacional de México.

Sin más por el momento le envió un cordial saludo.

A T E N T A M E N T E
Excelencia en Educación Tecnológica•
Por una Juventud Integrada al Desarrollo de México•

EDUCACIÓN
INSTITUTO TECNOLÓGICO DE TIJUANA
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

GUADALUPE HERNÁNDEZ ESCOBEDO
JEFE DE DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

ccp. Archivo
GHE/lap

Tijuana, B.C., 29 de Abril de 2024
Asunto: Se autoriza impresión de Trabajo de Tesis

C. DRA. MARÍA MAGDALENA SERRANO ORTEGA
JEFA DEL DEPTO. DE SERVICIOS ESCOLARES
PRESENTE.

En lo referente al trabajo de tesis escrito, con título **"Optimización de controladores difusos aplicando algoritmos bioinspirados en la nube"**, presentado por la ***C. ALEJANDRA MANCILLA SOTO,*** alumna del Doctorado en Ciencias en Computación con número de control **D20210003**, informamos a usted que después de una minuciosa revisión, de acuerdo con lo establecido en el reglamento vigente para este caso, nuestro dictamen es: Se aprueba en todas sus partes, en virtud de reunir los requisitos de un trabajo de grado de Doctorado y a la vez se autoriza al interesado para que proceda de inmediato a la impresión del mismo.

ATENTAMENTE

DR. OSCAR CASTILLO LOPEZ
PRESIDENTE

DR. JOSE MARIO GARCIA VALDEZ
SECRETARIO

DR. FEVRIER ADOLFO VALDEZ ACOSTA
VOCAL

DRA. ELBA PATRICIA MELIN OLMEDA
VOCAL

DR. JUAN RAMON CASTRO RODRIGUZ
VOCAL

c.c.p. Oficina de Titulación
c.c.p. División de Estudios de Posgrado e Investigación
c.c.p. Expediente
c.c.p. Interesado

EPMO/*inf

Tijuana, B.C., 29 de Abril de 2024
Asunto: Se autoriza impresión de Trabajo de Tesis

C. DR. GUADALUPE HERNÁNDEZ ESCOBEDO
JEFE DE DIVISIÓN DE ESTUDIOS DE POSGRADO E INV.
PRESENTE.

En lo referente al trabajo de tesis escrito, con título " **Optimización de controladores difusos aplicando algoritmos bioinspirados en la nube**", presentado por el ***C. ALEJANDRA MANCILLA SOTO,*** alumna del Doctorado en Ciencias en Computación con número de control **D20210003,** informamos a usted que se autoriza el escrito de tesis y se aprueba en todas sus partes, en virtud de reunir los requisitos de un trabajo de grado de Doctorado y a la vez se autoriza al interesado para que proceda de inmediato a la impresión del mismo y a presentar su examen de grado, ya que cumple con todos los requisitos.

ATENTAMENTE

DR. OSCAR CASTILLO LOPEZ
PRESIDENTE

DR. JOSE MARIO GARCIA VALDEZ
SECRETARIO

DR. FEVRIER ADOLFO VALDEZ ACOSTA
VOCAL

DRA. ELBA PATRICIA MELIN OLMEDA
VOCAL

DR. JUAN RAMON CASTRO RODRIGUZ
VOCAL

c.c.p. Oficina de Titulación
c.c.p. División de Estudios de Posgrado e Investigación
c.c.p. Expediente
c.c.p. Interesado

EPMO/*inf

DECLARACIÓN DE ORIGINALIDAD

Tijuana, BC., 29 de abril de 2024,

Yo, **Alejandra Mancilla Soto**, estudiante del Doctorado en Ciencias en Computación, en mi calidad de autor manifiesto que este documento de tesis es producto de mi trabajo original y que no infringe los derechos de terceros, tales como derechos de publicación, derechos de autor, patente y similaridad. Por lo tanto, la obra realizada es de mi exclusiva autoría y no infringí en copiar el texto o imágenes, de fuentes de información por lo cual soy responsable del escrito que aquí se presenta.

Así mismo, declaro que en las citas textuales que he incluido (las cuales aparecen entre comillas) y en los resúmenes que he realizado de publicaciones ajenas, indico explícitamente los datos de los autores y las publicaciones.

En caso de presentarse cualquier reclamación o acción por parte de terceros en cuanto a los derechos de autor sobre la obra en cuestión, acepto toda la responsabilidad de tal infracción y relevo de esta a mi director de tesis, así como al Tecnológico Nacional de México, al Instituto Tecnológico de Tijuana y a sus respectivas autoridades.

Alejandra Mancilla Soto
Estudiante de Doctorado en Ciencias en Computación

CARTA DECLARACIÓN DE PROPIEDAD INTELECTUAL

Tijuana, BC a 30 de abril del 2024

Yo **ALEJANDRA MANCILLA SOTO** reconozco que el Trabajo de Tesis de Doctorado que realice durante mis estudios en el Doctorado en Ciencias en Computación del Instituto Tecnológico de Tijuana fue parte del Proyecto de Investigación Titulado: **DESARROLLO DE ALGORITMOS EVOLUTIVOS MULTI POBLACIONES APLICADOS A LA OPTIMIZACIÓN DE CONTROLADORES DIFUSOS No. 18186.23-P** que desarrolla mi Co-Director de tesis el **Dr. José Mario García Valdez** y del cual es responsable del proyecto de investigación. Por esta razón, los métodos, modelos, algoritmos, y software realizados, así como datos y resultados obtenidos durante el desarrollo de mi tesis de doctorado son propiedad intelectual de mi Co-Director de Tesis, del Tecnológico Nacional de México, Instituto Tecnológico de Tijuana, y No podré utilizarlos por mi cuenta durante, Ni después de terminar mi beca o estudios, excepto a solicitud escrita para poder utilizarlos bajo una colaboración directa con mi co-director de tesis el cual es responsable del proyecto de investigación. Por tanto, estoy de acuerdo en que No podre utilizar ni tomar modelos, ni datos utilizados en este proyecto de investigación y en el desarrollo de tesis para: presentaciones, publicaciones ni desarrollo de mi propia investigación que pudiera desarrollar una vez concluidos mis estudios**.**

Atentamente

Alejandra Mancilla Soto

Estudiante del Doctorado en Ciencias en Computación
Instituto Tecnológico de Tijuana

Acknowledgments

This work is for all those people who trusted my effort and dedication to do this work and who were in some way supporting me at every moment.

I especially want to thank my family, parents, my husband Mario, and children Andrea and Mario Alejandro, who are my pillars and motivation.

To my friends who always motivated me and gave me their support. To my fellow graduate students who were always listening to my presentations.

I especially want to thank my advisors, Dr. Oscar Castillo López and Dr. José Mario García Valdez, for their effort and patience during these years of my graduate studies. Thank you for sharing your knowledge and guide on the research. Today, I have more tools to face upcoming challenges. Thank you so much.

A thank you to Dr. Elba Patricia Melin Olmeda for her management during my studies.

Finally, I extend my gratitude to the Tecnológico Nacional de México (TecNM) and the Instituto Tecnológico de Tijuana for providing the necessary resources for the development of this thesis.

Abstract

In the domain of autonomous systems, optimizing control mechanisms to achieve high precision and adaptability in path tracking remains a significant challenge. This research delves into applying multi-population-based metaheuristics to optimize fuzzy system controllers for path tracking, proposing a cloud deployment. Through a comprehensive analysis, we explore the applicability of evolutionary and swarm intelligence algorithms, for these tasks. A central point of our work is developing a novel approach for dynamic parameter adaptation for multi-population algorithms. This methodology aims to enhance fuzzy controllers and significantly speed up the execution time associated with parameter tuning. Our experimental framework evaluates the proposed approach across various scenarios, benchmarking against traditional optimization methods. The results reveal a marked improvement in the controllers' performance metrics, including response time, accuracy, and adaptability to dynamic environments. Our findings underscore the potential of utilizing cloud-native implementations of population-based algorithms on a cloud platform. This option is financially viable, eliminating the need for additional hardware investments and administrative expenses. This research contributes to the field by providing a robust framework for optimizing fuzzy system controllers using bioinspired algorithms. Our findings underscore the potential of bioinspired optimization in improving the capabilities of autonomous systems, paving the way for future innovations in this rapidly evolving field.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most engineering disciplines require optimization methods to solve complex problems involving many decision variables and very complex objectives. In the case of computer science, we often need to tune the parameters of algorithms and computational models, to increase their performance and reduce the error of the outputs. One of these cases is tuning the parameters of model-based controllers. For some years now, computer scientists have applied fuzzy logic theory to control problems, also needing optimization techniques [1, 2] to fine-tune the parameters of fuzzy controllers to increase their performance. Furthermore, depending on the type of controller, the search space can be of a considerable size, making it unfeasible to manually test all available options [3].

Due to their complexity and iterative nature, optimizing controllers using bio-inspired algorithms incurs a high computational cost. Researchers are actively exploring various strategies to mitigate these costs and reduce execution time. We found many works and techniques applied to these problems in the literature revision [2]. Still, we found that these were applied without considering the current industrial software development trends favoring cloud-native architectures. Even distributed proposals based on multi-populations often neglect using event-driven architectures and tools that nowadays are a common technique for scaling systems. Issues like migration between populations, dynamic parameter adaptation, and multi-population parametrization consider traditional distributed implementations. Few works in the

literature specify distributed algorithms and bioinspired techniques designed specifically to run in a cloud environment.

With the goal of enhancing the parameters of a fuzzy controller, this thesis proposes a strategy based on multi-populations, inspired by works like EvoSwarm [4], EvoSpace [5, 6], SofEA [7], and other cloud-based bioinspired algorithms. These types of algorithms have additional challenges. For instance, having more populations increases the number of parameters we need to set for each population. Next, we state the research problem we address in this work.

1.1 Research Problem

As we have seen in the literature, there are very few works that demonstrate favorable results in the optimization of controllers using metaheuristics in a distributed and asynchronous manner, so our contribution is the design, development, and implementation of a distributed and asynchronous architecture, where we use different metaheuristics and implement them following cloud-native principles to improve their performance and execution time.

The general objective is to propose parameterization techniques for distributed bioinspired optimization algorithms applied to optimizing fuzzy controllers.

The Specific research objectives:

- Apply distributed bioinspired optimization algorithms to solve benchmark control problems.
- Evaluate different parameterization techniques of distributed bioinspired optimization algorithms for optimization problems.
- Evaluate techniques for optimizing fuzzy controllers.
- Implement different distributed optimization metaheuristics using cloud computing platforms.
- Compare the statistical results obtained from our proposal with other fuzzy controller optimization metaheuristics.

Throughout this work, the trajectory tracking problem is chosen as the use case, since it is a resource-intensive optimization problem. The trajectory tracking model employed in this study is based on the algorithm published in the review of Paden et al. [8], which is a significant reference in the field. By building on this foundational work, we optimize the trajectory tracking using a kinematic model.

This work includes the development of mathematical models and control routines, all of which are implemented in Python. This programming choice is strategic, given Python's robust libraries and frameworks that support mathematical and scientific computations, making it an ideal environment for simulating complex systems.

1.2 Hypothesis

We summarize the primary research questions into a testable hypothesis:

Implementing a distributed, asynchronous metaheuristic optimization framework using cloud-native technologies and fuzzy logic for parameter adaptation significantly improves the computational efficiency and performance of fuzzy controllers in trajectory tracking problems compared to traditional single-population, synchronous metaheuristic approaches.

1. **Variables:**
    - Independent Variable(s): The type of optimization framework used (distributed and asynchronous metaheuristics with fuzzy logic parameter adaptation vs. traditional single-swarm synchronous approaches).
    - Dependent Variable(s): Computational efficiency (measured by execution time) and performance (measured by the accuracy of trajectory tracking using RMSE).
2. **Population:**
    - Fuzzy controllers applied to trajectory tracking problems.

- Optimization frameworks.

This hypothesis stems from the objectives of applying distributed bioinspired optimization algorithms and exploring different parameterization techniques. It also incorporates the specific architectural and methodological enhancements: cloud-native implementations and fuzzy parameter adjustments.

To test the hypothesis, we follow the following methodology:

1. **Experimental Design:** We compare the outcomes (both computational efficiency and trajectory accuracy) of fuzzy controllers optimized using our proposed distributed framework against those optimized with traditional methods.

2. **Data Collection:** We collect quantitative data on execution times and RMSE values from multiple runs of each algorithm configuration across simulated trajectory tracking scenarios.

3. **Statistical Analysis:** We use a Z-test for comparing means to determine if differences in performance and efficiency metrics are statistically significant, thereby supporting or refuting the hypothesis.

1.3 Contribution of this thesis

This body of work is anchored in the belief that the fuzzy logic controller optimization frontier can be significantly expanded by integrating cutting-edge computational techniques and methodologies. Our contributions are manifold and significant, spanning several key areas in the field of bioinspired optimization and fuzzy systems:

1. **Novel Implementation of Distributed Bio-Inspired Algorithms**: We have extended the fuzzy logic controller optimization literature by exploring the application of distributed bio-inspired algorithms in this field.

2. **Enhanced Fuzzy Controller Design for Autonomous Robots**: Through the utilization of population-based metaheuristics, we have optimized the design of a fuzzy path-tracking controller for a bicycle-like robot, improving the performance in path tracking—a critical task in robotics.

3. **Exploration of Random Parameters in Distributed Bioinspired Methods**: Our research introduces employing random parameters within distributed bio-inspired methods for fuzzy controller optimization, presenting a novel approach to reducing execution time while maintaining or improving performance metrics.

4. **Distributed and Asynchronous Optimization Techniques**: We propose a distributed and asynchronous bioinspired algorithm applying cloud-native patterns for parallel simulation execution. This work highlights the speedup and efficiency gains achievable through such technologies.

5. **Fuzzy Adaptation of Algorithm Parameters in Multi-Swarm PSO**: By integrating a fuzzy inference system into a multi-swarm PSO algorithm, we offer a sophisticated method for dynamic parameter adjustment, showcasing the potential for significant improvements in optimization processes.

6. **Cloud-Native Optimization Framework for Fuzzy Controller Optimization**: Our contribution in this area includes the development of a cloud-native framework that supports multiple population-based algorithms, demonstrating the scalability and performance benefits of such an approach in a cloud environment.

Each contribution is related to the publications listed below:

1. **Optimization of Fuzzy Logic Controllers with Distributed BioInspired Algorithms**

   In [2], we present a review of the state of the art; we analyze different works in the literature on applying bioinspired methods to tune the parameters for

fuzzy controllers. We find works using distinct parameter settings and algorithms and the type of implementation regarding the sequential or distributed execution. We found many works that use dynamic parameter adaptation, hybrid parameters, distributed processing, and other types of fuzzy logic, but few works in the literature specify distributed algorithms, but not in the same way as we are proposing, which, apart from being distributed and asynchronous, is designed to be run in a cloud environment.

2. **Optimal Fuzzy Controller Design for Autonomous Robot Path Tracking Using Population-Based Metaheuristics**

   In [9], we proposed, through the application of bioinspired algorithms, the tuning of a fuzzy controller. This allows the design of a controller using linguistic rules familiar to most users. We propose an evaluation method of each controller's performance using three different paths. We optimized the controller using popular bioinspired algorithms. When compared to published results, the experiments validate that tuned fuzzy controllers have better performance measured by the RMSE. A comparison of the best-optimized controller with the control law, GA, and PSO of each route used is also presented, where we see how the GA was better in two routes than the control law, but the PSO algorithm was the best in the three routes.

   The experiments highlight a critical issue in the current practice of fuzzy controller optimization, specifically the reliance on a singular problem case and performance metric. This approach can inadvertently lead to controllers that are overtrained, meaning they perform exceptionally well on the specific case and metric they were trained on but potentially lack the generality to perform equally well across a broader range of scenarios or different types of challenges.

3. **Optimization of Fuzzy Controllers using Distributed Bioinspired Methods with Random Parameters**

   In [10], we identified that one problem with multi-population optimization algorithms is finding the ideal configuration we will use to run the algorithms, such as the number of populations and processors needed, and setting the pa-

rameters affecting the emphasis on exploration and exploitation. We compare homogeneous configurations for all populations with heterogeneous configurations in which we randomly define the parameters. We want to evaluate whether this strategy helps us minimize evaluation time and minimize RMSE. We will use GA and PSO algorithms in this experiment. The results suggest that we obtain an RMSE similar to the previous results by using random parameters in our algorithms. Furthermore, we observed a reduction in execution time.

4. **Distributed and Asynchronous Population-Based Optimization Applied to the Optimal Design of Fuzzy Controllers**

   In [11], we propose a distributed and asynchronous bio-inspired algorithm that operates in parallel across multiple populations, employing a cloud-native pattern to facilitate asynchronous interactions through a distributed message queue. This method prevents downtime associated with synchronization cycles among nodes, enhancing efficiency. The proposed architecture has the ability to combine different metaheuristics, one for each population. We evaluated both sequential and distributed implementations. Additionally, the paper explores both homogeneous and heterogeneous configurations of population parameters. Remarkably, even with randomly configured heterogeneous parameters within the distributed populations, the results demonstrate a comparable level of error to other studies, while significantly cutting down execution times.

5. **Fuzzy Adaptation of Parameters in a Multi-Swarm Particle Swarm Optimization (PSO) Algorithm Applied to the Optimization of a Fuzzy Controller**

   In [12], we propose a multi-swarm PSO using a fuzzy inference system (FIS) to adapt the parameters at runtime, considering the population diversity and current number of iterations. We obtain the current parameters as the FIS output, these values set the social and cognitive coefficients used in the next iteration. We evaluated if the proposal reduced the evaluation time and the controller's RMSE. The experiments suggest that the newly proposed method

for tuning the algorithm's parameters yields an RMSE comparable to those reported in the literature.

1.4 Outline

This thesis is structured as follows: in chapter 1 we present the introduction to the topic of this thesis work, in chapter 2 we present the theoretical framework and the study of art, in the chapter 3 present the control problem, in the chapter 4 we show the architecture proposed and the chapter 5 and 6 details the research and findings that contribute to the field. The final chapter (Chapter 7) summarizes the conclusions of the thesis and suggests future research directions. Appendices provide additional context: *Appendix A* catalogs the research published during the thesis. *Appendix B* lists the tools and technologies used, including Python libraries and platforms, with relevant links. The document concludes with a complete bibliography that covers all the references cited throughout the study.

Chapter 2

Literature review

The development of model-based control (MBC) in the late 1960s, gave shape to Modern control. MBC typically utilizes nonlinear differential equations to accurately represent the dynamics of systems [13]. This approach involves first establishing a comprehensive mathematical model of the system—often called the 'plant'—and then crafting a controller specifically tailored to manage this model effectively.

The early 1970s marked a significant paradigm shift in control systems with the advent of Intelligent Control (IC) [14]. This new approach diverged from traditional model-based systems by avoiding reliance on mathematical models. Instead, IC uses human knowledge, operational research, and empirical evidence to formulate control actions.

Fuzzy Logic Control (FLC) has established itself as a remarkably successful intelligent control technique, evolving from the pioneering work of Mamdani [15] to its widespread application in contemporary settings [16]. Parallel to this, Data-driven control has emerged as a rapidly advancing alternative. This approach encompasses a wide array of techniques where both the mathematical model and the controller design are derived exclusively from data sets collected from the process under control.

Data-driven control strategies include reinforcement learning [17], robust control [18], and iterative learning control (ILC) [19]. These methods are often integrated with advanced computational tools like neural networks and genetic algorithms [20],

offering robust solutions across various applications.

A key advantage of Fuzzy Logic Controllers (FLC) is that they are human-readable and allow for both automatic and manual tuning. Similar to model-based controllers, designers of Intelligent Control (IC) systems are also required to finely optimize the controller parameters. This optimization is crucial to effectively handle the complexities encountered in real-world applications and is an important aspect of our research.

This chapter presents a concise review and discussion of the principal research domains pertinent to this work. First, in Section 2.1, we discuss the fundamental concepts and design issues of Fuzzy Inference Systems and their application in control. Next, a review of the area of bioinspired algorithms is presented in Section 2.2, in this section, the algorithms used in this work are presented in detail. Then, we present the subject central to this work, Distributed Multipopulation Algorithms in Section 2.3, Parallel GAs and Cloud Computing in Section 2.4.

2.1 Fuzzy Inference Systems as Controllers

Fuzzy control is implemented by creating a knowledge base that models imprecise knowledge and data in a language similar to natural language. Using fuzzy rules and variables, a Fuzzy Inference System (FIS) establishes the relationship between the controller's inputs and the corresponding outputs. From this perspective, Babuvska [21] characterizes these systems as "flexible mathematical functions which can approximate other functions or just data (measurements) with a desired accuracy".

The knowledge in a FIS is expressed as a set of Fuzzy rules. Fuzzy rules are formulated as "IF antecedent THEN consequent," where the antecedent is a proposition expressed as "x is A." Here, x represents a linguistic variable, and A is a linguistic term. These propositions employ fuzzy logic sets to define the variables and terms involved. The truth value of such a proposition corresponds to the degree of match between x and A. Linguistic variables (LVs) accept linguistic terms as values. For instance, the linguistic variable AGE might assume values such as $CHILDREN$, $TEENAGER$ or $ADULT$. The definitions of these terms are encapsulated in their

respective membership functions (MFs).

Each linguistic variable is characterized by a quintuple, LV=< v, T, X, g, m >, where v is the variable's name, T the set of its linguistic terms, X its domain, g a syntactic rule to generate linguistic terms, and m a semantic rule that maps each term t to its meaning $m(t)$—a fuzzy set within X.

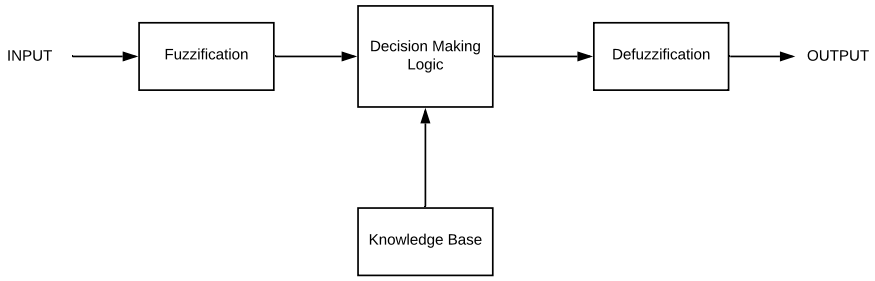


Figure 2.1: Fuzzy Inference System Block Diagram

Key elements of a FIS are depicted in Figure 2.1, as outlined in the referenced work [22]. We start from the left of the figure. The INPUT captures real-time feedback from the device during its operation, while the OUTPUT regulates the actuator within the device. Initially, the Knowledge Base is established with a set of fuzzy rules, as we described before, along with the definition of the Membership Functions. The Fuzzifier transforms numerical inputs into linguistic values defined as fuzzy logic terms. Taking this data as input, the Decision Making Logic executes the inference operations, applying the rules specified in the Knowledge Base to the linguistic inputs to derive fuzzy outputs. Finally, the Defuzzifier module converts these fuzzy outputs back into crisp numerical data, providing actionable outputs that can directly control system actuators or influence other system components.

Designing a basic fuzzy system might seem straightforward—essentially translating knowledge into fuzzy rules—but in reality, it is a complex engineering process involving several detailed steps. Initially, Problem Identification is crucial, where the most suitable type of fuzzy system for handling the data is determined. Following this, we must define the input and output variables, their fuzzy values, and their respective membership functions (Variable Parameterization). The third step

(Knowledge Base Definition), establishes the set of fuzzy rules that form the system's knowledge base. Next, the Fuzzy Inference process derives system outputs from the input variables using the fuzzy inference system. After inference, Defuzzification is required to convert the fuzzy output into a precise, or crisp, value. The final step, called System Tuning, involves adjustments and validation of the system to ensure its accuracy and reliability in real-world applications.

The design of a FIS requires a comprehensive understanding of the system's behavior, particularly in controller design. The tuning phase is crucial and typically involves conducting simulations or practical testing on actual devices. Fine-tuning can be achieved through several methods, such as increasing the granularity of variables by adding more MFs, adjusting the MF's parameters, adding or removing rules, or incorporating more input variables. Tweaking the input parameters of membership functions is a straightforward and commonly employed method to enhance the performance of a fuzzy controller, as widely documented in the literature [23].

Another critical decision involves choosing the type of FIS, dictated by the kind of fuzzy logic used. In scenarios with significant noise, a type-2 fuzzy logic system may offer superior performance [3]. However, implementing a type-2 system introduces greater complexity. Historically, researchers have applied various optimization algorithms to navigate this parameter space, aiming to identify sub-optimal yet effective settings for fuzzy controllers. In the next sections, we present a review and current advances on the field of bioinspired algorithms and their application to the enhancement of fuzzy control.

2.2 Bioinspired Algorithms

Bioinspired algorithms take inspiration from natural systems to design nondeterministic metaheuristics useful for optimization, learning, pattern recognition, and classification [24, 25]. Typically used in optimization, bioinspired algorithms represent all possible solutions to a problem as points within a solution space, where each point corresponds to a candidate solution. As a first step, a random population of

candidate solutions is created, then each candidate solution is evaluated using a fitness function that assesses its quality. Solutions with higher fitness scores indicate local optima in the solution space. Algorithms navigate the search space guided by these scores in a stochastic manner. However, their probabilistic nature can sometimes overlook both local and global optima, leading to potential stagnation at suboptimal points. To mitigate this, these algorithms apply two main strategies: exploration and exploitation. Exploration broadly searches the solution space to prevent entrapment in local optima, while exploitation focuses on refining locally promising solutions [26]. Bioinspired algorithms are considered to be metaheuristics as they do not need information about the optimization problem at hand. These algorithms normally do not use a gradient to guide the search for optimal solutions; which means, they do not require the problem to be differentiable.

This work focuses on bioinspired population-based metaheuristics for structural optimization [27, 28, 29, 30]. These algorithms are categorized into evolutionary algorithms (EAs), based on natural selection [31], swarm intelligence (SI) inspired by the social and cooperative behavior found in a bird flock [32], and other types. Notable EAs include Genetic Algorithms (GAs) [33, 34], Genetic Programming (GP) [31], and Differential Evolution (DE) [35]. Examples these algorithms include Particle Swarm Optimization (PSO) [36], Grey Wolf Optimization (GWO) [37], and Aquila Optimizer (AO) [38]. Other categories feature innovative approaches like the Harmony Search (HS) [39], inspired by jazz improvisation, and the Arithmetic Optimization Algorithm (AOA) [40].

For this thesis, we selected three representative metaheuristics to validate our proposal and these have demonstrated significant efficiency in parameter optimization [41]: Genetic Algorithms (GAs), Particle Swarm Optimization (PSO), and Grey Wolf Optimization (GWO). We now explain each algorithm in more detail.

2.2.1 Genetic Algorithms

GAs, as described by Holland [42], are the canonical population-based metaheuristic. Drawing inspiration from the natural selection process, these algorithms operate on a collection of potential solutions known as a population. This population un-

dergoes evolution over numerous generations until a satisfactory or optimal solution emerges [43]. Within each generation, solutions are assessed; the fittest survive and reproduce through genetic mechanisms such as crossover (exploration) and mutation (exploitation), producing new offspring. A replacement mechanism then selects the most adapted offspring to form the next generation. Typically, these metaheuristics initiate with a randomly generated collection of potential solutions, which are evaluated against a performance metric. A nature-inspired algorithm is applied to generate new candidates based on each solution's quality and structural attributes.

2.2.2 Particle Swarm Optimization

Unlike Genetic Algorithms (GAs) that involve genetic operations and selection processes, PSO simulates a continuous flow of particles across the search landscape.

In PSO, particles are initialized randomly within the search space and move according to both their personal experiences and the collective knowledge of the swarm. This knowledge reflects the quality of solutions, akin to fitness in GAs. Crucially, particles are not replaced between iterations but can adjust their positions based on interactions and shared information within the swarm, allowing them to identify local or global optima effectively.

Each particle in the swarm retains information about the most favorable position it has discovered (personal optimum) and the best position identified by its neighbors (global optimum), enhancing the algorithm's capability to converge on optimal solutions. This dynamic adjustment of position and velocity with each iteration is informed by both local and global discoveries, resembling the way birds adjust their flight based on the movements of their flock and its leaders.

These are the main steps in a PSO algorithm:

1. **Create Particles**.

   Each particle is defined by a position, velocity, and value, which evolve as the particle traverses through the search space. Additionally, a particle keeps a record of the best position it has encountered during its journey. This mechanism allows for ongoing assessment and adaptation based on the particle's

experiences, optimizing its path toward the solution. Upon the creation of a new particle, it is assigned initial values for its position and velocity, with the velocity typically set to zero. The other attributes of the particle are determined after its first evaluation. This process allows the system to gauge the particle's initial effectiveness in the search space and adjust its trajectory and velocity accordingly based on performance metrics.

2. **Evaluation**.

   The evaluation process for a particle involves applying the objective function to the values representing the particle's current position. If the outcome of this evaluation yields the best value that the particle has encountered thus far, this position is then recorded as the particle's best position. This mechanism ensures that each particle continuously optimizes its path based on the most successful positions it has discovered over time.

3. **Particle Movement**.

   A particle moves to a new position by updating its velocity and subsequently adjusting its position based on this new velocity. This movement allows the particle to navigate through the search space, aiming to converge towards optimal solutions by using its personal best experiences and the collective knowledge of the swarm.

The following equation updates the velocity 2.1:

$$v_i\left(\mathbf{t}+1\right) = \mathbf{w}v_i(\mathbf{t}) + c_1 r_1\left[\hat{x}_i\left(\mathbf{t}\right) - x_i\left(\mathbf{t}\right)\right] + c_2 r_2\left[g\left(\mathbf{t}\right) - x_i\left(\mathbf{t}\right)\right] \tag{2.1}$$

where:

$v_i\left(\mathbf{t}+1\right)$: velocity of particle **i** at time $(\mathbf{t}+1)$, this is the new velocity.

$v_i(\mathbf{t})$: velocity of particle **i** at time **t**, this is the current velocity.

w: inertia coefficient reduces or increases the particle's velocity.

c_1: cognitive coefficient.

r_1: random vector in [0,1] with the same length as the velocity vector.

$\hat{x}_i(\mathbf{t})$: the best position that particle **i** has been in so far.

$x_i(\mathbf{t})$: position of particle **i** at time **t**.

c_2: social coefficient.

r_2: random vector in [0,1] with the same length as the velocity vector.

$g(\mathbf{t})$: the best position in all the swarm at time **t**, this is known as the global best.

Once we establish the new velocity, we can update the particle's position with this equation 2.2:

$$x_i(\mathbf{t}+1) = x_i(\mathbf{t}) + v_i(\mathbf{t}+1)\,\mathrm{T} \tag{2.2}$$

where T is a constant in time T=1

2.2.3 Grey Wolf Optimization Algorithm

Inspired by the leadership hierarchy and hunting mechanism of grey wolves in nature, Mirjalili [37] proposed the Grey Wolf Optimizer (GWO). The algorithm considers that grey wolves prefer to live in groups known as packs, each consisting of approximately 5 to 12 members. The novelty of the algorithm is that, in this case, every individual within the group adheres to a strict social dominance hierarchy (Figure 2.2), this behavior is not considered by the canonical GA or PSO algorithms.

The GWO hierarchy consists of the following levels:

The dominant wolf in the pack is the alpha wolf α and commands the obedience of all other members. The beta wolves β are subordinates supporting the alpha in decision-making and are viewed as potential successors to the alpha position. The delta wolves δ must submit to both the alpha and beta but submit the omega. Delta wolves fulfill various roles within the pack, including scouts, sentinels, elders, hunters, and caregivers. At the bottom of the hierarchy are the omega wolves ω

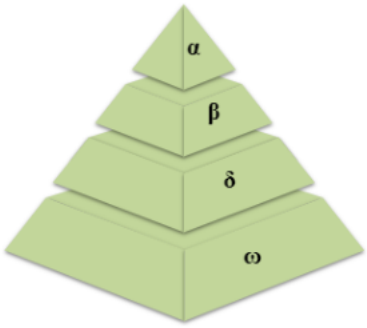


Figure 2.2: GWO Hierarchy.

considered the scapegoats of the pack, being the least important members. Omegas are typically the last to eat, only after all other members have had their share.

The GWO algorithm uses the following steps inspired by the hunting behavior of gray wolves [37].

- Tracking, chasing, and approaching the prey.
- Chase, surround, and harass the prey until it stops moving.
- Attack towards the prey.

The formula for updating the position of a gray wolf is as follows 2.3:

$$\mathrm{X}_i^d(t+1) = \mathrm{X}_p^d(t) - \mathrm{A}_i^d \left| \mathrm{C}_i^d \mathrm{X}_p^d(t) - \mathrm{X}_i^d(t) \right| \tag{2.3}$$

where:

t: iteration number.

X_i: the position vector of a grey wolf.

X_p: the position vector of the prey.

$\mathrm{A}_i^d = 2a \cdot r_1 - a$

$$\mathrm{C}_i^d = 2 \cdot r_2$$

where:

r_1 and r_2 are random vectors in [0,1].

a is a variable that linearly decreases from 2 to 0, i.e., $a(t) = 2 - 2t/maxIter$, where $maxIter$ is the maximum number of iterations.

We use the following equation to update the position of alpha wolves 2.4:

$$\mathrm{X}_{i,\alpha}^d(t+1) = \mathrm{X}_\alpha^d(t) - \mathrm{A}_{i,1}^d \left| \mathrm{C}_{i,1}^d \mathrm{X}_\alpha^d(t) - \mathrm{X}_i^d(t) \right| \tag{2.4}$$

The position beta wolves is dictated by 2.5:

$$\mathrm{X}_{i,\beta}^d(t+1) = \mathrm{X}_\beta^d(t) - \mathrm{A}_{i,2}^d \left| \mathrm{C}_{i,2}^d \mathrm{X}_\beta^d(t) - \mathrm{X}_i^d(t) \right| \tag{2.5}$$

This is the equation that dictates how deltas move 2.6:

$$\mathrm{X}_{i,\delta}^d(t+1) = \mathrm{X}_\delta^d(t) - \mathrm{A}_{i,3}^d \left| \mathrm{C}_{i,3}^d \mathrm{X}_\delta^d(t) - \mathrm{X}_i^d(t) \right| \tag{2.6}$$

Finally, ω wolves should update its position following their superiors α, β, and δ with the following equations 2.7.

$$\mathrm{X}_i^d(t+1) = \frac{\mathrm{X}_{i,\alpha}^d(t+1) + \mathrm{X}_{i,\beta}^d(t+1) + \mathrm{X}_{i,\delta}^d(t+1)}{3} \tag{2.7}$$

Compared to other optimization algorithms, GWO requires the tuning of fewer parameters. This simplicity in parameter tuning significantly reduces the effort and time needed to adapt the algorithm to different problems. The GWO algorithm has been successfully applied to a wide range of optimization problems, including but not limited to, engineering design optimization, feature selection, renewable energy optimization, and scheduling problems. Its versatility is one of its most significant advantages [37].

2.2.4 Tuning Fuzzy Controllers using Bioinspired Algorithms

In this section, we explore recent advancements in the optimization of fuzzy controller parameters using bioinspired techniques.

In one of the earlier works, Balcucho et al. [44] used a GA and simulated annealing to improve the design of fuzzy controllers by tuning their membership rules and functions.

A novel hybrid approach combining Particle Swarm Optimization (PSO) and Genetic Algorithms (GA) for the design of fuzzy logic controllers (FLC) used for the trajectory tracking of autonomous mobile robots was proposed by Martinez-Soto et al. [45]. This hybrid method leverages the strengths of both PSO and GA to efficiently determine the optimal parameters for the membership functions of the FLC, thereby achieving enhanced control performance in guiding the robots along their designated paths.

In a related effort, Hernández et al. [46] employed the Grey Wolf Optimizer (GWO), to optimize fuzzy controllers for a two-wheeled autonomous mobile robot. This approach underscores the utility of bio-inspired algorithms in refining control systems for robotics.

In their work, Lagunes et al. [47] developed a methodology using the Firefly algorithm for optimizing the parameters of membership functions in a fuzzy controller specifically designed for a mobile autonomous robot. This optimization aims to enhance the performance of the robot's actuators, enabling more precise and efficient control. By fine-tuning the membership functions within the fuzzy control system, the researchers were able to achieve improved responsiveness and accuracy in the robot's movements.

In a recent study, by Arostegui et al. [48] applied ACO to enhance the membership functions of a non-optimized Fuzzy PD + I controller. This approach was then utilized to control the heat exchanger within a pasteurization plant. The performance of this bio-inspired algorithm was evaluated by comparing it with the results obtained from traditional models of a PID controller and a non-optimized FLC, showcasing the potential improvements in control and efficiency that such biological behavior-based algorithms can offer.

In their study, Oscar Castillo et al. [49] evaluated the performance of two bioinspired optimization methods: Ant Colony Optimization (ACO) and Gravitational Search Algorithm (GSA). They introduced a dynamic parameter adaptation ap-

proach to improve the performance of both algorithms. This adaptation involves dynamically altering certain algorithm parameters in each iteration, thereby modulating the balance between exploitation and exploration within the search space. The adjustments were based on several execution metrics, such as the elapsed iterations' percentage or the population's diversity.

The researchers utilized a type-2 fuzzy logic decision system to implement these dynamic adaptations. This system adjusted the algorithms' parameters during each iteration, aiming to refine their search process. The primary application of these enhanced algorithms was to tune a fuzzy controller designed to minimize errors in simulations involving complex nonlinear plants. Specifically, they focused on optimizing the membership function parameters for the input variables of the controller, thereby enhancing its performance.

In this work, we are interested in the dynamic adaptation of the parameters to enhance the bioinspired algorithms used in fuzzy control. The next works are related to this subject:

Castillo et al. [41] propose a type 1 fuzzy logic system (T1FLS) for the dynamic modification of the main parameters for three algorithms BCO, DE, and HS. In this work, they conduct a detailed analysis and comparison of both the modified and original versions of these algorithms, with a particular emphasis on fuzzy controller design. Statistically, their findings indicate that the enhanced versions of these systems, which incorporate fuzzy logic, exhibit superior error performance.

Bernal et al. [50] optimized the Galactic Swarm Optimization algorithm (GSO) by dynamically adjusting two of its parameters using a fuzzy system. They evaluated the fuzzy-enhanced GSO on a suite of mathematical benchmark functions and applied it to the fuzzy controller of the water tank problem. Their study concluded with a comparative analysis of the results, showcasing how the proposed method performs relative to other metaheuristics.

Peraza et al. [51] explored the improvement of the original Harmony Search (HS) algorithm by integrating fuzzy logic, resulting in the development of the Fuzzy Harmony Search (FHS) algorithm. The enhancement involves the dynamic adaptation of key HS parameters—Harmony Memory Acceptance (HMR) and Pitch Adjust-

ment Rate (PArate)—using a fuzzy system. The fuzzy system's rules are designed to manage the search process's intensification and diversification. This method was rigorously tested on a suite of mathematical functions from the CEC 2017 competition, encompassing a broad range of problem types, including unimodal, multimodal, hybrid, and composite functions. These tests aimed to validate the efficiency of the FHS algorithm in handling various optimization challenges.

Table 2.1: Review of the state of the art.

Reference	Bioinspired Algorithms	Parameter Adaptation
Arostegui [48]	ABC	No
Balcucho [44]	GA, SA	No
Bernal [50]	GSO	No
Castillo (2018) [49]	ACO, GSA	Yes-FL
Castillo (2019) [41]	BCO, DE, HS	Yes
Hernández [46].	GWO	No
Lara-Lagunes [47]	Firefly	Yes
Martinez-Soto [45] [41]	PSO, GA	Yes
Peraza [51]	HS	Yes

In this Table 2.1, we highlight the key trends in the application of bio-inspired metaheuristics for optimizing fuzzy controllers. A recurrent theme across these studies is the focus on fine-tuning the parameters of membership functions within fuzzy controllers to enhance their performance. Another significant trend is the emphasis on refining the underlying search algorithms themselves. The dynamic adaptation of parameters, which governs the balance between exploration (searching new areas) and exploitation (focusing on known good areas), emerges as a common strategy to improve optimization outcomes. This approach allows algorithms to heuristically navigate through the search space, leading to faster convergence and potentially finding more appropriate solutions. Moreover, there is a growing interest in presenting combinations of metaheuristics. Despite these advancements, the review identifies a gap in the current landscape: Currently, there is a lack of distributed algorithms or parallel implementations using a multi-population approach. This gap suggests an opportunity for future research to explore the potential of distributed computing and parallel processing techniques in enhancing the scalability and speed

of bio-inspired optimization algorithms. We explore the literature of this approach in the next section.

2.3 Distributed Multi-population Algorithms

We determined that adjusting the parameters of MFs of fuzzy controllers through population-based metaheuristics requires significant computational resources because establishing the quality of each potential solution needs the execution of multiple simulations [52, 9]; this is a challenge inherent to population-based metaheuristics. On the other hand, evolutionary algorithms evaluate the fitness of each candidate solution independently; this means we can use parallel processing to speed up the execution time. In our review, we only found a handful of attempts to distribute the tuning of fuzzy controllers; even more, any of these works considers or takes advantage of recent cloud technologies [53, 54, 55].

The majority of the research we mentioned above focuses on the parallel evaluation of candidate solutions. This emphasis is due to the fact that this is typically the most computationally demanding step of the optimization algorithm. We call this approach *global parallelization* [56]. An important property of this implementation is that any individual has the potential to mate with any member of the population, keeping the same panmictic-like properties of a single global population. In this approach, only the fitness evaluation step is done in parallel, and most implementations adhere to conventional leader/worker architectures.

On the other hand, multi-population or island models, divide the original population into several demes or subpopulations, each capable of independently running a Genetic Algorithm (GA) in parallel. Migration plays a crucial role in these parallel strategies because it has an impact on how fast the solution is found. Migration dictates the frequency and selection of those individuals that will be exchanged between populations. These multi-population methods not only have the advantage of speeding the execution time but, as experimental evidence suggests, also prevent premature convergence to local optima by preserving a higher diversity throughout the multi-populations [57]. Starkweather et al. [58] found that distributed genetic

algorithms often outperform their single-population counterparts, though this is not universally the case. Their findings highlighted the impact of different migration strategies on performance.

Multi-population approaches are not only found in GAs, they are also found in other population-based metaheuristics, for instance, there are many works in the literature are dedicated to developing and enhancing multi-swarm implementations of the PSO algorithm [59]. In PSO, researchers put more focus on the topology of communication between swarms, since the algorithm is dependent on the position, velocity, and distance between particles, the position of each swarm with respect to the others becomes an important design decision. An important aspect in the design of a Multi-Swarm PSO (MS-PSO) is that now we can have different configurations in each swarm, changing the parameters to finely tune the balance between exploration and exploitation phases, a topic thoroughly investigated in this field. In fact, this is true for all population-based metaheuristics.

Furthermore, multi-populations also bring the possibility of running entirely different metaheuristics in each population; recent works show that this can also benefit the system's overall performance [60, 61]. A critical aspect when designing multi-population-based algorithms is the coordination among nodes running these algorithms in parallel. Research in Parallel Genetic Algorithms (PGAs) has delved into both synchronous and asynchronous modes of parallelization. Synchronous parallelization, exemplified by the controller/worker model, requires the controller to wait for all workers to complete their tasks before proceeding. In contrast, asynchronous parallelization allows processes to continue independently without waiting for others, enhancing scalability and reducing total execution time. Comparative studies generally favor asynchronous methods for their speed.

2.4 Parallel GAs and Cloud Computing

Cloud computing is becoming a standard way of running computer science experiments. This shift is largely attributed to its cost-effective, pay-as-you-go model, eliminating the hefty upfront investment and ongoing management expenses associ-

ated with local computing infrastructures. Furthermore, cloud computing simplifies defining the infrastructure within the code itself, greatly enhancing the reproducibility of scientific computing results. A basic example of the benefits of having easily reproducible code is the rise of web-based interactive computing platforms such as the Jupyter Notebook project [62] or Google's Colaboratory [63] platforms. When executing the interactive code, in these platforms, we can easily choose to scale the execution environment, using computing resources that greatly exceed what we have in our personal computers. This capability is a key advantage, enabling more complex and resource-intensive operations beyond our local hardware's limitations. A cloud-native implementation of a computational experiment can also scale from a local development environment to a wide range of execution options via cloud computing services like Amazon's AWS or Microsoft Azure. Cloud computing has evolved from hosting monolithic applications on virtual machines executing on remote data centers to other architectures. Presently, prevailing design patterns in cloud computing adhere to the principles of microservices [64] and serverless [65]. Microservices architecture [66] emphasizes breaking down applications into smaller, loosely coupled components, fostering flexibility and scalability. On the other hand, serverless computing further refines this paradigm by abstracting away infrastructure concerns, enabling developers to focus on coding while minimizing the management of computational resources. The combined use of these design patterns enhances the efficiency and reproducibility of computational experiments in cloud-native environments.

Cloud-native applications bring new methodologies and techniques [67] to the development of enterprise applications. To better exploit the capabilities of the cloud infrastructure, applications are implemented as reactive systems [68] that are generally more scalable, flexible, and fault-tolerant; these can be implemented as a loosely coupled collection of microservices. These application components are easily implemented and deployed to cloud environments, where processing nodes can be defined and deployed using scripts, and scalable message queue services are provided to send and receive events between them. Furthermore, best practices propose the use of continuous deployment procedures using software tools for automatic software

provisioning, configuration management, and application deployment. Additionally, scalability in such systems is achieved through the automatic provisioning of additional computing nodes as demand increases or in response to node failures, ensuring consistent performance and reliability.

Microservices are typically executed within isolated runtime environments called containers [69]. These containers do more than just provide a runtime environment; they encapsulate all the necessary components required for the microservice to function autonomously. This includes software libraries, binaries, and configuration files. Essentially, containers package everything that a node needs to operate. The strength of a containerized application lies in its ease of operation on automation platforms. These platforms can take containerized microservice-based applications from a local computer to be deployed in a cloud, or an ephemeral infrastructure [70, 71].

The integration of cloud-based evolutionary algorithms into cloud computing has been a gradual evolution within the field. Two notable examples of the integration of evolutionary algorithms with cloud computing include the Offspring framework developed by Vecchiola et al. [72] and the FlexGP system by Sherry et al. [73]. The Offspring framework implements a multi-objective Evolutionary Algorithm (EA) designed to operate on Aneka Enterprise Clouds. The system is built on top of a task model with a plugin architecture, enhancing its flexibility and extensibility. On the other hand, the FlexGP system is a pioneering large-scale Genetic Programming (GP) system designed for cloud deployment. It adopts an Island model approach, specifically implemented on Amazon EC2, utilizing a socket-based client-server architecture. Furthermore, Valenzuela and García-Valdez [74] implemented a pool-based evolutionary algorithm using EC2 instances as workers, using a distributed pool for asynchronous collaboration between workers. Another feature of cloud computing is to provide infrastructure as a service (IaaS).

Several works starting to use these platforms, EvoSpace [6] used the PiCloud platform. PiCloud was a cloud for developers to run Python-based applications and tasks. It was designed to simplify the deployment and scaling of Python code in the cloud. Salza and Ferrucci took similar steps [75, 76] when they proposed an

architecture to extend the reach of evolutionary algorithms into the cloud. This pioneering work laid the foundation for subsequent developments, as demonstrated in their subsequent paper [77].

The mentioned papers mark a transition toward incorporating cloud-native elements into the domain of evolutionary algorithms. Notably, they introduced aspects like the integration of messaging queues and adopting CoreOS as an operating system specifically tailored for efficient container utilization in evolutionary algorithms. However, it's worth noting that despite the infusion of cloud-native features, the management of containers in these instances still adheres to a more traditional approach from the perspective of distributed Evolutionary Computing (EC). Specifically, a master-worker architecture is employed, with communication facilitated through RabbitMQ, a messaging queue. In this setup, replicated workers are responsible for executing tasks in parallel, aligning with the distributed nature of EC methodologies.

While the papers introduce cloud-native elements, the underlying container management strategy retains a classical distributed EC framework, emphasizing the orchestration of tasks through a master node and distributed workers communicating via a messaging queue. This hybrid approach leverages both cloud-native technologies and established EC paradigms to enhance the scalability and efficiency of evolutionary algorithms in distributed environments. Dziurzanski et al. [78] implements an island model using a container-based architecture.

Another popular approach is offered by Pool-based systems [7, 79] in which a pool of candidate solutions is shared among all computing resources. In a pool-based system, worker nodes pull population samples from the pool and run several iterations of population-based algorithms instead of just doing the evaluation of candidate solutions. These systems behave more like serverless systems [66], which are closer to the nature of cloud-native systems.

Similar to the multi-population concepts in EC, other search algorithms include the concept, currently, there are many proposals for Multi-Swarm configurations [59] for the PSO algorithms.

Chapter 3

Optimization Problem

In this chapter, we will outline the optimization problem we will use throughout the thesis. We consider the optimization a fuzzy path-tracking controller, based on rear-wheel feedback for an autonomous robot. In this work, we are interested in population-based metaheuristics, for optimizing the design of a fuzzy controller. In the next section 3.1, we present the kinematic model for the rear-wheel controller. In section 3.2, we present the design of a parametrizable fuzzy controller. This controller is designed to accept various parameters of Membership Functions (MFs) and generate a corresponding candidate controller, tailored to meet specific performance criteria. Finally in section 3.3, we formally define the optimization problem, detailing the optimization procedure, fitness function, and complexity analysis.

3.1 Kinematic Model

In this thesis, we propose a rear-wheel controller for a bicycle-type kinematic robot that adheres to the framework presented by Paden et al. [8]. As a bicycle, the robot consists of two wheels in tandem, a rear wheel, and a front wheel used for steering along an axis perpendicular to the plane of motion and the steering angle is represented by δ (see Figure 3.1). The length of the rigid link connecting both wheels is denoted by l. Because we are implementing a rear-wheel control, the nonholonomic constraints consider the rear wheel's position [80, 81]. The position of the midpoint

of the rear wheel is given by x_r, and y_r, while θ is the link's orientation angle with respect to the x-axis. The model incorporates the following differential constraints:

$$\begin{aligned} \dot{x}_r &= v_r \cos(\theta), \\ \dot{y}_r &= v_r \sin(\theta), \\ \dot{\theta} &= \tfrac{v_r}{l} \tan(\delta). \end{aligned} \tag{3.1}$$

The controller must provide a steering angle that does not go out of the steering limits of the robot, this means that $\delta \in [\delta_{min}, \delta_{max}]$. Another constraint is the minimum and maximum velocities, $v \in [v_{min}, v_{max}]$. The steering angle is related to the heading rate ω by

$$\delta = \arctan\left(\frac{l\omega}{v_r}\right), \tag{3.2}$$

and the heading dynamics are

$$\dot{\theta} = \omega, \quad \omega \in \left[\frac{v_r}{l}\tan(\delta_{min}), \frac{v_r}{l}\tan(\delta_{max})\right]. \tag{3.3}$$

.

The feedback of the controller is the rear-wheel position, as illustrated by Figure 3.1. The path, delineated in red within the figure, is a continuous function described in [82]. The nearest point on the reference path (defined in our thesis by a cubic spline) is given by

$$s(t) = \arg\min_{\gamma} \|(x_r(t), y_r(t)) - (x_{ref}(\gamma), y_{ref}(\gamma))\| \tag{3.4}$$

and the tracking error vector is

$$d(t) = (x_r(t), y_r(t)) - (x_{ref}(s(t)), y_{ref}(s(t))). \tag{3.5}$$

The heading error is based on a unit vector $\hat{t}$ (shown in green on Figure 3.1) tangent to the path at $s(t)$ given by

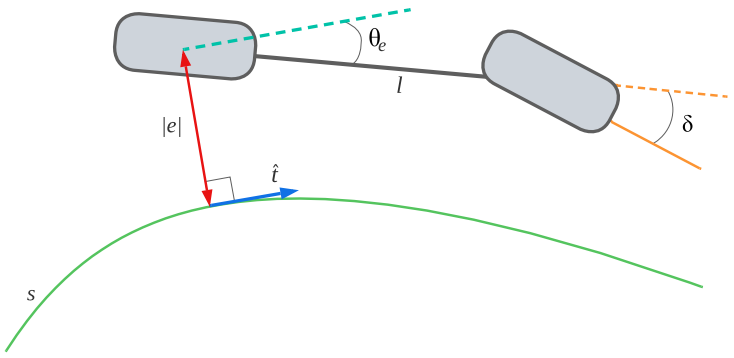


Figure 3.1: The rear-wheel feedback control involves monitoring the error variable e, depicted in red, which represents the deviation from the path measured relative to the rear wheel. When $e > 0$, the deviation is to the right of the path, whereas $e < 0$ indicates deviation to the left. Additionally, θ_e signifies the disparity between the tangent at the nearest point on the path and the current heading θ. The controller output is the angular velocity ω, subsequently utilized to determine the front wheel's steering angle δ.

$$\hat{t} = \frac{\left(\frac{\partial x_{ref}}{\partial s}\Big|_{s(t)}, \frac{\partial y_{ref}}{\partial s}\Big|_{s(t)} \right)}{\left\| \left(\frac{\partial x_{ref}(s(t))}{\partial s}, \frac{\partial y_{ref}(s(t))}{\partial s} \right) \right\|}. \tag{3.6}$$

The cross-product of the two vectors gives us the error e

$$e = d_x \hat{t}_y - d_y \hat{t}_x. \tag{3.7}$$

The heading error uses the angle θ_e between the robot's heading vector and $\hat{t}$

$$\theta_e(t) = \theta - arctan_2 \left(\frac{\partial x_{ref}(s(t))}{\partial s}, \frac{\partial y_{ref}(s(t))}{\partial s} \right). \tag{3.8}$$

3.2 Fuzzy Controller Design

We must carefully select the variables we'll be fuzzifying to develop a fuzzy controller for our problem. The first input is the error (e) representing the distance between

the rear wheel and the nearest point on the path. Additionally, we include the angle θ_e, which denotes the angular disparity between the vehicle's orientation and the path. The polarity of this angle depends on the vehicle's position relative to the path. The fuzzy controller's output, denoted as the heading rate ω, is the variable needed to calculate the steering angle δ. It is important to note that the robot maintains a constant target velocity, we will not implement a fuzzy controller for the velocity v, in this work, we use the following proportional controller:

$$a = K_p(v_{ref} - v_r). \tag{3.9}$$

3.2.1 Parametrizable Fuzzy Controller

We are not designing a static fuzzy controller. We need a design that a bioinspired metaheuristic will later optimize. In this work, we optimize only the membership functions's parameters, because this is the main strategy used in the current literature. The early results of this design, in which we optimized a FIS with three MFs for each variable was published in a book chapter [2]. Also, the current design was also published as part of a peer-reviewed journal publication [9].

The current knowledge base consists of 25 fuzzy rules that encapsulate the expertise typically employed by a driver to maintain proximity to a designated path through adjustments to the front wheel's steering. Initially, the application of these rules did not yield the expected performance. To address this, we undertook a tuning process involving multiple simulations in a trial-and-error fashion. The refined set of fuzzy rules is detailed in Table 3.1. This design significantly enhanced the controller's functionality. The structure of the rule set will not be optimized, and the parameters of the MFs are fixed in this version. We now center our attention on the design of the MFs, as these need to be parametrizable structures to facilitate optimization via metaheuristic techniques. We describe our approach to this problem and describe the main structures in detail in the following sections.

Table 3.1: Fuzzy Rules for the non-optimized controller utilizing Five MFs

R_1:	If θ_e is	`hi_neg`	and e is	`hi_neg`	then ω is	`hi_pos`
R_2:	If θ_e is	`hi_neg`	and e is	`med_neg`	then ω is	`hi_pos`
R_3:	If θ_e is	`hi_neg`	and e is	`low`	then ω is	`hi_pos`
R_4:	If θ_e is	`hi_neg`	and e is	`med_pos`	then ω is	`med_pos`
R_5:	If θ_e is	`hi_neg`	and e is	`hi_pos`	then ω is	`low`
R_6	If θ_e is	`med_neg`	and e is	`hi_neg`	then ω is	`med_pos`
R_7:	If θ_e is	`med_neg`	and e is	`med_neg`	then ω is	`med_pos`
R_8:	If θ_e is	`med_neg`	and e is	`low`	then ω is	`med_pos`
R_9:	If θ_e is	`med_neg`	and e is	`med_pos`	then ω is	`med_pos`
R_{10}:	If θ_e is	`med_neg`	and e is	`hi_pos`	then ω is	`low`
R_{11}:	If θ_e is	`low`	and e is	`hi_neg`	then ω is	`hi_pos`
R_{12}:	If θ_e is	`low`	and e is	`med_neg`	then ω is	`low`
R_{13}:	If θ_e is	`low`	and e is	`low`	then ω is	`low`
R_{14}:	If θ_e is	`low`	and e is	`med_pos`	then ω is	`low`
R_{15}:	If θ_e is	`low`	and e is	`hi_pos`	then ω is	`hi_neg`
R_{16}:	If θ_e is	`med_pos`	and e is	`hi_neg`	then ω is	`low`
R_{17}:	If θ_e is	`med_pos`	and e is	`med_neg`	then ω is	`med_neg`
R_{18}:	If θ_e is	`med_pos`	and e is	`low`	then ω is	`med_neg`
R_{19}:	If θ_e is	`med_pos`	and e is	`med_pos`	then ω is	`med_neg`
R_{20}:	If θ_e is	`med_pos`	and e is	`hi_pos`	then ω is	`med_neg`
R_{21}:	If θ_e is	`hi_pos`	and e is	`hi_neg`	then ω is	`low`
R_{22}:	If θ_e is	`hi_pos`	and e is	`med_neg`	then ω is	`med_neg`
R_{23}:	If θ_e is	`hi_pos`	and e is	`low`	then ω is	`hi_neg`
R_{24}:	If θ_e is	`hi_pos`	and e is	`med_pos`	then ω is	`hi_neg`
R_{25}:	If θ_e is	`hi_pos`	and e is	`hi_pos`	then ω is	`hi_neg`

3.2.2 Parametrizable Membership Functions

We detail in this section the methodology for optimizing the parameters of the MFs. Initially, it is essential to determine which parameters will remain constant and which will undergo optimization. By convention, we ensure that the MFs are symmetrical with respect to zero; consequently, the midpoint of the triangular MF labeled as `low` is invariably set to zero. Furthermore, the extremities of the trapezoidal MFs categorized as `high` are consistently fixed at 50 and -50 or 5 and -5, depending on the side of the error.

Throughout the work of the thesis, the number of parameters of the MFs that we optimized changed. At first, parameters associated with ω were constant to constrain the search space. In that first attempt, the parameters were only 9, because the granularity of the membership functions had only three membership functions. In the final version, the ω variable parameters were also optimized, yielding an optimization problem of 15 parameters. The final version of the MF definitions for the controller is illustrated in Table 3.2.

Table 3.2: Configuration of the fuzzy controller having 15 parameters with a granularity involving five symmetric Membership Functions (MFs).

Variable	**Linguistic Value**	**MF**	**Parameters**
θ_e	high negative	μ_{trap}	[-50, -5, -b, -b+c]
θ_e	medium negative	μ_{tria}	[-d-e, -d, -d+e]
θ_e	low	μ_{tria}	[-a, 0, a]
θ_e	medium positive	μ_{tria}	[-d-e, d, d+e]
θ_e	high positive	μ_{trap}	[b-c, b, 5, 50]
$error$	high negative	μ_{trap}	[-50, -5, -g, -g+h]
$error$	medium negative	μ_{tria}	[-i-j, -i, -i+j]
$error$	low	μ_{tria}	[-f, 0, f]
$error$	medium positive	μ_{tria}	[i-j, i, i+j]
$error$	high positive	μ_{trap}	[g-h, g, 5, 50]
ω	high negative	μ_{trap}	[-50, -5, -l, -l+m]
ω	medium negative	μ_{tria}	[-n-o, -n, -n+o]
ω	low	μ_{tria}	[-k, 0, k]
ω	medium positive	μ_{tria}	[n-o, n, n+o]
ω	high positive	μ_{trap}	[l-m, l, 5, 50]

Another critical factor in the parameter-tuning process is setting the range of permissible values for each parameter. This will prevent the optimization algorithm from proposing MFs that are not feasible and also reduce the search space.

Conventionally, when employing a population-based metaheuristic, all parameters are maintained within the same value range. During the work of this thesis, we tested several options and have publications using previous alternatives [2].

In the current work published in [9], we introduced a method to modify the tuning range of values for tuning the MFs while ensuring that the parameters consistently

fall within the $[0, 1]$ range. The core of our technique involves a normalization process that recalibrates the input variables to be within a predetermined range before they are used in the membership functions. The normalization is mathematically represented as follows:

$$\text{parameter} \leftarrow \min + (\max - \min) * \text{parameter}$$

Here, *min* and *max* denote the minimum and maximum limits of the allowable range for the parameters, respectively. This formula adjusts each parameter based on its unique allowable range, effectively acting as an "aperture factor". This ensures that while each parameter can vary within a specific range tailored to its role within the MF, all parameters are scaled to remain within $[0, 1]$ a common range used in continuous optimization. Based on our experience, we have determined specific ranges for each parameter, which are detailed in Table 3.3.

Table 3.3: Ranges defined for each parameter for the 5MF controller.

Parameter	**Range**	**Parameter**	**Range**	**Parameter**	**Range**
a	[0,1]	f	[0, 1]	k	[0, 1]
b	[0.5,2]	g	[0.5, 2]	l	[0.5, 2]
c	[0,2]	h	[0, 2]	m	[0, 2]
d	[0.5,1.5]	i	[0.5,1.5]	n	[0.5,1.5]
e	[0,1]	j	[0, 1]	o	[0, 1]

3.3 Optimization Problem Formulation

We now have a parametrizable fuzzy controller, and can formally define the optimization problem. The optimization challenge involves identifying the optimal parameter vector, denoted as x^{mf}, which defines the membership functions in such a way as to minimize the tracking error. The optimization problem can be formally expressed as follows:

$$\arg\min_{x^{mf}}\{FC_{error}\} \tag{3.10}$$

in our work, we proposed using a fitness function FC_{error} that was obtained by running three simulations and averaging the RMSE:

$$FC_{error} = \frac{1}{3}\sum_{k=1}^{3} rmse(FC(x^{mf}), s_k, t_{max}) \tag{3.11}$$

k represents the number of predefined paths, each denoted by s_k. Each simulation involves executing a control problem as detailed in Section 3.1, wherein control operations are managed by the candidate fuzzy controller $FC(x^{mf})$, with MFs specified by x^{mf}. The variable t_{max} represents the total number of cycles completed in a single simulation. As previously mentioned, the primary objective of the controller is to minimize the tracking error. To quantify this, the Root Mean Square Error (RMSE) can be calculated using the tracking error vector, which is defined in Equation 3.12:

$$rmse = \sqrt{\frac{\sum_{t=1}^{t_{max}}(x_r(t), y_r(t)) - (x_{ref}(s_k(t)), y_{ref}(s_k(t)))^2}{t_{max}}} \tag{3.12}$$

The parameter vector x^{mf} specifies the settings for the membership functions as outlined in Table 3.2, and for the knowledge base detailed in Table 3.1:

$$x^{mf} = \{x_1^a, x_2^b, x_3^c, x_4^d, x_5^e, x_6^f, x_7^g, x_8^h, x_9^i, x_{10}^j\} \tag{3.13}$$

The range of all parameters is between 1 and 0:

$$0 \leq x_i \leq 1 \tag{3.14}$$

3.3.1 Metaheuristic Optimization Procedure

The procedure outlined in this section is designed to serve as a general framework for bio-inspired, population-based metaheuristics. This strategy is applicable across various algorithms, including GA, PSO, GWO, and other similar methodologies.

In population-based metaheuristics, it is essential to assess the fitness of each candidate solution within the population. This evaluation process is depicted in Figure 3.2, where each candidate solution is represented by a parameter vector x^{mf}. To evaluate each solution, an instance of the fuzzy controller is generated. This controller is then integrated into the simulation environment, where it operates in conjunction with predefined paths that the mobile robot is set to follow. The effectiveness of each candidate solution is quantified by the RMSE of the accumulated positional errors e observed throughout the simulations. The average RMSE (see Equation 3.12) serves as the fitness measure for each candidate solution.

3.3.2 Fitness Function

In our early research for this thesis, we utilized Genetic Algorithm (GA) specifically to evolve controllers on a singular path and publish these preliminary results(see [2]), in that work, we observed that using only a single track as the fitness function could potentially result in over-training, a phenomenon akin to that in supervised learning algorithms. Specifically, the controller optimized by the GA tended to be overly specialized to the path used during its development. To mitigate this, we expanded our evaluation methodology to include three distinct paths, calculating fitness based on the average RMSE across three simulations. We conducted experiments using both single and triple-path setups. Each path was defined using cubic splines based on a list of coordinates, a method commonly employed in the field [83].

The paths are delineated in Figure 3.3. The initial path, labeled "m", was derived from the library of [84] and features a challenging initial narrow left curve yet is differentiable throughout. The other paths, designated "A" and "S", incorporate smoother transitions with extended straight segments and varying curvatures. All paths utilized seven anchor points for the spline construction.

3.3.3 Complexity of the Optimization Procedure

The computational complexity of the optimization procedure depends mainly on the complexity of the fitness function. Estimating the cost of each simulation step

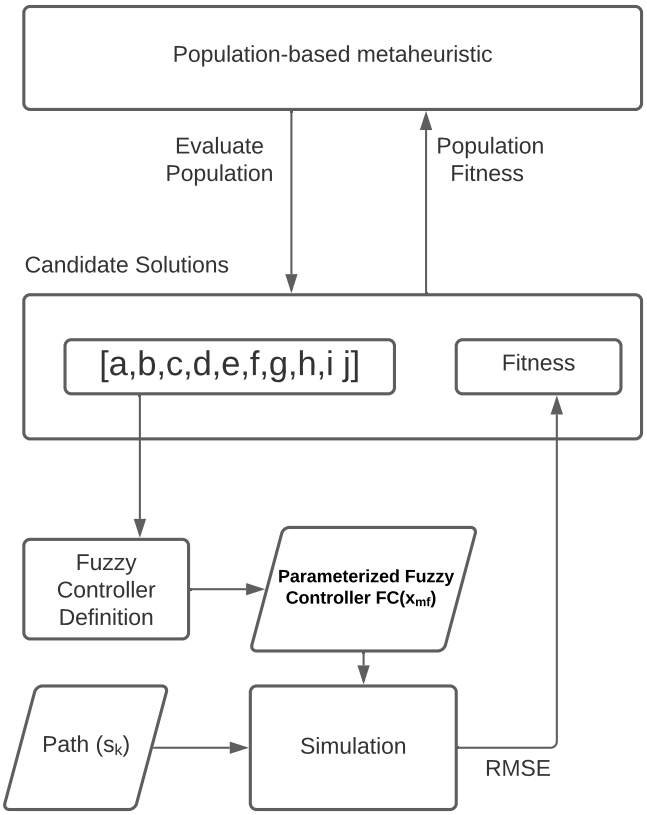
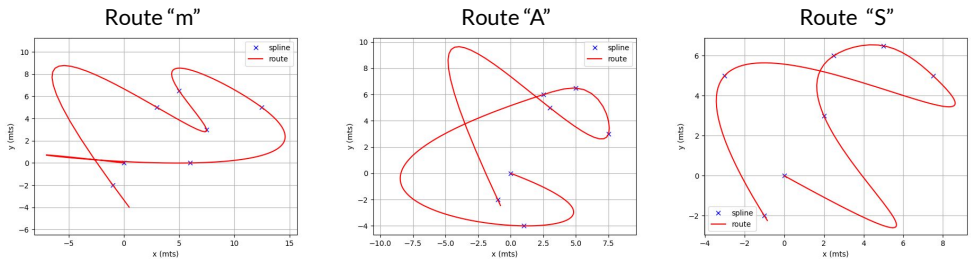


Figure 3.2: To assess the fitness of each candidate solution, a controller is configured using the specified parameters and subsequently evaluated through one or more simulations. The average RMSE of the tracking performance determines the fitness of each candidate solution. This evaluation process is systematically repeated for every member of the population.



ax = [0, 6, 12, 5, 7.5, 3, -1]
ay = [0, 0, 5, 6.5, 3, 5, -2]

ax = [0, 1, 2.5, 5, 7.5, 3, -1]
ay = [0, -4, 6, 6.5, 3, 5, -2]

ax = [0, 2, 2.5, 5, 7.5, -3, -1]
ay = [0, 3, 6, 6.5, 5, 5, -2]

Figure 3.3: Tracks Utilized for Fitness Evaluation: Defined by Cubic Splines, with Parameters Detailed Below Each Illustration.

is challenging due to its dependency on an ordinary differential equation solver (ODEInt), specifically a multi-step solver (lsode), where the number of iterations varies with the initial conditions. Furthermore, the calculation of the nearest point to the path, as defined in Equation 3.4, involves a local optimization to find γ that determines the nearest $(x_{\text{ref}}(\gamma), y_{\text{ref}}(\gamma))$. Despite these complexities, the function evaluation can generally be assumed to exhibit domain-dependent, high-order complexity, minimally $\mathcal{O}(n^3)$.

In terms of population-based algorithms, particularly those explored in this chapter, the typical process involves an initial random population initialization, which has a complexity of $\mathcal{O}(n)$, with n representing the number of candidate solutions. Subsequent to evaluation, the solution update step usually incurs a complexity of $\mathcal{O}(m \times n) + \mathcal{O}(m \times n \times l)$, where m is the number of iterations, and l is the number of parameters in the fitness function. Additional steps may include the retention of only those solutions that achieve a fitness improvement, adding an extra $\mathcal{O}(n)$ to the cost, and some algorithms sort the population by fitness after each iteration,

which introduces a complexity of $\mathcal{O}(n \log n)$.

This section outlines the optimization problem addressed in this thesis. The problem is characterized by a substantial search space, necessitating significant computational resources for experimental execution. In the subsequent chapter, we will introduce a distributed optimization method designed to enhance the efficiency of the optimization process by reducing execution times.

Chapter 4

Proposed Architecture

In this chapter, we detail the distribution model that forms the cornerstone of this thesis. Initially, we discuss the principal elements of an event-driven architecture, subsequently linking these components to the multi-population-based algorithm employed in our study. Additionally, we will explain how these abstract components correlate with specific design decisions made during our implementation.

4.1 Populations and Workers

In this model, rather than using a singular population, we propose a set P consisting of n distinct populations. Each of these populations contains fewer candidate solutions compared to those found in traditional sequential evolutionary algorithms, which is why they are sometimes termed subpopulations. Each population, denoted as $population_i$, is characterized by a tuple (X_j, P, S_j): X_j representing the current state j of the multiset X comprising n candidate solutions x_n. This framework focuses primarily on the states of populations before and after several evolutionary iterations via a metaheuristic algorithm, disregarding intermediate state changes and retaining only the optimal individual from each iteration for statistical purposes, stored within an ordered set S_j.

For example, consider a Genetic Algorithm (GA): it begins with a population X_j and after the j^{th} execution of the algorithm, transitions to state X_{j+1}. The

parameter set P includes the specific metaheuristic m applied to X along with the necessary parameters MP for its operation. Other configuration parameters involve the population size n, and the number of iterations (generations) it that are executed as $m(X_j, MP, it)$.

Each population undergoes several executions throughout the algorithm's life-cycle, constrained by the parameter nc (number of cycles). In the last step of the cycle, a combination process occurs where populations interchange candidate solutions, enhancing diversity (described further in Section 4.3). This cyclical process allows for repeated metaheuristic executions on populations after mixing.

This event-driven model effectively separates population management from the execution of metaheuristics by worker processes. Here, a population object acts as a static data structure encapsulating (X_j, P, S_j). The execution cycle's dynamics are illustrated in Figure 4.1, where a group of workers (W) systematically processes a population $population_{i,j}$ through a metaheuristic, subsequently integrating the evolved population $population_{i,j+1}$ into the mixing stage to perpetuate the cycle.

Workers operate either as standalone processes or threads, retrieving population data from a queue, executing the designated metaheuristic, and forwarding the updated population to the mixing process.

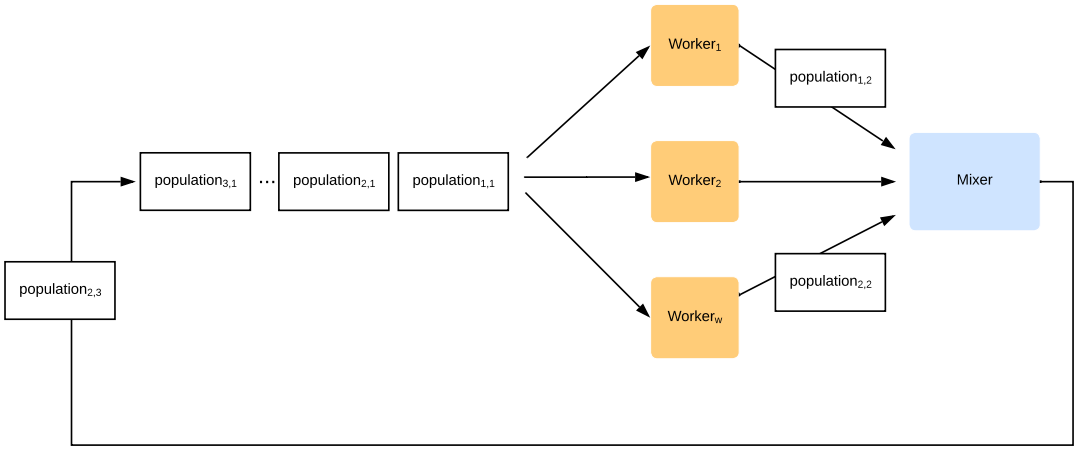


Figure 4.1: The proposed multi-population cycle.

In our implementation, populations are encoded as JSON documents, a format chosen for its interoperability across different software components. This format's widespread support allows most programming languages to easily parse these documents into native data structures. Furthermore, the workers responsible for executing the algorithms are implemented as Docker containers. These containers encapsulate the necessary Python code and libraries, enabling them to process and exchange populations represented as JSON documents.

Until now, the terms *receive* and *send* have been used generically to describe the transaction of data among components without delving into the specific mechanics of these interactions. The forthcoming section will elaborate on the communication strategy employed to facilitate these operations.

4.2 Message queues

Message queues facilitate an asynchronous communication pattern among processes or threads, a setup where the sender and receiver do not need to access the queue at the same time. This asynchronous nature allows for scalability in the number of participants—either senders or receivers—without necessitating modifications to the existing configuration. In many cloud-based systems, message queuing services are indispensable due to their ability to scale inter-process communication effectively.

In this study, we implement message queues to enable scalable communication between the architectural components of our system. Specifically, we utilize two distinct queues:

- `input-queue`.

  As the initial step of the algorithm, new populations are dispatched to this queue. After mixing, combined populations are also sent here to continue the multi-population cycle. Workers continuously pull from this queue to receive populations that need processing.

- `output-queue`.

Workers submit the evolved populations to this queue. Simultaneously, the mixing process extracts populations from this queue to combine two or more populations, and continue the evolution cycle within the system.

Figure 4.2 illustrates the operational cycle of a Worker within the system. Each worker operates in an infinite loop, initially in a blocking operation, until able to pull a message from the **input-queue**. Upon retrieving a population with valid data, the designated metaheuristic algorithm is executed starting from the current state of that population. Following the completion of this algorithm, the processed population is then pushed to the **output-queue**. The worker subsequently resumes the cycle by pulling the next population from the **input-queue**.

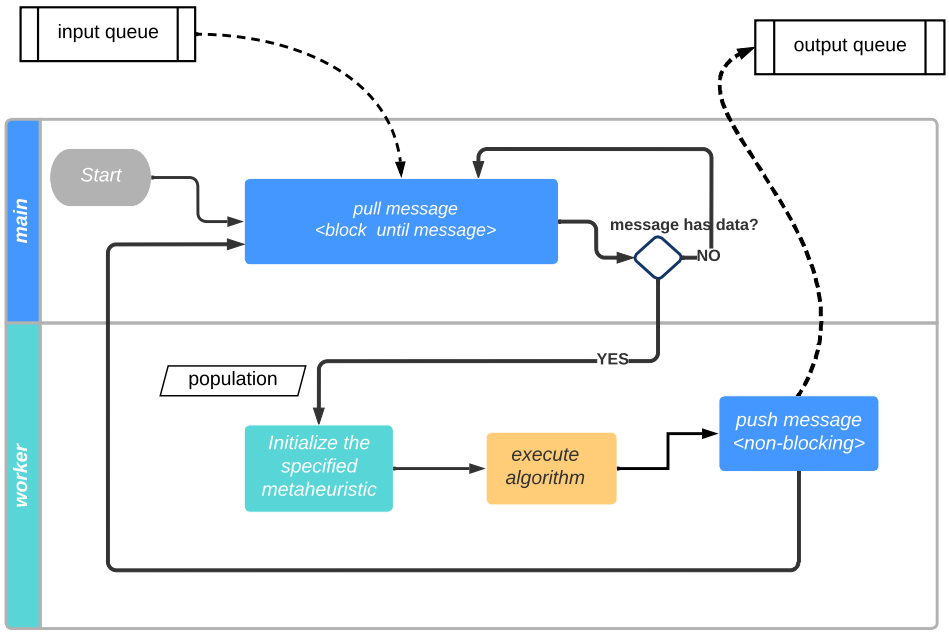


Figure 4.2: The diagram shows the worker's infinite loop operation and communication through the input and output queues.

4.3 Mixer

The aforementioned components form the basis of a scalable method for executing isolated algorithms. However, to evolve into an effective multi-population model, a critical element must be incorporated: the recombination or migration between populations. This is essential to prevent premature convergence, a common issue in smaller populations that tend to stabilize more quickly than larger ones [85].

To address this, we introduce a buffer-based **Mixer** component designed for the recombination of populations. This mixer employs a priority queue where elements are prioritized based on their fitness, and only the top k individuals are retained. The mixing algorithm initiates only when the buffer contains at least k individuals, ensuring a rich diversity for effective recombination.

Before delving into the specifics of the recombination process, it is necessary to outline the communication topology that underpins how populations interact. Populations are temporarily stored in a message queue that defines the sequence in which components access the data. This setup naturally forms a ring topology, where each population is linked to two others—one preceding and one succeeding it in the queue. The temporal sequence within the queue means that populations arriving later are recombined later, potentially delaying the dissemination of superior solutions throughout the network.

This characteristic could be advantageous, as it might enhance genetic diversity across the model, albeit possibly at the cost of slower convergence to optimal solutions. After conducting preliminary experiments, we adopted an elitist strategy, which yielded more promising results. In this optimized approach, a buffer of size k retains the top k individuals from all received populations. Subsequently, any population delivered to the mixer undergoes a replacement of its k least fit solutions with the top k from the buffer(shown Figure 4.3).

The mixing process is additionally responsible for the following tasks:

1. **Setup** This is a one-time running task consisting of reading the algorithm's configuration, creating the initial population structures, and pushing the messages (populations) to the *input-queue*.

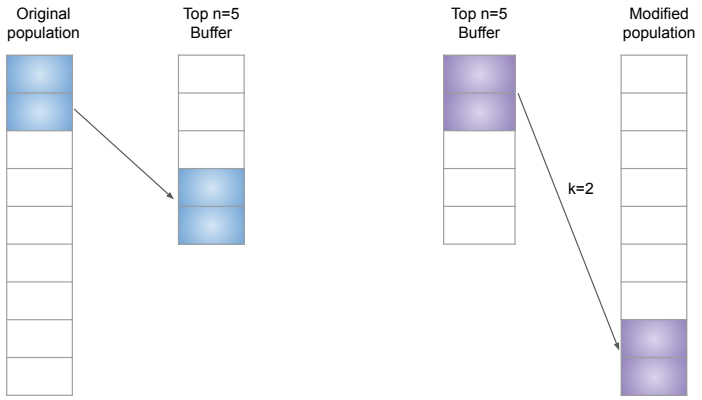


Figure 4.3: "K-Best" Strategy.

2. **Finishing the algorithm** The mixer also keeps track of the number of populations that have been pulled from the *output-queue*. It stops the algorithm if one of these two conditions is true: 1) the number of cycles has been reached, or 2) the error of the best fuzzy controller found so far reaches an appropriate average RMSE.

Figure 4.4 shows each process in their designated swim lanes, clarifying the responsibilities for implementing components of the algorithm as discussed in previous sections. Actions are sequentially numbered in yellow squares to facilitate a comprehensive understanding of the algorithm's flow. The mixing process initiates the cycle by loading a configuration file that specifies the number of populations, chosen metaheuristics, and their parameters (1). It then generates and dispatches population messages to the *input-queue* (2). Worker Containers (3) retrieve these messages as outlined in Section 4.1, execute the respective metaheuristic, and transfer the updated population states to the *output-queue* (4). Subsequent to this, the mixing process extracts populations from the *output-queue* as per the methodology

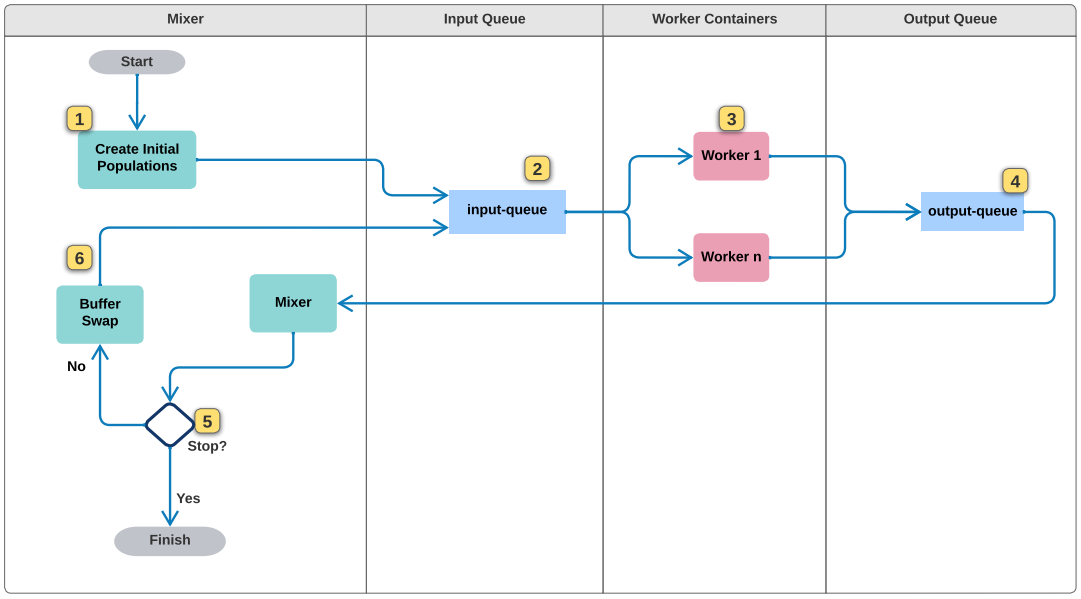


Figure 4.4: The architecture diagram illustrates each process within its respective swimlane, detailing the message-data flow, the roles of various message queues, and the responsibilities of each component involved. Sequential actions are denoted by numbers enclosed in yellow boxes, and further explanations of each step are integrated into the main text of this document.

described in Section 4.3. This process either terminates upon satisfying predefined conditions (5) or continues to exchange the top k individuals in the buffer with the bottom k individuals of the current population (6). The updated population is then reintroduced into the *input-queue* to sustain the operational loop.

In the existing setup, each worker is equipped with the Python libraries essential for operating the components previously detailed. This modular, component-based architecture affords the flexibility needed to address various fuzzy control problems.

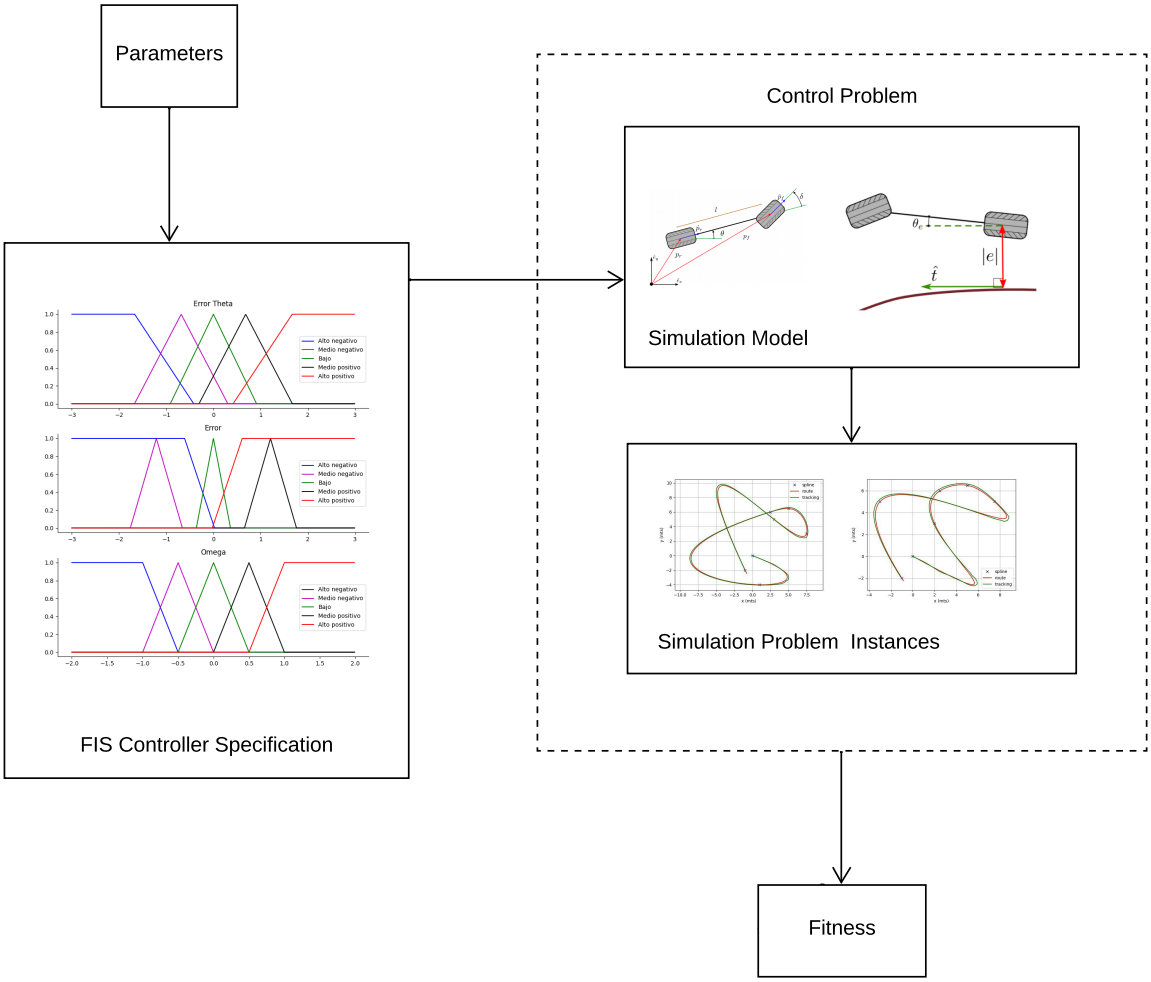


Figure 4.5: The provided diagram details the data flow between high-level components essential for optimizing the parameters of a fuzzy controller. The process begins with an input comprising a set of parameters that represent a candidate solution, and culminates in the output, which evaluates the fitness or quality of his solution. Central to this system is the FIS Controller Creation component, which creates a controller instance from the initial parameters. This instance is then rigorously tested within a defined Control Problem environment.Each Control Problem is supported by a Simulation Model that includes both the kinematic model and the requisite simulation execution code. The functionality of the controller is assessed across multiple problem instances. For instance, a path-tracking controller might be evaluated on various path configurations. During these tests, the robot's error in following each path is recorded, and the controller's overall performance is quantified by averaging these error values.

Chapter 5

Multi-population Algorithm Design for Fuzzy Control

This chapter delineates the experiments and results obtained from various configurations and enhancements to the algorithms proposed throughout this thesis. The findings, which have been documented in several publications, these results are presented here in chronological order for coherence and clarity.

Initially, we provide a detailed account of the general configuration parameters utilized across the experiments. These parameters remained largely unchanged throughout the study, thereby ensuring consistency in the experimental conditions.

5.1 General Experimental Setup

All the experiments optimize the controller described in Section 3.2.1 the main change is the number of parameters used. As previously stated, the fitness function calculates the RMSE of the simulation to assess performance. To establish the average RMSE, the same tracks are used for all experiments, these are detailed in Section 3.3.2.

To optimize the use of computational resources, simulations were selectively terminated early if the robot significantly deviated from the path or failed to approach the path's final point. In such instances, to maintain the objective of minimizing

error, a punitive fitness score was assigned: 5000 for substantial deviation and 2000 for not finishing near the endpoint.

As a benchmark for our analysis, we will compare our results against the control law described in [8]:

$$\omega = \frac{v_r \mathcal{K}(s)\cos(\theta_e)}{1-\mathcal{K}(s)e} - (k_\theta |v_r|)\theta_e - \left(k_e v_r \frac{\sin(\theta_e)}{\theta_e}\right) e, \tag{5.1}$$

the parameters of the simulations we run throughout the thesis are summarized in Table 5.1.

Table 5.1: Simulation and controller parameters.

Parameter	**Value**
Steering angle limits	$\lvert\delta\rvert \leq \frac{\pi}{4}$
Initial position and heading	$x_r(0), y_r(0), \theta(0) = (0,0,0)$
Rigid link length	$l = 2.5$
Maximum time of a simulation in seconds	50
Control law parameters	$k_e = 0.3, k_\theta = 1.0$
Target velocity	$v_r = \frac{10}{3}$
PID Velocity Controller parameters	$K_p = 1, v(0) = 0, a(0) = 0$

5.2 Sequential Algorithms

To start the work, we first established a baseline by comparing the optimization capabilities of a selection of bioinspired metaheuristics. We published these results in [9]. These algorithms ran using a single population, optimized ten parameters for the MFs, leaving the output the heading rate ω fixed. The fixed parameters used for ω are summarized in Table 5.2.

Table 5.2: Fixed parameters for the heading rate ω.

Variable	**Linguistic Value**	**MF**	**Parameters**
ω	high negative	μ_{trap}	$[-50, -5, -1, -0.5]$
ω	medium negative	μ_{tria}	$[-1, -0.5, 0]$
ω	low	μ_{tria}	$[-0.5, 0, 0.5]$
ω	medium positive	μ_{tria}	$[0, 0.5, 1]$
ω	high positive	μ_{trap}	$[0.5, 1, 5, 50]$

We compared the following algorithms:

1. Genetic Algorithm (GA) [42]
2. Particle Swarm Optimization (PSO) [32]
3. Aquila Optimization (AO) [38]
4. Grey Wolf Optimizer (GWO) [37]
5. Arithmetic Optimization Algorithm (AOA) [40]
6. Harmony Search (HS) [39]

All algorithms were standardized to a population size of 50, with the number of iterations fixed at 20. Based on these settings, the total number of function evaluations calculated for each algorithm is 1000. The specific parameters used for the comparative algorithms are detailed in Table 5.3. These parameters resulted from experimental runs. For the GA, Gaussian mutation was used because we are doing a continuous optimization. We selected a one-point crossover because this method tends to be less disruptive than multi-point crossover and gives a good balance between exploration and exploitation. This is also the reason behind the mutation probability of 0.3 and a value of 3 for tournament selection.

In PSO, r_1 and r_2 are drawn from a uniform distribution in the range [0, 1]. This is a typical solution to maintaining the stochastic and adaptive nature of the algorithm. Empirical studies and benchmark tests often use $C_1 = C_2 = 2$ to achieve

a balance between exploration and exploitation. These values are a standard setup because it has consistently performed well across a wide range of problems. This setting provides a good starting point for tuning PSO in specific applications and is often used in comparative studies to ensure consistency and fairness. By limiting the maximum speed of particles to [-0.25,0.25] the algorithm ensures that particles move incrementally within the search space. This controlled approach helps in finely exploring the neighborhood of promising regions without skipping over them.

Table 5.3: Configuration parameters for the algorithms used in the comparison.

Algorithm	**Parameter**	**Value**
GA	Selection	Tournament Selection (k=3)
	Mutation	Gaussian ($\mu = 0.0$ and $\sigma = 0.2$)
	Mutation probability	0.3
	Crossover	One point crossover (probability = 0.7)
PSO	Topology	Fully connected
	Speed limit	Min=-0.25,Max=0.25
	Cognitive and Social constants	$C_1 = 2, C_2 = 2$
	r_1 and r_2	uniformly random in [0,1]
AO	α	0.1
	δ	0.1
GWO	Convergence parameter (a)	Linearly decreased from 2 to 0
AOA	α	5
	μ	0.5
HS	HMConsidering Rate (HMCR)	0.95
	Pitch Adjusting Rate	0.05

The experimental setup was conducted on a Desktop PC equipped with an AMD Ryzen 9 3900x 12-core processor, capable of handling 24 threads, and 48 GB of RAM, running Ubuntu Linux 21.04. The experiments were implemented using Python 3.7.5. The source code and data utilized in these experiments are publicly available in the following GitHub repository: `https://github.com/mariosky/fuzzy-control`.

For the statistical analysis, the performance of the algorithms was compared based on the mean, median, and standard deviation of the Root Mean Square Error (RMSE) from 30 separate executions of each algorithm. Additionally, to further assess performance disparities between algorithms, a Wilcoxon rank-sum test was conducted.

5.2.1 Results

The descriptive statistics for the results are documented in Table 5.4. Given the non-deterministic nature of the optimization processes employed, some outliers were observed at both extremes of performance. Notably, the Genetic Algorithm (GA) exhibited outliers with an RMSE of 0.18205, while the Archimedes Optimization Algorithm (AOA) had outliers at an RMSE of 0.7978. Among the algorithms tested, the Particle Swarm Optimization (PSO) algorithm achieved the lowest average RMSE (0.00546) and recorded the best individual controller performance (RMSE = 0.00158). The AOA algorithm demonstrated a median RMSE of 0.00523.

Table 5.4: Descriptive statistics (n=30) results for sequential algorithms.

Algorithm	**Average RMSE**	**Standard Deviation**	**Median**	**Min**	**Max**
GA	0.01564	0.03163	0.00918	0.00574	0.18205
PSO	**0.00546**	0.00202	0.00536	**0.00158**	0.00102
AO	0.00695	0.00210	0.00696	0.00355	0.01407
GWO	0.00617	**0.00170**	0.00643	0.00315	0.00849
AOA	0.03413	0.14403	**0.00523**	0.00198	**0.79780**
HS	0.00774	0.00273	0.00740	0.00320	0.01454

Figure 5.1 illustrates that the Genetic Algorithm (GA) displays the poorest median performance and overall results among the algorithms evaluated. In contrast, the other algorithms demonstrated comparably competitive outcomes.

Table 5.5 presents the results of the Wilcoxon rank-sum test, applying an alternative hypothesis H_1 that assesses whether the algorithm in each row has a statistically lower median than that of the algorithm in each column. The significance level is

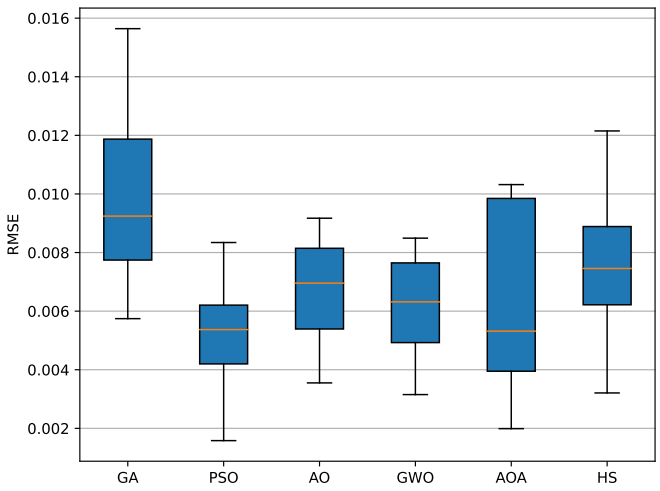


Figure 5.1: Box plot showing the results of 30 experiments, with outliers filtered in order to show the plot at a larger scale.

set at $\alpha = 0.05$, with results yielding a p-value below this threshold highlighted in bold.

Among the six algorithms tested, the PSO algorithm exhibited superior performance, outperforming the GA, AO, GWO, and HS algorithms. Notably, the GA algorithm was outperformed by all other algorithms. The AOA maintained a robust performance, as any other algorithm did not statistically outperform it. Meanwhile, the GWO algorithm achieved more victories over the HS algorithm than any other, positioning it as the second most successful in terms of the number of wins.

Overall, the majority of the algorithms (PSO, AO, AOA, and GWO) achieved commendable results.

To facilitate a qualitative comparison, we selected two of the best controllers from the 30 experimental runs: one from the most effective algorithm (PSO) and another from the least effective (GA). These controllers were then benchmarked against the baseline established by the control law detailed in Equation 5.1, with the comparative results presented in Table 5.6.

Table 5.5: Wilcoxon rank-sum test between algorithms, showing p-values for H_1 : $A < B$

	GA	PSO	AO	GWO	AOA	HS
GA		1.00E+00	1.00E+00	1.00E+00	9.99E-01	9.99E-01
PSO	**6.44E-09**		**2.23E-03**	**3.80E-02**	1.73E-01	**9.54E-05**
AO	**1.20E-05**	9.98E-01		9.22E-01	7.76E-01	1.33E-01
GWO	**1.19E-07**	9.63E-01	7.96E-02		4.80E-01	**1.31E-02**
AOA	**1.02E-03**	8.31E-01	2.28E-01	5.25E-01		1.10E-01
HS	**1.24E-03**	1.00E+00	8.70E-01	9.87E-01	8.92E-01	

Table 5.6: RMSE of the best controllers in all three paths.

Path	**Control law**	**PSO**	**GA**
Path m	2.003	**0.00217**	0.00396
Path A	0.014	**0.00168**	0.00575
Path S	0.521	**0.00195**	0.00845

5.2.2 Discussion

The analysis reveals that even the fuzzy controller optimized using the GA algorithm managed to outperform the baseline control law on two distinct paths. Moreover, the results obtained with the PSO algorithm were exceptionally competitive when compared to the baseline. Furthermore, we have shown that by adding additional simulations to establish the quality of candidate solutions, during the evolution, the generated controllers have better performance than those evolved with just one simulation.

5.3 Multi-Population Metaheuristics

In this study, we have opted to utilize both the Genetic Algorithm (GA) and the Particle Swarm Optimization (PSO) algorithms for the multi-population-based optimization framework. This decision was influenced by the outcomes of our previous experimental analyses and the distinct methodologies employed by each algorithm. GA is typically recognized as a quintessential evolutionary algorithm, noted for its efficacy in combinatorial and discrete optimization tasks. Despite yielding the least favorable results among the metaheuristics we previously evaluated, its inclusion here aims to explore whether employing a multi-population approach could potentially enhance its performance.

Conversely, the PSO algorithm demonstrated the most impressive results in our prior experiments, which was anticipated given its suitability for continuous optimization problems like the one at hand. Moreover, historical data from continuous optimization benchmarks suggest that a hybrid approach combining these two metaheuristics tends to surpass the performance of each algorithm when implemented independently. The forthcoming experiments are designed to verify whether this synergistic effect can be replicated in our current context.

In population-based metaheuristics, a critical initial task for designers is to set the algorithm's parameters. As previously discussed, bio-inspired metaheuristics involve parameters that influence the balance between exploring and exploiting the search space [86]. Over-exploitation can lead to premature convergence, while excessive exploration may result in a quasi-random search, frequently shifting to new areas. Achieving an optimal balance is crucial for avoiding local minima and effectively searching promising areas.

In scenarios utilizing multiple populations, adjusting parameters across the populations to encourage exploration or exploitation is feasible, thereby maintaining a balanced approach. One method to manage this balance is the heterogeneous strategy proposed by Gong et al. [87], which involves random parameterization for each population. Despite its simplicity, this approach has shown promise in other multi-population algorithms.

Alternatively, a homogeneous strategy might be employed, where a single, well-

adjusted parameter configuration is applied uniformly across all populations. This thesis compares the outcomes of these heterogeneous and homogeneous strategies in multi-population metaheuristic algorithms.

The parameters used for the sequential implementations of the Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) are the same as in the previous experiment (detailed in Table 5.3). For a fair comparison with their multi-population counterparts, it is important to maintain an equivalent number of function evaluations across all setups, ensuring that each performs a comparable amount of processing. In the sequential models, the total number of function evaluations, calculated as the product of population size and number of iterations, is set at 1000.

The parameters for the distributed, multi-population versions of the GA and PSO are documented in Table 5.7.

Table 5.7: Summary of the parameters utilized for the multi-population versions of the Genetic Algorithm (GA) and Particle Swarm Optimization (PSO). The parameters are divided into three main sections: Homogeneous Parametrization, Heterogeneous Parametrization, and General Multi-Population Parameters. The latter includes those parameters that are the same across algorithms.

Homogeneous Parametrization		
Algorithm	**Parameter**	**Value**
GA	Selection	Tournament Selection (k =3)
	Mutation	Gaussian ($\mu = 0.0$ and $\sigma = 0.2$)
	Mutation probability	0.3
	Crossover	One point (probability = 0.7)
PSO	Topology	Fully connected
	Speed limit	Min= -0.25, Max= 0.25
	Cognitive and Social	$C_1 = 2, C_2 = 2$
Heterogeneous Parametrization		
Algorithm	**Parameter**	**Value or Random Range**
GA	Selection	Tournament Selection (k =3)
	Mutation	Gaussian ($\mu = 0.0$ and $\sigma = 0.2$)
	Mutation probability	[0.1, 0.5]
	Crossover	One point (probability = [0.3, 0.9])
PSO	Topology	Fully connected
	Speed limit	Min=$[-0.20, -0.30]$, Max=[0.20, 0.30]
	Cognitive and Social	$C_1 = [1.0, 2.0], C_2 = [1.0, 2.0]$
General Multi-Population Parameters		
	Population Size	9
	Number of Populations	7
	Iterations per pull	4
	Cycles	4
	Number of Function Evaluations	1008

We replicated the parameter set from the sequential models across all islands for the homogeneous parametrization, as these parameters previously yielded favorable results.

In the heterogeneous case, random adjustments were made to those parameters that influence the balance between exploration and exploitation: the GA's mutation probability varies within $[0.1, 0.5]$, and its crossover probability within $[0.3, 0.9]$. Similarly, for the PSO, the velocity limits are set between $[-0.30, -0.20]$ for the minimum and $[0.20, 0.30]$ for the maximum, with both attraction coefficients C_1 and C_2 ranging from $[1.0, 2.0]$.

In the distributed configurations, the global population is segmented into seven isolated populations, each having a size of nine. This segmentation has a combined population size of 63 candidate solutions. Using smaller populations reduces the computational burden. Each island's population is subjected to four generations (iterations) per evolutionary cycle. With the entire distributed algorithm completing four cycles in total. As a result, the cumulative number of function evaluations marginally surpasses that of the sequential models, reaching a total of 1008. This figure closely aligns with the computational demands of the single-population setups.

In our comparative analysis, we also evaluated a hybrid configuration comprising four populations employing the Particle Swarm Optimization (PSO) algorithm and three populations implementing the Genetic Algorithm (GA). Both algorithms in this combined version adhered to the same parameter settings as previously described. This mixed approach was designed to investigate whether integrating different search strategies could yield better results when compared to the single-algorithm configurations. These combinations were tested under both homogeneous and heterogeneous parametrization strategies.

To compare the distributed versions of these algorithms, we used the optimization problem detailed in Chapter 3

The experiments were conducted using a workstation equipped with an AMD Ryzen 9 3900x 12-core CPU and 48 GB of RAM, in Ubuntu 21.04 and CPython 3.7.5. We utilized containers orchestrated by a *docker-compose* script for the distributed tests. The *docker-compose* service adds a Redis container for managing in-memory queues and defines a *worker* container that scales to seven instances.

All containers were terminated after each trial to ensure experimental integrity, guaranteeing a fresh start for subsequent tests. The mixing process, in contrast, was

executed directly on the host.

The performance of the algorithms was evaluated based on the mean, median, and standard deviation of the Root Mean Square Error (RMSE) from 30 separate executions of each configuration. Additionally, we recorded the execution time for each algorithm run.

Speedup is an essential performance metric in this study, as our primary objective is to enhance program execution efficiency. Our definition of speedup is based on the framework provided by Touati et al. [88]:

Define $\mathcal{C}$ as the baseline and $\mathcal{C}'$ as the alternative. Let X represent the random variable for the execution time of $\mathcal{C}$, with $\mathcal{X} = x_1, \ldots, x_n$ constituting a sample of n execution times. Similarly, the alternative $\mathcal{C}'$ is executed m times, with its execution times represented by the random variable Y and the sample set $\mathcal{Y} = y_1, \ldots, y_m$.

Given these data sets, we calculate the observed speedup in mean execution times using the formula:

$$\frac{\overline{X}}{\overline{Y}} = \frac{\sum_{i=1}^{n} x_i}{\sum_{j=1}^{m} y_j} \times \frac{m}{n}. \tag{5.2}$$

The implications and detailed discussion of these results are presented in the following section.

5.3.1 Results

In this section, we detail the experimental outcomes from comparing the previously described sequential and distributed implementations of the optimization algorithm.

Initially, we present the RMSE performance of the optimized controllers. Subsequently, our focus shifts to analyzing the completion time of these experiments, providing insights into the efficiency gains potentially offered by the different implementations.

RMSE

Table 5.8 displays the RMSE results for various configurations tested. The first two columns contain data from the sequential algorithms, as reported in Section 5.2.1.

Among these, the Particle Swarm Optimization (PSO) algorithm exhibited the most favorable outcomes, achieving a median RMSE of **0.00536160**. Notably, the distributed heterogeneous PSO-GA version was the second most effective, with a median RMSE of <u>0.00610783</u>, closely followed by the multi-algorithm PSO version at 0.00628949.

A statistical Z-test was conducted to compare the distributed and sequential versions of the same algorithms. The results did not provide sufficient evidence to reject the null hypothesis $H_0 : \mu_{seq} \leq \mu_{dist}$ at the $\alpha = 0.05$ significance level. These findings suggest that the performance of distributed and sequential versions is comparable. Furthermore, the random parametrization employed in the multi-population version appears to offer slight advantages over the homogeneous parametrization, eliminating the necessity to experimentally optimize the parameter values.

Table 5.8: Outcomes from 30 executions of each algorithm, with controller performance quantified by the RMSE of the most effective solution identified in each run. The results for the sequential algorithms are shown in the first two columns, followed by the outcomes for the distributed versions. Within the table, results for the homogeneous configurations are displayed on the left, while those for the heterogeneous configurations are on the right. The best-performing result is highlighted in bold, and the second-best result is underlined.

	Sequential		Homogeneous Parameters (Distributed)			Heterogeneous Parameters (Distributed)		
	GA	**PSO**	**GA**	**PSO**	**PSO-GA**	**GA**	**PSO**	**PSO-GA**
Average	0.01564	**0.005464**	0.01091	0.00645	0.00656	0.01029	0.00632	0.00634
Std. Dev.	0.03163	0.00202	0.0060	0.00148	0.00185	0.00332	0.00165	0.00135
Median	0.00918	**0.00536**	0.00955	0.00643	0.00625	0.01021	0.00628	<u>0.00610</u>
Min	0.00574	**0.00158**	0.00384	0.00360	0.00336	0.00399	0.00310	0.00388
Max	0.18205	0.01026	0.03455	0.01002	0.0116	0.01584	0.00891	**0.00889**

Execution time speedup

Table 5.9 displays the execution times for the sequential and multi-population configurations of the algorithms, expressed in seconds. Additionally, the speedup relative to the sequential alternative is annotated in subscripts. For the PSO-GA combined

version, the execution time of the sequential PSO algorithm serves as the reference for speedup calculations.

The first two columns list the base execution times for the sequential GA and PSO versions. It is observed that the GA algorithm completes executions faster than the PSO; however, as previously discussed, it does not perform as well in terms of RMSE (Table 5.8). The execution times for both the homogeneous and heterogeneous configurations of the algorithms are also provided. Notably, the distributed PSO versions achieve approximately a six-fold speedup, whereas the GA versions only reach up to a five-fold speedup in the heterogeneous configuration. Among these, the heterogeneous PSO-GA variant records the highest average speedup, outperforming the homogeneous PSO-GA configuration.

Figure 5.2 presents a box plot of these execution times, highlighting significant differences between the configurations. Statistical verification using a Z-test (n=30, $\alpha = 0.05$, independent samples) confirms the observed disparities for the GA (p=0.000171) and PSO-GA (p=0.000264) configurations, unlike the PSO, where the difference was not statistically significant (p=0.5104).

Table 5.9: Data from 30 observations regarding the execution time, expressed in seconds, for each algorithm evaluated. It includes results for the sequential implementations (displayed in the first two columns) and the distributed versions of the multi-population-based algorithms: Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and the hybrid PSO-GA. For each algorithm, the execution times for the homogeneous configurations are shown on the left side of the table, while those for the heterogeneous configurations appear on the right.

	Sequential		Homogeneous Parameters			Heterogeneous Parameters		
			Distributed					
	GA	**PSO**	**GA**	**PSO**	**PSO-GA**	**GA**	**PSO**	**PSO-GA**
Average	1999.34	2851.61	$421.67_{4.74}$	$415.64_{\mathbf{6.86}}$	$431.52_{6.60}$	$395.75_{5.05}$	$415.79_{6.84}$	$409.90_{\underline{6.95}}$
Std. Dev.	82.44	278.73	28.54	23.47	22.60	26.54	20.47	24.85
Median	1990.46	2770.62	$417.84_{4.72}$	$412.68_{\mathbf{6.71}}$	$432.36_{6.40}$	$392.80_{5.06}$	$414.66_{6.66}$	$408.48_{\underline{6.79}}$
Min	1876.40	2457.98	364.65	365.81	394.61	340.65	374.83	372.20
Max	2253.71	3822.01	526.76	467.47	478.80	461.46	461.96	465.27

5.3.2 Discussion

These findings confirm that employing a multi-population-based strategy not only facilitates a significant speedup but also maintains performance levels comparable to those of the sequential algorithms. Additionally, the combined PSO-GA algorithm offers better execution times, combining the continuous optimization capabilities of a PSO algorithm with the quicker execution of the GA metaheuristic. Notably, the heterogeneous configuration, which involves randomized parameter settings, further optimized execution times for the PSO-GA variant.

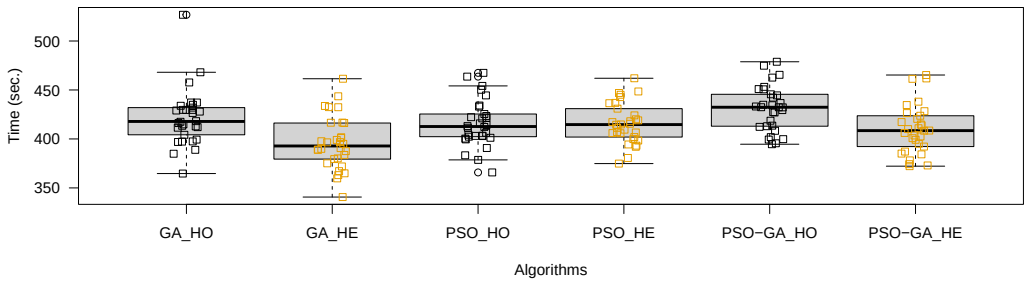


Figure 5.2: Box plot of the execution times, measured in seconds, from 30 trials of the distributed versions of the multi-population-based algorithms: Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and the hybrid PSO-GA. For each algorithm type, the homogeneous configurations are depicted on the left side of the plot, distinguished by black outlines and labeled with the suffix HO. Conversely, the heterogeneous configurations are shown on the right side, highlighted with orange outlines and marked by the suffix HE.

These experiments not only demonstrate the potential optimizations that can be achieved with distributed bioinspired algorithms but also highlight the significant speedup attainable through an implementation based on an event-driven architectural pattern. Another notable advantage of this approach is its reproducibility; using a standardized container-based deployment, other researchers can easily replicate the experiments under identical software conditions. Furthermore, the containerized setup facilitates the execution of the algorithms in cloud environments using the same code base and Docker scripts. This adaptability allows for the use

of more powerful virtual machines, thereby enhancing the scalability of the research framework.

5.4 Fuzzy Dynamic Adaptation of Parameters

Designers need to strategically balance exploitation and exploration when configuring the Particle Swarm Optimization (PSO) algorithm. The parameters affecting these search strategies are the Cognitive Coefficient (C_1), which controls how much the best personal solutions influence the swarm and the Social Coefficient (C_2) controlling how much the best global solution influences the swarm. As discussed before, achieving a balance between these coefficients is and is recognized as a significant challenge in current search and optimization methodologies [89].

We propose a fuzzy strategy that aims to enhance the balance between exploration and exploitation in a distributed PSO algorithm. We analyzed how two versions of a distributed parametrization strategy behave in a particular optimization problem [11] to evaluate if we can use this approach in future research [36]. Our objective is to enhance the design and overall performance of the current implementation. In this work, we use a technique called Dynamic Parameter Adaptation [90]. It is currently a popular method for enhancing population-based metaheuristics. The idea is to update the algorithm's parameters while it is in execution to get better results [10]. This work uses a knowledge-based system composed of fuzzy IF-THEN rules to dynamically adapt the parameters.

In this thesis, we introduce a fuzzy strategy designed to optimize the balance between exploration and exploitation within a distributed Particle Swarm Optimization (PSO) algorithm. Our approach involves examining how two distinct versions of a distributed parametrization strategy perform on the optimization problem at hand, with the goal of determining their potential applicability in future investigations [36]. The primary aim is to refine the design and enhance the overall efficacy of the existing algorithmic framework.

A key component of our methodology is the utilization of Dynamic Parameter Adaptation [90], a technique gaining traction for improving population-based meta-

heuristics. This method involves real-time adjustment of the algorithm's parameters during execution to achieve better results [10]. The adaptation of parameters is governed by a FIS designed for this purpose.

Effective exploration of the solution space is crucial in optimization tasks to prevent premature convergence to local optima, and to ensure a comprehensive search. This heuristic has the goal of maintaining a diverse population with varied characteristics to cover a broader area of the search space (see Figure 5.3). In this context, diversity within the population serves as an indicator of the search dynamics: low diversity implies that the particles are primarily exploiting the search space, whereas high diversity indicates active exploration [91].

Ideally, a high level of diversity is beneficial at the beginning of the search process to maximize the exploration potential. Conversely, reduced diversity towards the end of the search helps to refine the solutions to converge on optimal results.

Figure 5.3: Dynamic Parameter Adaptation

In a single swarm algorithm (Figure 5.4). We want to have a high diversity at the beginning of the run. For example, IF Generation is LOW then C_1 must be LOW while C_2 is HIGH. We expect to have high diversity at the beginning of the search because each particle is created with random values, and from there, diversity begins to decrease as the algorithm advances, in which case the particles begin exploring

and finish exploiting the search space [92]. At the beginning of the search, we want to set the cognitive coefficient (C_1) to a low value, and the social coefficient (C_2) will be high because we want to promote exploring the search space. With this knowledge, we propose a FIS (Figure 5.5), to adapt the coefficients C_1 and C_2 of the PSO algorithm, in which we have the number of cycles and diversity as input variables and C_1 and C_2 as output variables.

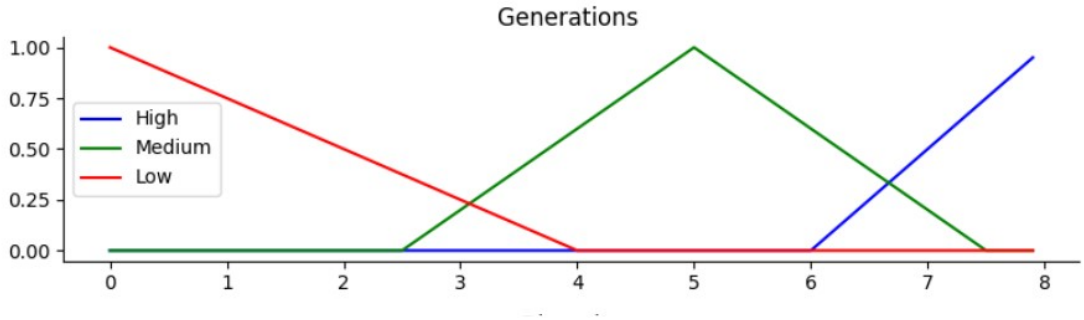


Figure 5.4: FM's a single swarm algorithm.

If we consider a PSO algorithm (Figure 5.4), initiating the run with high diversity is beneficial. This approach is conceptualized in a FIS as follows: if the generation count is low (indicating the early stages of the algorithm), then the cognitive coefficient (C_1) should be set low, while the social coefficient (C_2) should be high. This setting aids in promoting exploration over exploitation, which is crucial early on when each particle starts with random values, and overall diversity is naturally higher [92].

As the algorithm progresses, diversity typically diminishes as particles increasingly focus on exploring and eventually exploiting the search space. We propose the FIS depicted in Figure 5.5 to dynamically adjust to these changes. Because we are doing the adaptation on a multi-swarm PSO, this implementation takes the number of cycles of the distributed algorithm and the measure of diversity as inputs and adjusts the values of C_1 and C_2 accordingly. Such adaptive measures aim to ensure that the PSO algorithm maintains an effective balance between exploration and exploitation throughout its execution.

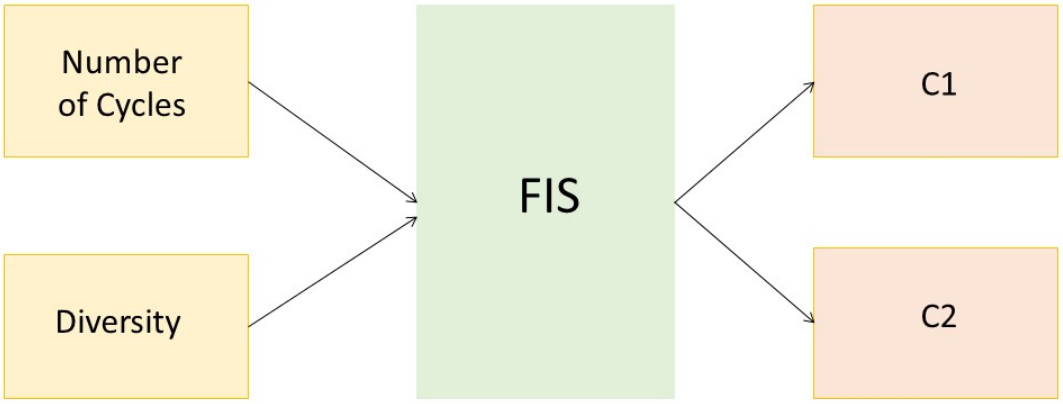


Figure 5.5: FIS for adaptation of parameters

Figure 5.6 provides an illustrative example of how diversity fluctuates across different generations within a Particle Swarm Optimization (PSO) algorithm. Diversity metrics are continuously monitored and used to adjust the cognitive coefficient (C_1) and the social coefficient (C_2) at each cycle. These adjustments occur during the combiner phase for each swarm, ensuring that the algorithm dynamically responds to changes in diversity throughout its execution.

As previously discussed, parameter adaptation can be implemented on each iteration of the local PSO algorithm during each cycle of the combiner. In our investigation, we explored both strategies to determine their efficacy. The findings indicated that adjusting parameters by cycles was equally effective as per-iteration adjustments (as shown in Figure 5.7). Consequently, we opted for cycle-based adaptation in our experiments due to its comparable performance benefits.

Following the proposed event-driven algorithm (Chapter 4), the dynamic adaptation of C_1 and C_2 after each swarm reaches the combinator module.

Figure 5.8 illustrates the membership functions used in our study, depicting their interrelations with the input variables (diversity and cycle) and the output variables C_1 and C_2. Through extensive experimentation, it was observed that the diversity metric for the swarms typically does not exceed 20 in most scenarios addressed. Based on these findings, we configured the experimental setup to include ten cycles of swarm combinations, which adequately provided the necessary number of function

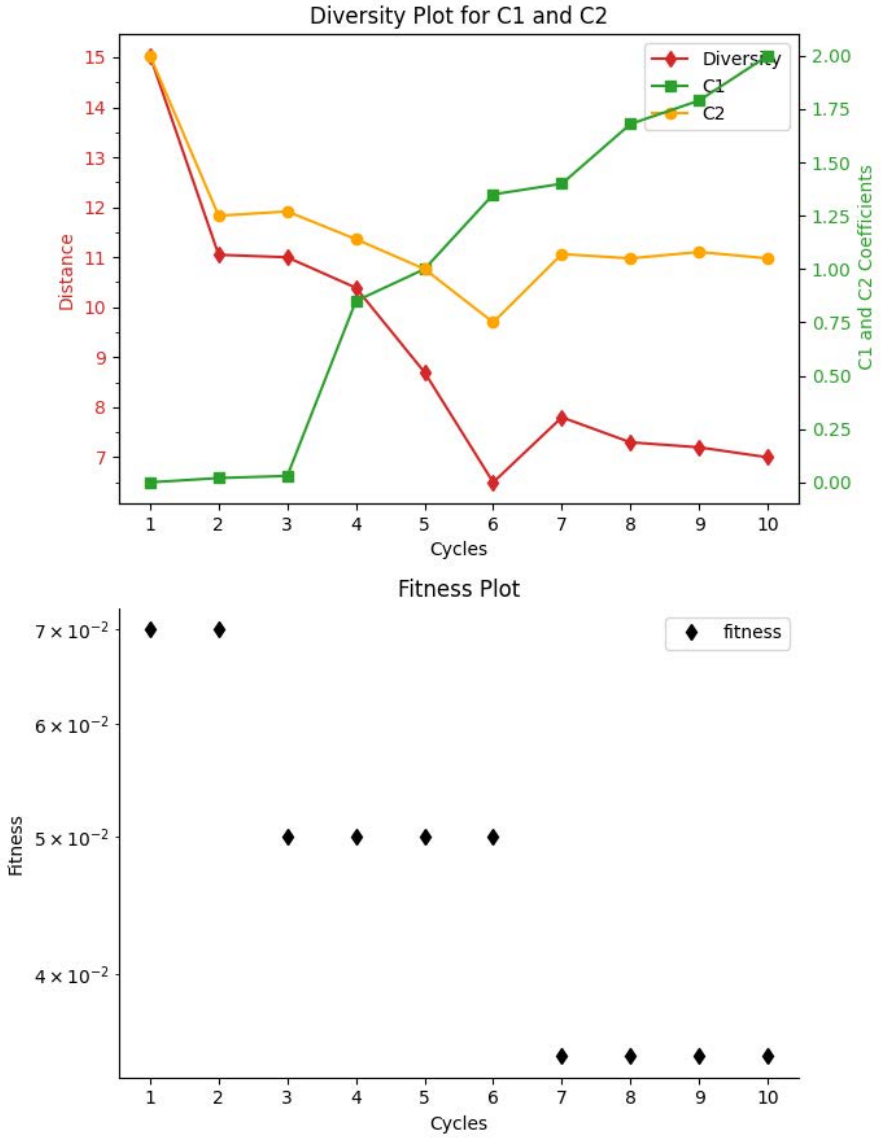
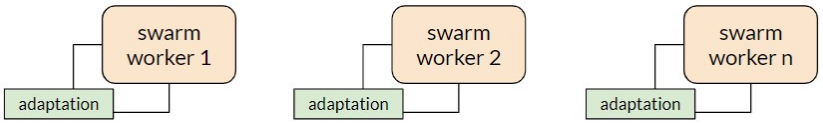
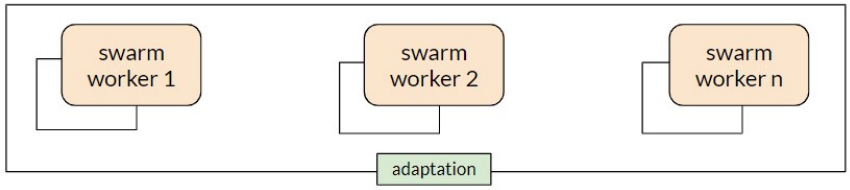


Figure 5.6: Example of changes of diversity in each cycle



(a) Adaptation after each iteration of the local level.



(b) Adaptation after each cycle of the combiner.

Figure 5.7: Parameter adaptation at different times.

evaluations for subsequent comparisons.

Consistent with established norms, we adjusted the values of the cognitive coefficient (C_1) and the social coefficient (C_2) to fall within the range of 1 to 2. This setting was chosen to optimize the balance between exploration and exploitation as dictated by the dynamic conditions of the swarm's diversity and the algorithm's progression through its cycles.

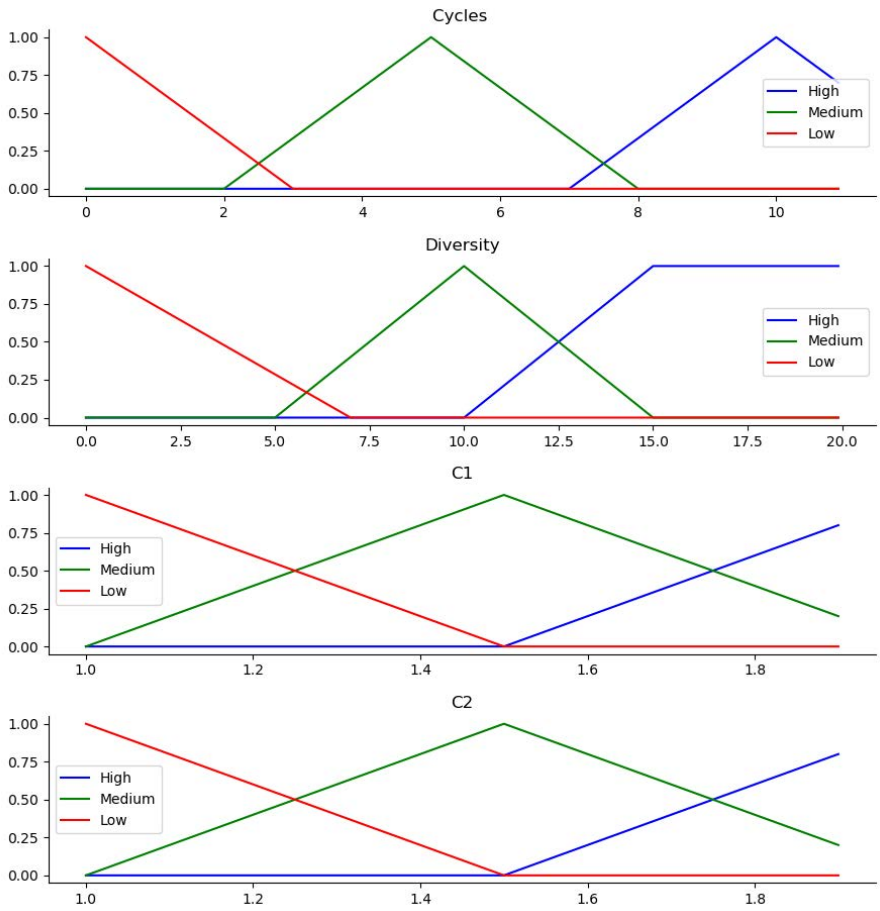


Figure 5.8: Membership Functions

The FIS implemented to adjust the parameters C_1 and C_2 is outlined in Table 5.10.

Table 5.10: Fuzzy Rules

R_1	if cycle is	hi	and diversity is	hi	then C_1 is	hi
					and C_2 is	low
R_2	if cycle is	hi	and diversity is	med	then C_1 is	hi
					and C_2 is	med
R_3	if cycle is	hi	and diversity is	low	then C_1 is	hi
					and C_2 is	low
R_4	if cycle is	med	and diversity is	hi	then C_1 is	med
					and C_2 is	hi
R_5	if cycle is	med	and diversity is	med	then C_1 is	med
					and C_2 is	med
R_6	if cycle is	med	and diversity is	low	then C_1 is	med
					and C_2 is	low
R_7	if cycle is	low	and diversity is	hi	then C_1 is	low
					and C_2 is	hi
R_8	if cycle is	low	and diversity is	med	then C_1 is	low
					and C_2 is	med
R_9	if cycle is	low	and diversity is	low	then C_1 is	low
					and C_2 is	hi

We optimized the membership functions for the control problem presented on Chapter 3. The parameters of the multi-swarm version of the PSO algorithm are shown in Table 5.11.

The multi-swarm is composed of sixteen swarms with ten particles each. Each local PSO will run for eight iterations of the algorithm before returning the resulting swarm state to the output queue. All swarms will complete ten cycles, meaning they will go through the combiner and back to the input queue a total of ten times each.

The multi-swarm configuration employed in this study consists of sixteen individ-

ual swarms, each containing ten particles. Each local PSO executes eight iterations before the resulting swarm is pushed back to the output queue. All swarms are designed to complete ten cycles. This entails each swarm passing through the combiner and returning to the input queue ten times.

Table 5.11: Experimental Setup

Algorithm	**Parameter**	**Range [min,max]**
MS-PSO	Communication Topology	Fully connected
	Speed.	Min=[-0.20, -0.30]
		Max=[0.20, 0.30]
	Cognitive and Social C_1, C_2	Dynamic by FIS
Swarms	Swarm Size	10
	Number of Swarms	16
	Number of Iterations	8
	Number of Cycles	10
	Dimension.	15
	#Funcion Evaluations	12800

5.4.1 Results

Table 5.12 presents the outcomes of our experiments, comparing the performance of distributed multi-swarm PSO configurations. Specifically, as explored in our preceding experiments, we compared swarms using dynamically adjusted parameters against those initialized with random parameters.

The results indicate that the dynamic parameter adaptation approach consistently outperforms the distributed version with heterogeneous parameters. This is evidenced by the lower average Root Mean Square Error (RMSE) recorded across thirty experimental runs, as detailed in Table 5.12.

A Z-test was conducted to statistically compare the performance of algorithms using heterogeneous parameters against those employing dynamic parameter adaptation. The results of this test are detailed in Table 5.13. The analysis concluded that there is insufficient evidence to reject the null hypothesis, as the p-values obtained did not fall within the traditionally accepted confidence intervals. This sug-

Table 5.12: Average RMSE obtained in thirty runs.

RMSE		
	Heterogeneous Parameters	Dynamic Parameter Adaptation
AVERAGE	0.00305315	0.00284276
STDDEV	0.00080043	0.00064820
MEDIAN	0.00290360	0.00271963
MIN	0.00127626	0.00168746
MAX	0.00496247	0.00448633

gests that the differences in performance between the two configurations might not be statistically significant.

Table 5.13: Statistical test: Z-test.

p-values, α=0.05, 30 independent samples, unequal variances.		
$H_a : \mu_{eAi} < \mu_{eAj}$	Heterogeneous Parameters	Dynamic Parameter Adaptation
Heterogeneous Parameters		0.8659
Dynamic Parameter Adaptation	0.1341	

Figure 5.9 visually represents the experimental results using a box plot. This graphical depiction further corroborates that there is no significant difference in the median values between the two algorithm configurations being compared.

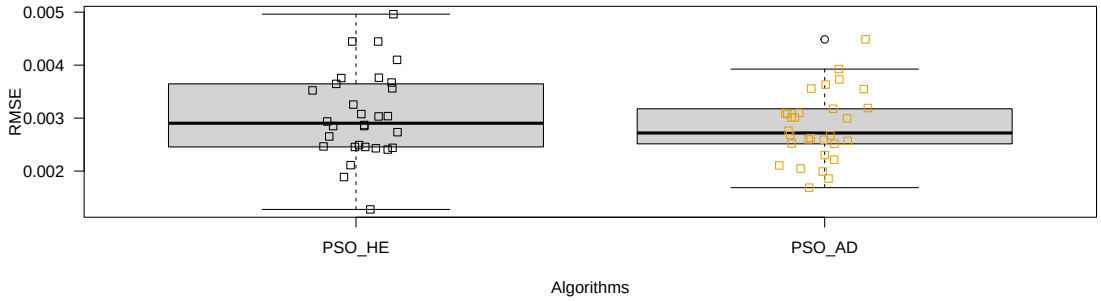


Figure 5.9: Box plot of algorithms that use heterogeneous parameters versus those that use dynamic parameter adaptation.

5.4.2 Discussion

We introduced a dynamic parameter adaptation technique using fuzzy logic within a distributed multi-swarm PSO framework. The system was specifically applied to optimize the tuning of MF parameters in a fuzzy controller. To evaluate the efficacy of our proposed method, we conducted experimental comparisons between two distinct strategies: fuzzy adaptation and heterogeneous parameter configurations.

The results indicated that both strategies performed comparably in terms of optimizing the search space. This finding suggests that while the fuzzy adaptation approach is valid, further research could explore additional metrics to better gauge the optimal balance between exploration and exploitation within the search space.

Chapter 6

Cloud Implementation

The adoption of cloud-native technologies for distributed computing presents a set of unique challenges. It is important to take into account the technologies offered by the cloud provider to gain the benefits of the architecture while preserving the intended functionality. This chapter introduces a container-based implementation of a cloud-native multi-population PSO on Amazon's Elastic Container Service. The study examines the impact of the design parameters, such as the number of swarms and the number of worker containers implementing isolated PSO algorithms. Furthermore, the chapter proves that the proposed solution offers a robust alternative for both local and cloud environments. The results underscore the benefits of containerization for cloud-based bioinspired computing.

Building on the foundation of our previous implementations, this chapter details the deployment process of a reactive, container-based, multi-swarm PSO algorithm specifically designed for deployment in Amazon's AWS. This design decomposes the previous application into docker containers using an event-driven architecture. We again use the optimization problem described in Chapter 3. This optimization problem is computationally intensive, primarily because it requires evaluating the fitness of all potential solutions. Despite the computational demands of this optimization problem, the independent nature of solution evaluation in evolutionary algorithms allows for the efficient parallelization of work. This examination focuses on the intricacies of deploying such an algorithm within a cloud environment, leveraging the

scalability and flexibility of AWS services. The use of containerization is a key feature of this deployment because it offers the benefit of environment consistency and scalability. This approach aligns with the principles of reactive systems, ensuring that the deployed algorithm is efficient in resource usage and robust and adaptable to varying computational loads. Furthermore, our application can be easily replicated and even run with other cloud providers.

In summary, our approach addresses a fuzzy control problem through the application of a cloud-native distributed Particle Swarm Optimization (PSO) algorithm characterized by its minimal set of tunable parameters. We emphasize the significance of our design choices and elucidate the deployment process leveraging various cloud technologies provided by Amazon's AWS.

In this chapter, we also implement a cost analysis, comparing multiple local and cloud deployment configurations. By scrutinizing the associated costs, we aim to provide insights into the economic considerations associated with implementing our proposed solution in different computing environments.

6.1 Cloud-based Deployment

We have several options for the deployment of cloud-native application to AWS, the most common approach is to use single or several virtual servers using EC2 instances. In our current deployment, we opted to use Amazon Elastic Container Service (ECS).

ECS is a fully managed container orchestration service, and it allows the user to run, stop, and manage Docker containers on a cluster, simplifying the process of deploying, managing, and scaling containerized applications. ECS uses *Task Definitions* as a blueprint for a set of containers that run together on the same host. It defines parameters such as which Docker images to use, the CPU and memory requirements, and networking information, among others.

A task definition is similar to a `docker-compose` file in the sense that defines the interaction of several containers. To minimize the need to manage the underlying infrastructure, we use what is called `Fargate Tasks`, a serverless compute engine

that eliminates the need for launching and managing EC2 instances ourselves.

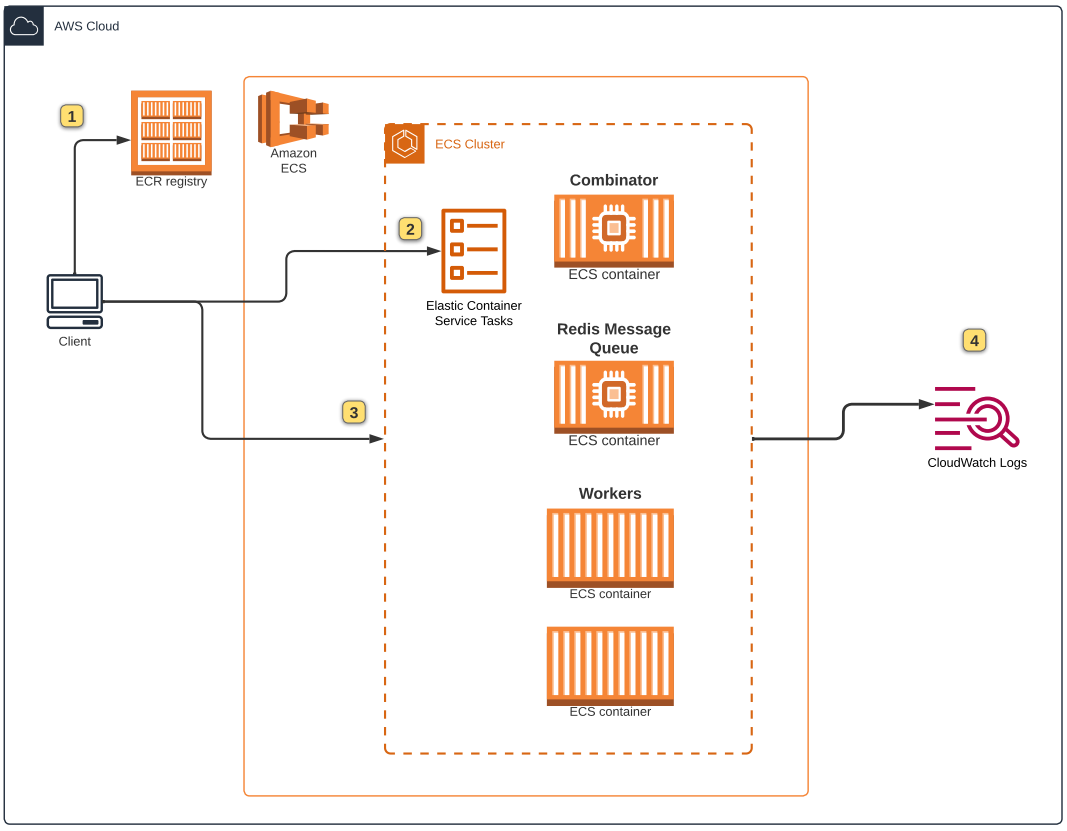


Figure 6.1: Implementation using AWS cloud technologies

We only need to specify the resource requirements needed for our task. Figure 6.1, shows the main cloud computing technologies from Amazon's AWS used in the deployment of the multi-swarm algorithm. We had specified a Docker container definition for each component described previously using a `Docker` file. With this Docker definition, we could run the algorithm locally using a `docker-compose` script. As a first step for cloud deployment, we must create a repository for each container definition in the Amazon Elastic Container Registry (ECR) (1). ECR is a fully managed container registry service provided by AWS. This registry stores, manages, and deploys Docker container images. The advantage of having an image repository

is that we can keep several versions of each image. As a second step (2), we defined a Fargate Task Definition for each component. We then created an ECS cluster, a logical grouping of container instances or tasks to run our containerized applications. We now run the Task Definitions (3) and observe the execution logs in Amazon CloudWatch (4); this is a monitoring and observability service. These steps can be realized using a browser-based user interface or using a command-line interface.

Each of the services mentioned above has a cost that is charged monthly, the current prices are shown in Table 6.1. First, we must consider the cost of storing the Docker container images in the ECR. We have 50GB a month for free, but it is important to remember that some container images can grow to several GB depending on the operating system, languages, libraries, and software included in the container. The data transfer cost for a private registry is negligible; there is no cost for uploading the images and no data transfer to the outside. When we specify the computing resources for running a certain Fargate Task, we select the number of vCPUs to assign, and depending on our selection, there is a minimum and maximum amount of memory we can assign to the task. At this time, when selecting 1 vCPU, the minimum memory is 2 GB, and the maximum is 8 GB, in 1 GB increments. We selected 3 GB memory for our containers. To run these experiments, we did not need additional ephemeral storage; we used the 20 GB included for free. Finally, we used the CloudWatch service extensively to keep track of the algorithm and summarize the results.

To test the deployment, we are comparing the execution time of three configurations running on Amazon's Elastic Container Service against a local execution using a Mac Studio workstation. We used the same resource-intensive optimization problem described in Chapter 3.

The current implementation uses the Python language, which means that even if there are multiple CPUs in the computer, the Python interpreter process uses only a single thread or CPU at a time. For this reason, we only run workers in Fargate instances using a single vCPU. Each vCPU is a hyper thread of an Intel Xeon CPU core. We validated this by running a few experiments using instances with 4vCPUs, but we did not find a substantial difference in the compute time, so the additional

Table 6.1: Cost of AWS Fargate Services, the Elastic Container Registry is needed to upload images for creating and running the containers; each container runs in a Task using one vCPU, 3 GB of RAM, and 20GB of ephemeral Storage. The results of the experiment are stored in Amazon Cloud Watch logs.

Service	Price	Free tier
Elastic Container Registry		
Storage	$0.10 per GB	50 GB
All data transfer IN	$0.00 per GB	
Data Transfer OUT		
Next 9.999 TB / month	$0.09 per GB	
AWS Fargate		
per vCPU per hour	$0.040480	
per GB per hour	$0.004445	
Ephemeral Storage GB per hour	$0.000111	20 GB
Amazon Cloud Watch		
Collect (Data Ingestion)		
Standard	$0.50 per GB	0 to 5 GB
Infrequent Access	$0.25 per GB	0 to 5 GB

cost is not justified.

For the cloud configurations, we compare two configurations: 36 and 40 workers. This means 36 and 40 vCPUs. This number of processors is higher than what we normally find in a PC Workstation. We are comparing the results with a local implementation using a high-end workstation, in this case, a 2022 Mac Studio with an Apple M1 Ultra chip with 20 CPUs (16 performance and 4 efficiency). We use 16 workers in this configuration to fully exploit the maximum number of performance cores.

We normally set the number of swarms and the number of workers with a similar value, if we increase the number of swarms, many will remain in the `input-queue` waiting for a worker to be available. If we have more workers than swarms, we will have workers waiting for swarms to be available. To test how much the ratio of swarms/workers affects the time of execution, we test with these three configurations: the same number of swarms and workers with 36 each, 36 workers and 40

swarms, and finally, 40 workers and 36 swarms.

The multi-swarm is composed of 36 or 40 swarms with ten particles each. Although this is a small size for a traditional single swarm PSO, in the case of an MS-PSO, the total size of the swarm is the sum of all swarms. Each local PSO will run four iterations of the algorithm before returning the resulting swarm state to the `output-queue`. All swarms will complete ten cycles, meaning they will go through the combiner and back to the `input-queue` a total of ten times each.

In Table 6.2 are shown the parameters of the multi-swarm version of the PSO algorithm.

Table 6.2: Experimental Setup, these values were obtained from initial experiments and these are the same parameters used in our previous work [52]. The number of Function Evaluations are different because some parameters are adjusted to the number of swarms or workers used.

Algorithm	**Parameter**	**Range[min,max]**	
MS-PSO	Communication Topology	Fully connected	
	Speed.	Minimum = $[-0.20, -0.30]$	
		Maximum = [0.20, 0.30]	
	Coefficients C_1, C_2	[1.0, 2.0]	
	Local	Cloud	
Size	10	10	10
Swarms	16	36	40
Iterations	8	4	4
Cycles	10	10	10
Dimensions	15	15	15
# Function Evaluations	12800	14400	16000

We use the number of function evaluations per second (EPS) as a metric to compare the time-to-solution performance. As we mentioned before, the fitness evaluation function is the most computationally demanding component of the PSO algorithm, and this is why the number of calls to this function is a commonly used metric to evaluate the amount of work needed to find a solution independently of the parameters or hardware used. In this case, we change the parameters of the MS-PSO algorithm depending on the number of swarms and workers available. These changes in the configuration yield different numbers of function evaluations for each configuration. To consider this, we compare the execution speed by EPS, higher

values are better.

6.2 Results

Table 6.3 shows the results for the proposed configurations, giving the best RMSE obtained, the total time (in seconds), and the function evaluations per second of the experiments. We also included the cost per execution according to the number of workers and the time to finish, considering only the vCPU and Memory costs. We can see a significant increase in the #FE per second when the number of swarms equals the number of workers.

Table 6.3: Function Evaluations per second comparison between local and three configurations on AWS Elastic Container Service

	Local	*AWS Elastic Container Service*		
Configuration	Mac Studio	1	2	3
Workers	16	36	36	40
Swarms	10	36	40	36
#FE	12800	14400	16000	14400
RMSE	0.00271	0.002935	0.002858	0.002620
Time in seconds	754.968	884.050	1084.470	938.231
Evals. per second	16.954	16.289	14.753	15.348
Cost per run (USD)		0.4524	0.570	0.559

These results can be explained by analyzing the timeline of three representative runs.

First, we describe the format in which the results are presented. Figures 6.2 to 6.4 shows the timelines for the proposed configurations. The x-axis shows the time elapsed in seconds from the beginning of the experiment. Each row represents a particular worker, identified by a sequential number. Each box represents the time range in seconds, beginning when a swarm is pulled from the `input-queue` until pushed to the `output-queue`. A sequential integer identifies each swarm. The elapsed time is not the same in all cases because the time to complete the path

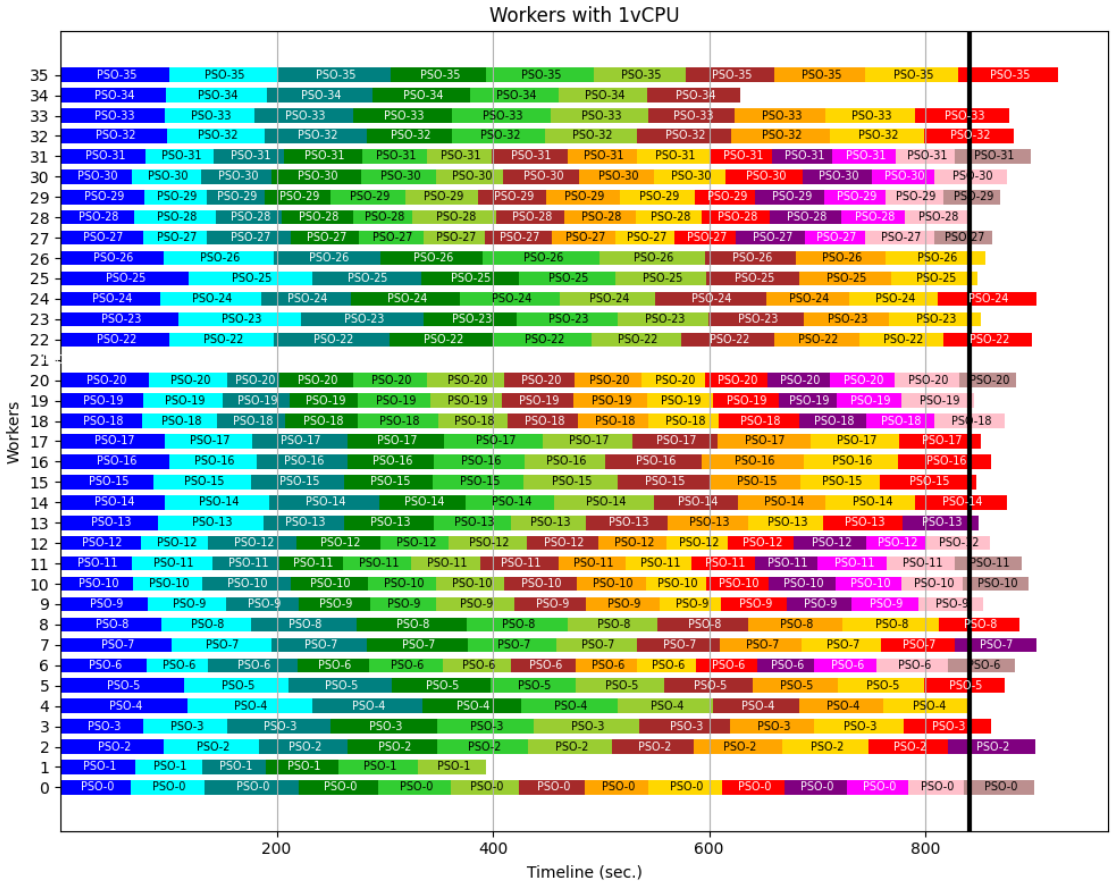


Figure 6.2: Execution of 36 workers and 36 swarms. The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container.

depends on the controller's performance. Moreover, when the error or distance to the path is large, the simulation is interrupted, thus taking less time. To identify each swarm, the color corresponds to the order in which each worker received it, and this is to avoid those cases where the same swarm is sequentially received by the same worker twice. In this case, it is difficult to distinguish between the two boxes if the swarm is of the same color. The vertical bold line indicates when the number of evaluations has been reached. At this point, no more swarms are pushed to the `input-queue`. Because workers take swarms asynchronously, the work already in

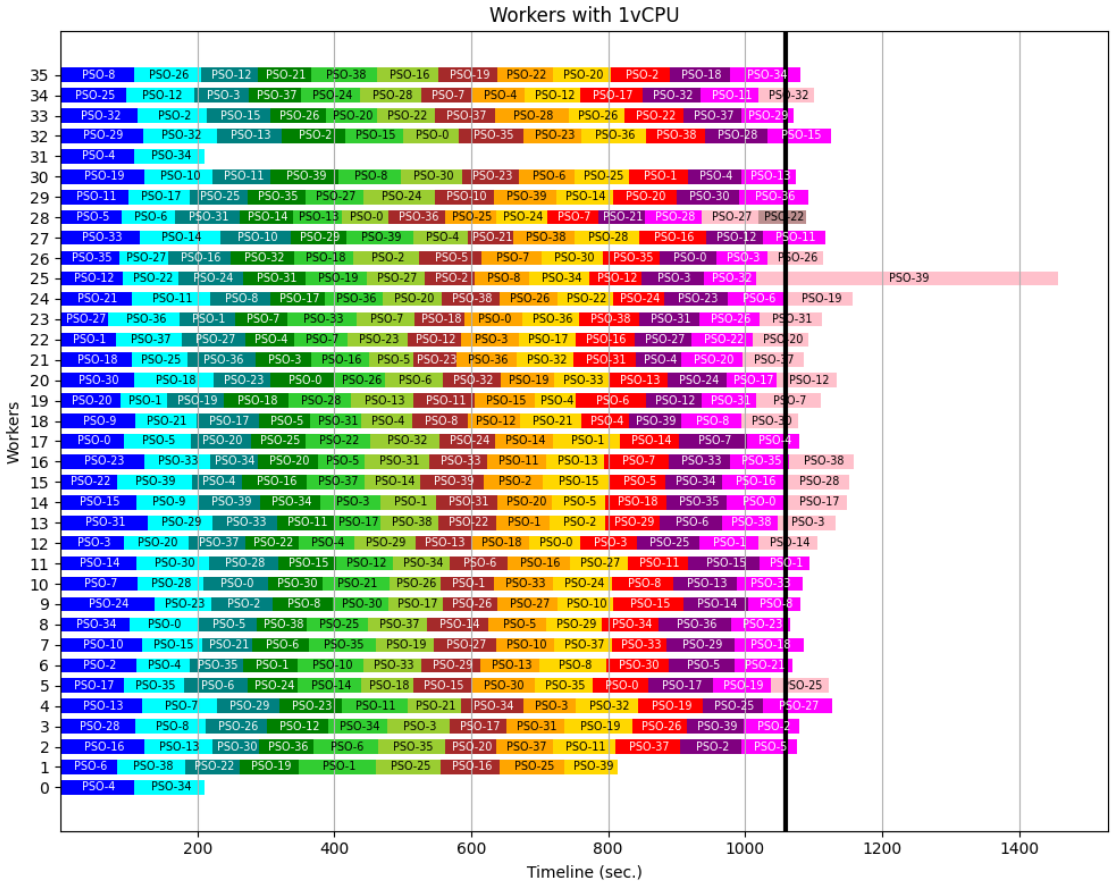


Figure 6.3: Execution of 36 workers and 40 swarms. The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container.

the `input-queue` is still pulled by workers, and the work they are processing at that time is also finished. This is why more work is processed beyond the bold line.

Figure 6.2 shows the timeline for the experiment using 36 workers and 36 swarms. When we have this configuration, it is common for each swarm to be executed exclusively by the same worker. This is because the `input-queue` will be empty most of the time. Once a worker finishes processing a certain swarm, it will be available for the next swarm in the `input-queue`. The `combinator` returns the same swarm to the `initial-queue`, and the same worker is available to take the same swarm again.

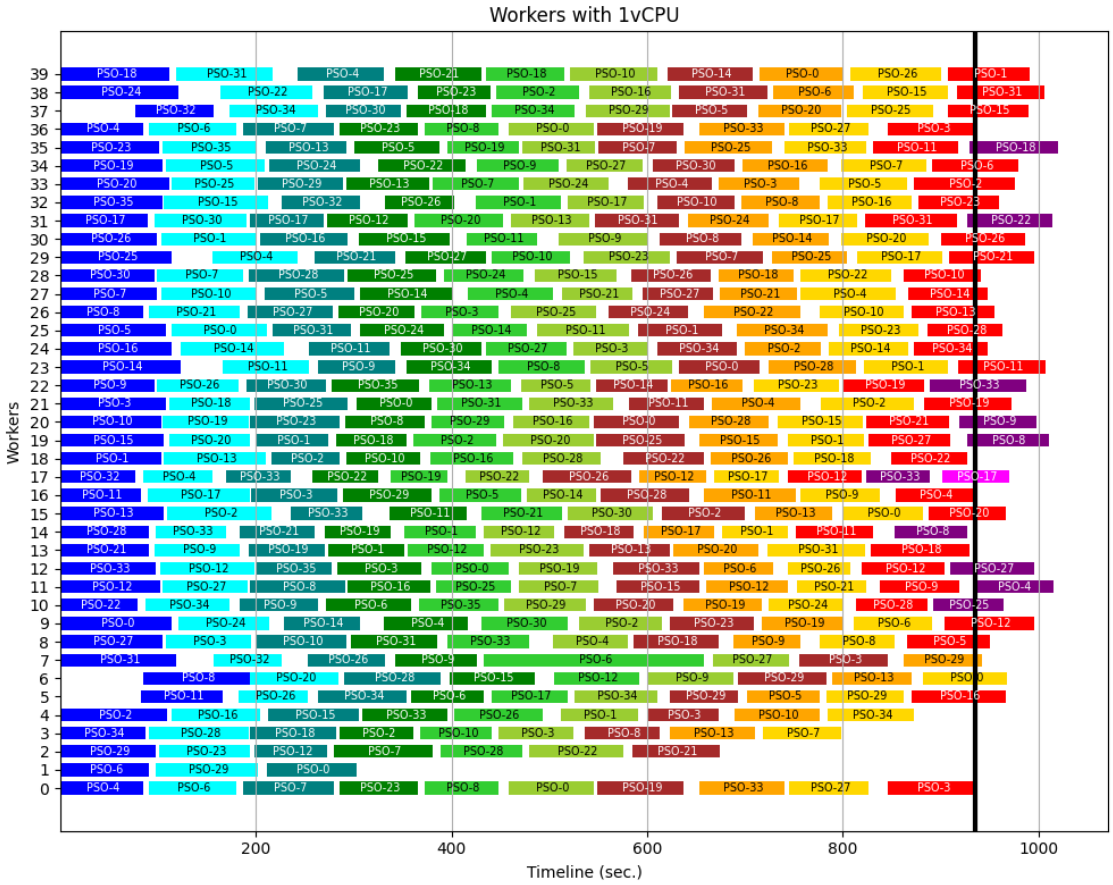


Figure 6.4: Execution of 40 workers and 36 swarms. The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container.

Furthermore, when this condition happens, workers do not have a noticeable waiting time. We can see that this happens in all the workers. It is important to notice that *worker 21* in Figure 6.2 takes a swarm, but this swarm is never pushed back into the `output-queue`. This could be because of an error in the swarm configuration or a bug in the code. Currently, we do not have a fault-tolerant solution, for instance, to restart a worker if the work is not returned after a certain amount of time. But in any case, the nature of the event-driven solution still works well. The extra work needed to reach the number of evaluations needed to complete the experiment is distributed

among all the other workers. Figure 6.3 shows the case when we have more swarms than workers; consequently, swarms are not always processed by the same worker as in the previous case. We need to assess in a future work if this behavior benefits the search algorithm. The drawback of this case is that the work takes more time because we have fewer resources to do the computation. Finally, Figure 6.4 shows the case where we have spare workers. The problem with this approach is that the workers must wait until a new population is available. For example, *worker 5* needs to wait until a swarm is available. In this case, population PSO-11 is received by *worker 5* after *worker 16* finished working on the swarm. Nevertheless, even if we have more computing resources,due to the additional downtime of workers, the total time is not the best of the three configurations mentioned.

When comparing against the local execution, results indicate that we needed about 2.25 times the number of workers to match the time-to-finish performance of a local execution. There are several factors that could have an impact on this difference. The communication time for a local container and the network is faster than in a cloud environment; the M1 chip tied with a fast memory bandwidth (800GB/s) is faster than a vCPU. Moreover, there could be differences in the virtualization environment in which the docker host runs; the Docker engine used in Mac Studio is optimized for the M1 chip. On the other hand, running these types of experiments in a cloud environment does not incur an initial investment on a workstation computer.

6.3 Statistical Z-test

The results of a statistical Z-test for Root Mean Square Error (RMSE) are presented in Table 6.4. In the table, the alternative hypothesis (Ha) states that the mean RMSE ($\mu_e A_i$) for one group is less than that for another group ($\mu_e A_j$). For instance, the comparison between 36 workers with 36 swarms versus 40 swarms results in a p-value of 0.6624, which is not less than 0.05, suggesting that the difference is not statistically significant. However, a p-value of 0.0585 for the comparison between 40 workers with 36 swarms and 36 workers with 36 swarms is quite close to

the threshold, which might warrant further investigation or consideration of other factors.

Table 6.4: RMSE Statistical Z-test.

p-values, α=0.05, independent samples, unequal variances, 30 samples			
$H_a : \mu_{eAi} < \mu_{eAj}$	36 workers 36 swarms	36 workers 40 swarms	40 workers 36 swarms
36 workers 36 swarms		0.6624	0.9415
36 workers 40 swarms	0.3376		0.8811
40 workers 36 swarms	0.0585	0.1189	

We show a box plot of the data in Figure 6.5, and we can see that there is no significant difference between the median of the same algorithms.

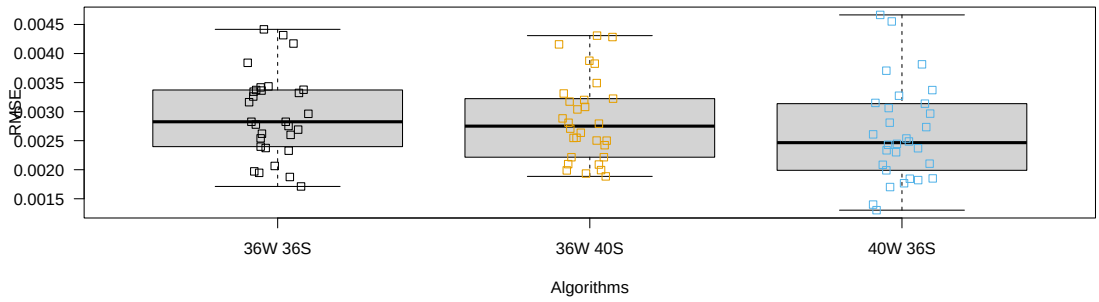


Figure 6.5: Box plot of RMSE results for different Cloud configurations.

We show in Table 6.5 the results of the statistical Z-test, where we compared the execution time in seconds for completing the total number of evaluations made in each configuration group. In each group, we changed the number of workers and the number of swarms. We conclude that in the controller configuration with 36 workers and 36 swarms, there is sufficient evidence to accept the alternative hypothesis (H_a) that states that this configuration executes faster than the other two. Also, there is evidence suggesting that the worst configuration is 36 workers with 40 swarms.

Table 6.5: Time in sec. Statistical Z-test.

p-values, α=0.05, independent samples, unequal variances, 30 samples			
$H_a : \mu_{tAi} < \mu_{tAj}$	36 workers 36 swarms	36 workers 40 swarms	40 workers 36 swarms
36 workers 36 swarms		2.20E-16	0.002385
36 workers 40 swarms	1		1
40 workers 36 swarms	0.9976	9.81E-11	

We show a box plot of the data in Figure 6.6, and we can see that there is a significant difference between the median of the execution time of some configurations.

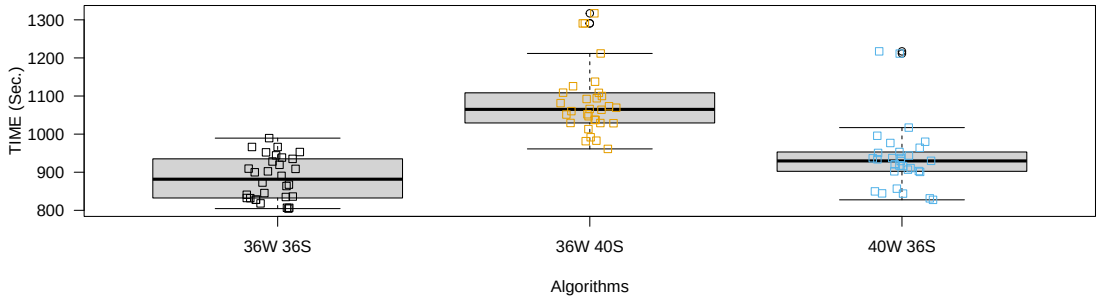


Figure 6.6: Box plot of execution time in seconds results for different Cloud configurations

6.4 Discussion

We compared three configurations to highlight the differences between worker and swarm ratios. We discovered that the most effective configuration with respect to RMSE is the configuration with more workers and fewer swarms, but if we want to minimize the execution time then the configuration with the same number of workers and swarms proved to be the best. We found that for certain use cases, the number of workers needs to be increased when deploying the algorithm to a cloud

provider. Cloud deployment of cloud-native implementation of population-based algorithms is a cost-effective alternative without investing in additional hardware or administrative costs.

For future research, we aim to expand upon the design options on the algorithmic side and on the deployment side. On the algorithm, we can dynamically change the size or number of swarms (and workers) to ascertain potential performance gains. When implementing these options, we can test the auto-scaling features of the cloud platform with the intent to optimize resource utilization without degrading the algorithm's exploratory capacities.

Chapter 7

Conclusions and future work

In this work, we focused on the optimization of controllers using metaheuristics in a distributed and asynchronous manner. Our contributions covered the design, development, and implementation of a distributed and asynchronous architecture, where we use different metaheuristics and implement them following cloud-native principles to improve their performance and execution time.

We adapted a simulator software for evaluating fuzzy controllers and proposed a parametrizable controller. We compared several state-of-the-art evolutionary algorithms for optimizing the membership functions using a technique we developed. We found that the PSO algorithm gave the best results for this optimization problem.

According to the results of these first experiments, we observed that we could optimize fuzzy controllers by using bioinspired algorithms in a centralized manner. Still, we found that this has a high computational cost. Therefore, as we progressed in the work, we proposed architecture that follows some of the cloud-native techniques used today. Using container technologies to implement a distributed worker alternative to the classical Island Model found in distributed evolutionary algorithms.

We proposed and validated different parameterization techniques of bioinspired optimization algorithms in a distributed and asynchronous manner. With this, we observe performance improvements in the execution time of the algorithms without increasing the RMSE of the resulting controllers.

To improve the distributed muli-population model, we tested with homogeneous, heterogeneous and dynamic parametrization of the algorithms, as well as various configurations in the parameters of the algorithms, we performed statistical tests to compare these strategies to determine which configurations were the better. With this, we observe performance improvements in the execution time of the algorithms without increasing the RMSE of the resulting controllers.

Based on the results obtained in previous experiments, we adapted to the best configuration and deployed our cloud-native solution to a cloud provider. We compared the three cloud configurations to highlight the differences between the proportions of workers and swarms (populations), observing that there is not a significant difference with respect to RMSE on these configurations. Nevertheless, the most effective configuration with respect to execution time is the configuration with the same number of workers and swarms. We also noted that, for certain use cases, when deploying to a cloud provider, increasing the number of workers is necessary to achieve better execution times when we compare the results against a high-end workstation running locally.

We concluded that deploying population-based algorithms in the cloud is a cost-effective alternative as you don't have to invest in additional hardware or administrative costs.

For all of the above, we can say that this research can be taken up by other researchers and reproduced, we can expand the design options on the algorithmic and the implementation side. The algorithms should be improved to achieve faster execution time and minimize error. We can also try other metaheuristics and dynamically change the size or number of swarms (and workers) to determine possible performance gains. By implementing these options, we can test the auto-scaling features of the cloud platform with the intention of optimizing resource utilization without degrading the exploratory capability of the algorithm.

Appendix A

Publications

A.1 Conference Publications

1. *Mancilla, A., Castillo, O., Valdez, M. G. (2023). Fuzzy Adaptation of Parameters in a Multi-Swarm Particle Swarm Optimization (PSO) Algorithm Applied to the Optimization of a Fuzzy Controller. New Horizons for Fuzzy Logic, Neural Networks and Metaheuristics. (accepted for publication)*

2. *Mancilla, A., Castillo, O., García-Valdez, M. (2023) Optimization of Fuzzy Controllers using Distributed Bioinspired Methods with Random Parameters, Hybrid Intelligent System Based on Extensions of Fuzzy Logic, Neural Networks and Metaheuristics (pp. 189-197). Springer, Cham.*

3. *Mancilla, A., Castillo, O., Valdez, M. G. (2022). Mixing Population-Based Metaheuristics: An Approach Based on a Distributed-Queue for the Optimal Design of Fuzzy Controllers. In International Conference on Intelligent and Fuzzy Systems (pp. 839-846). Springer, Cham.*

4. *Mancilla, A., Castillo, O., Garcia-Valdez, M. (2022). Optimization of Fuzzy-Control Parameters for Path Tracking of a Mobile Robot Using Distributed Genetic Algorithms. In New Perspectives on Hybrid Intelligent System Design based on Fuzzy Logic, Neural Networks and Metaheuristics (pp. 167-177). Springer, Cham.*

5. *Mancilla, A., Castillo, O., Valdez, M. G. (2021, August). Evolutionary Approach to the Optimal Design of Fuzzy Controllers for Trajectory Tracking. In International Conference on Intelligent and Fuzzy Systems (pp. 461-468). Springer, Cham.*

6. *Mancilla, A., Castillo, O., Valdez, M. G. (2021). Optimization of Fuzzy Logic Controllers with Distributed Bio-Inspired Algorithms. In Recent Advances of Hybrid Intelligent Systems Based on Soft Computing (pp. 1-11). Springer, Cham.*

A.2 Journal Publications

1. *García-Valdez, M., Mancilla, A., Castillo, O., Merelo-Guervós, J. J. (2023). Distributed and Asynchronous Population-Based Optimization Applied to the Optimal Design of Fuzzy Controllers. Symmetry, 15,467.*

2. *Mancilla, A., García-Valdez, M., Castillo, O., Merelo-Guervós, J. J. (2022). Optimal Fuzzy Controller Design for Autonomous Robot Path Tracking Using Population-Based Metaheuristics. Symmetry, 14(2), 202.*

Appendix B

Tools used in this thesis

- Main Git repository that includes the software developed for this thesis
  url: `https://github.com/mariosky/fuzzy-control`
- Python language
  url: `https://www.python.org/`
- Numpy - The fundamental package for scientific computing with Python
  url: `https://numpy.org`
- Docker
  url: `https://www.docker.com/`
- Amazon AWS
  url: `https://aws.amazon.com/`
- PyCharm: the Python IDE for Professional Developers by JetBrains
  url: `https://www.jetbrains.com/pycharm/`
- R: The R Project for Statistical Computing
  url: `https://www.r-project.org/`
- Matplotlib — Visualization with Python
  url: `https://matplotlib.org/`

- Overleaf - Online LaTeX and Rich Text collaborative writing and publishing tool
  url: `https://www.overleaf.com/project`

References

[1] R. D. H. Beleño, G. B. Vítor, J. V. Ferreira, and P. S. Meirelles, "Planeación y Seguimiento de Trayectorias de un Vehículo Terrestre con Base en el Control de Dirección en un Ambiente Real," *Scientia et technica*, vol. 19, no. 4, pp. 407–412, 2014. Publisher: Universidad Tecnológica de Pereira.

[2] A. Mancilla, O. Castillo, and M. G. Valdez, "Optimization of fuzzy logic controllers with distributed bio-inspired algorithms," *Recent Advances of Hybrid Intelligent Systems Based on Soft Computing*, pp. 1–11, 2021.

[3] O. Castillo, L. Amador-Angulo, J. R. Castro, and M. Garcia-Valdez, "A comparative study of type-1 fuzzy logic systems, interval type-2 fuzzy logic systems and generalized type-2 fuzzy logic systems in control problems," *Information Sciences*, vol. 354, pp. 257–274, 2016.

[4] M. G. Valdez and J. J. M. Guervós, "A container-based cloud-native architecture for the reproducible execution of multi-population optimization algorithms," *Future Generation Computer Systems*, vol. 116, pp. 234–252, 2021.

[5] M. García-Valdez, L. Trujillo, F. F. De Vega, J. J. M. Guervós, and G. Olague, "Evospace: a distributed evolutionary platform based on the tuple space model," in *European conference on the applications of evolutionary computation*, pp. 499–508, Springer, 2013.

[6] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. F. De Vega, and G. Olague, "The evospace model for pool-based evolutionary algorithms," *Journal of Grid Computing*, vol. 13, no. 3, pp. 329–349, 2015.

[7] J. J. Merelo, C. M. Fernandes, A. M. Mora, and A. I. Esparcia, "SofEA: A pool-based framework for evolutionary algorithms using couchdb," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pp. 109–116, 2012.

[8] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016. Publisher: IEEE.

[9] A. Mancilla, M. García-Valdez, O. Castillo, and J. J. Merelo-Guervós, "Optimal fuzzy controller design for autonomous robot path tracking using population-based metaheuristics," *Symmetry*, vol. 14, no. 2, p. 202, 2022.

[10] A. Mancilla, O. Castillo, and M. García-Valdez, "Optimization of fuzzy controllers using distributed bioinspired methods with random parameters," in *Hybrid Intelligent Systems Based on Extensions of Fuzzy Logic, Neural Networks and Metaheuristics*, pp. 189–197, Springer, 2023.

[11] M. García-Valdez, A. Mancilla, O. Castillo, and J. J. Merelo-Guervós, "Distributed and asynchronous population-based optimization applied to the optimal design of fuzzy controllers," *Symmetry*, vol. 15, no. 2, p. 467, 2023.

[12] A. Mancilla, O. Castillo, and M. García-Valdez, "Fuzzy adaptation of parameters in a multi-swarm particle swarm optimization (pso) algorithm applied to the optimization of a fuzzy controller," in *New Horizons for Fuzzy Logic, Neural Networks and Metaheuristics*, Springer, 2024, to be published.

[13] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, pp. 35–45, 03 1960.

[14] K. Fu, "Learning control systems and intelligent control systems: An intersection of artifical intelligence and automatic control," *IEEE Transactions on Automatic Control*, vol. 16, no. 1, pp. 70–72, 1971.

[15] E. H. Mamdani, "Application of fuzzy algorithms for control of simple dynamic plant," *Proceedings of the institution of electrical engineers*, vol. 121, no. 12, pp. 1585–1588, 1974.

[16] D. Driankov and A. Saffiotti, *Fuzzy logic techniques for autonomous vehicle navigation*, vol. 61. Physica, 2013.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] A. Bisoffi, C. De Persis, and P. Tesi, "Data-driven control via petersen's lemma," *Automatica*, vol. 145, p. 110537, 2022.

[19] H.-S. Ahn, Y. Chen, and K. L. Moore, "Iterative learning control: Brief survey and categorization," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 6, pp. 1099–1121, 2007.

[20] S. L. Brunton and J. N. Kutz, *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

[21] R. Babuška, "Fuzzy systems, modeling and identification," *Delft University of Technology, Department of Electrical Engineering Control Laboratory, Mekelweg*, vol. 4, 1996.

[22] M. G. Simoes, "Introduction to fuzzy control," *Colorado School of Mines, Engineering Division, Golden, Colorado*, vol. 8, pp. 18–22, 2010.

[23] O. Castillo and P. Melin, "Soft computing for control of non-linear dynamical systems," *Applied Soft Computing*, vol. 3, p. 301, 2003.

[24] C. Caraveo, F. Valdez, and O. Castillo, "Optimization of fuzzy controller design using a new bee colony algorithm with fuzzy dynamic parameter adaptation," *Applied Soft Computing*, vol. 43, pp. 131–142, 2016.

[25] A. Cárdenas Cardona, "Inteligencia artificial, métodos bio-inspirados: un enfoque funcional para las ciencias de la computación," B.S. thesis, Universidad Tecnológica de Pereira, 8 2012.

[26] I. R. Medina, "Revisión de los algoritmos bio inspirados," *Universidad de Manchester*, pp. 1–31, 2014.

[27] R. l. Perez and K. Behdinan, "Particle swarm approach for structural design optimization," *Computers & Structures*, vol. 85, no. 19-20, pp. 1579–1588, 2007.

[28] İ. Durgun and A. R. Yildiz, "Structural design optimization of vehicle components using cuckoo search algorithm," *Materials Testing*, vol. 54, no. 3, pp. 185–188, 2012.

[29] A. R. Yildiz, "A new hybrid particle swarm optimization approach for structural design optimization in the automotive industry," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 226, no. 10, pp. 1340–1351, 2012.

[30] Z. W. Geem, K. S. Lee, and C.-L. Tseng, "Harmony search for structural design," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pp. 651–652, 2005.

[31] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[32] J. Kennedy, "Swarm intelligence," in *Handbook of nature-inspired and innovative computing*, pp. 187–219, Springer, 2006.

[33] J. H. Holland *et al.*, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[34] A. E. Eiben and J. E. Smith, "Genetic algorithms," in *Introduction to evolutionary computing*, pp. 37–69, Springer, 2003.

[35] D. Karaboğa and S. Ökdem, "A simple and global optimization algorithm for engineering problems: differential evolution algorithm," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 12, no. 1, pp. 53–60, 2004.

[36] M. Clerc, *Particle swarm optimization*, vol. 93. John Wiley & Sons, 2010.

[37] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in engineering software*, vol. 69, pp. 46–61, 2014.

[38] L. Abualigah, D. Yousri, M. Abd Elaziz, A. A. Ewees, M. A. Al-qaness, and A. H. Gandomi, "Aquila optimizer: A novel meta-heuristic optimization algorithm," *Computers & Industrial Engineering*, vol. 157, p. 107250, 2021.

[39] Z. W. Geem, J. H. Kim, and G. V. Loganathan, "A new heuristic optimization algorithm: harmony search," *simulation*, vol. 76, no. 2, pp. 60–68, 2001.

[40] L. Abualigah, A. Diabat, S. Mirjalili, M. Abd Elaziz, and A. H. Gandomi, "The arithmetic optimization algorithm," *Computer methods in applied mechanics and engineering*, vol. 376, p. 113609, 2021.

[41] O. Castillo, F. Valdez, J. Soria, L. Amador-Angulo, P. Ochoa, and C. Peraza, "Comparative study in fuzzy controller optimization using bee colony, differential evolution, and harmony search algorithms," *Algorithms*, vol. 12, no. 1, p. 9, 2019.

[42] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[43] S. Muelas, J. Pena, A. LaTorre, and V. Robles, "Algoritmos distribuidos heterogéneos para problemas de optimización continua," in *VI Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB*, pp. 425–432, 2009.

[44] J. A. Balcucho Mogollón and O. E. Gualdrón Guerrero, "Optimización de la base de reglas de un controlador difuso, mediante técnicas estocásticas como algoritmos genéticos y el simulated annealing," *Universidad de Pamplona, Revista Colombiana de Tecnologías de Avanzada, Pamplona, Norte de Santander, Colombia*, 2010.

[45] R. Martinez-Soto, O. Castillo, L. T. Aguilar, and I. S. Baruch, "Bio-inspired optimization of fuzzy logic controllers for autonomous mobile robots," in *2012 annual meeting of the north american fuzzy information processing society (NAFIPS)*, pp. 1–6, IEEE, 2012.

[46] E. Hernández, O. Castillo, and J. Soria, "Optimization of fuzzy controllers for autonomous mobile robots using the grey wolf optimizer," in *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 1–6, IEEE, 2019.

[47] M. L. Lagunes, O. Castillo, and J. Soria, "Optimization of membership function parameters for fuzzy controllers of an autonomous mobile robot using the firefly algorithm," *Fuzzy logic augmentation of neural and optimization algorithms: theoretical aspects and real applications*, pp. 199–206, 2018.

[48] A. V. Arosteguí Arias *et al.*, "Optimización de un esquema de control difuso mediante algoritmos evolutivos (artificial bee colony)," B.S. thesis, Quito, 2019.

[49] O. Castillo, "Dynamic parameter adaptation based on using interval type-2 fuzzy logic in bio-inspired optimization methods," in *International Conference on Innovations in Bio-Inspired Computing and Applications*, pp. 1–12, Springer, 2018.

[50] E. Bernal, O. Castillo, J. Soria, and F. Valdez, "Fuzzy galactic swarm optimization with dynamic adjustment of parameters based on fuzzy logic," *SN Computer Science*, vol. 1, no. 1, p. 59, 2020.

[51] C. Peraza, F. Valdez, and O. Castillo, "Harmony search with dynamic adaptation of parameters for the optimization of a benchmark set of functions," in *Hybrid Intelligent Systems in Control, Pattern Recognition and Medicine*, pp. 97–108, Springer, 2020.

[52] A. Mancilla, O. Castillo, and M. G. Valdez, "Evolutionary approach to the optimal design of fuzzy controllers for trajectory tracking," in *Intelligent and Fuzzy Techniques for Emerging Conditions and Digital Transformation* (C. Kahra-

man, S. Cebi, S. Cevik Onar, B. Oztaysi, A. C. Tolga, and I. U. Sari, eds.), (Cham), pp. 461–468, Springer International Publishing, 2022.

[53] J. C. Cortes-Rios, E. Gómez-Ramírez, H. A. Ortiz-de-la Vega, O. Castillo, and P. Melin, "Optimal design of interval type 2 fuzzy controllers based on a simple tuning algorithm," *Applied Soft Computing*, vol. 23, pp. 270–285, 2014.

[54] S.-K. Oh, H.-J. Jang, and W. Pedrycz, "The design of a fuzzy cascade controller for ball and beam system: A study in optimization with the use of parallel genetic algorithms," *Engineering Applications of Artificial Intelligence*, vol. 22, no. 2, pp. 261–271, 2009.

[55] S. Ciurea, "Determining the parameters of a sugeno fuzzy controller using a parallel genetic algorithm," in *2013 19th International Conference on Control Systems and Computer Science*, pp. 36–43, IEEE, 2013.

[56] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE transactions on evolutionary computation*, vol. 6, no. 5, pp. 443–462, 2002.

[57] Y. Li and X. Zeng, "Multi-population co-genetic algorithm with double chain-like agents structure for parallel global numerical optimization," *Applied intelligence*, vol. 32, no. 3, pp. 292–310, 2010.

[58] T. Starkweather, D. Whitley, and K. Mathias, "Optimization using distributed genetic algorithms," in *International Conference on Parallel Problem Solving from Nature*, pp. 176–185, Springer, 1990.

[59] H. Ma, S. Shen, M. Yu, Z. Yang, M. Fei, and H. Zhou, "Multi-population techniques in nature inspired optimization algorithms: A comprehensive survey," *Swarm and evolutionary computation*, vol. 44, pp. 365–387, 2019.

[60] M. García-Valdez and J. J. Merelo, "Event-driven multi-algorithm optimization: Mixing swarm and evolutionary strategies," in *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pp. 747–762, Springer, 2021.

[61] A. Mancilla, O. Castillo, and M. G. Valdez, "Mixing population-based metaheuristics: An approach based on a distributed-queue for the optimal design of fuzzy controllers," in *International Conference on Intelligent and Fuzzy Systems*, pp. 839–846, Springer, 2022.

[62] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows.," *Elpub*, vol. 2016, pp. 87–90, 2016.

[63] T. Carneiro, R. V. M. Da Nóbrega, T. Nepomuceno, G.-B. Bian, V. H. C. De Albuquerque, and P. P. Reboucas Filho, "Performance analysis of google colaboratory as a tool for accelerating deep learning applications," *IEEE Access*, vol. 6, pp. 61677–61685, 2018.

[64] J. Thönes, "Microservices," *IEEE Software*, vol. 32, pp. 116–116, Jan 2015.

[65] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[66] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502 – 514, 2020.

[67] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016*, pp. 287–288, 2016.

[68] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "The reactive manifesto," 2014.

[69] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, "Microservices: architecture, container, and challenges," in *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)*, pp. 629–635, IEEE, 2020.

[70] J. Gilbert, *Cloud Native Development Patterns and Best Practices: Practical architectural patterns for building modern, distributed cloud-native systems*. Packt Publishing Ltd, 2018.

[71] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing–a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[72] C. Vecchiola, M. Kirley, and R. Buyya, "Multi-objective problem solving with offspring on enterprise clouds," *arXiv preprint arXiv:0903.1386*, 2009.

[73] D. Sherry, K. Veeramachaneni, J. McDermott, and U. M. O'Reilly, "Flex-GP: Genetic programming on the cloud," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7248 LNCS, pp. 477–486, 2012.

[74] R. M. Valenzuela and M. G. Valdez, "Implementing pool-based evolutionary algorithm in Amazon Cloud Computing Services," in *Design of Intelligent Systems Based on Fuzzy Logic, Neural Networks and Nature-Inspired Optimization*, pp. 347–355, Springer, 2015.

[75] P. Salza and F. Ferrucci, "An approach for parallel genetic algorithms in the cloud using software containers," *arXiv preprint arXiv:1606.06961*, 2016.

[76] P. Salza, F. Ferrucci, and F. Sarro, "Develop, deploy and execute parallel genetic algorithms in the cloud," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pp. 121–122, 2016.

[77] A. De Lucia and P. Salza, *Parallel Genetic Algorithms in the Cloud*. PhD thesis, University of Salerno, 2017.

[78] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki, and L. S. Indrusiak, "Scalable distributed evolutionary algorithm orchestration using docker containers," *Journal of Computational Science*, vol. 40, p. 101069, 2020.

[79] J. J. Merelo-Guervos, A. Mora, J. Cruz, A. Esparcia-Alcázar, and C. Cotta, "Scaling in distributed evolutionary algorithms with persistent population," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1 –8, june 2012.

[80] D. Pamucar and G. Ćirović, "Vehicle route selection with an adaptive neuro fuzzy inference system in uncertainty conditions," *Decision Making: Applications in Management and Engineering*, vol. 1, no. 1, pp. 13–37, 2018.

[81] A. De Luca, G. Oriolo, and C. Samson, "Feedback control of a nonholonomic car-like robot," in *Robot motion planning and control*, pp. 171–253, Springer, 1998.

[82] C. Samson, "Path following and time-varying feedback stabilization of a wheeled mobile robot," in *International Conference on Control, Automation, Robotics and Vision, 1992*, 1992.

[83] K. Zhang, J.-X. Guo, and X.-S. Gao, "Cubic spline trajectory generation with axis jerk and tracking error constraints," *International Journal of Precision Engineering and Manufacturing*, vol. 14, no. 7, pp. 1141–1146, 2013.

[84] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques, "PythonRobotics: a Python code collection of robotics algorithms," *CoRR*, vol. abs/1808.10703, 2018. _eprint: 1808.10703.

[85] C. Li, T. T. Nguyen, M. Yang, S. Yang, and S. Zeng, "Multi-population methods in unconstrained continuous dynamic environments: The challenges," *Information Sciences*, vol. 296, pp. 95–118, 2015.

[86] X.-S. Yang, Z. Cui, R. Xiao, A. H. Gandomi, and M. Karamanoglu, *Swarm intelligence and bio-inspired computation: theory and applications*. Newnes, 2013.

[87] Y. Gong and A. Fukunaga, "Distributed island-model genetic algorithms using heterogeneous parameter settings," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, pp. 820–827, IEEE, 2011.

[88] S.-A.-A. Touati, J. Worms, and S. Briais, "The speedup-test: a statistical methodology for programme speedup analysis and computation," *Concurrency and computation: practice and experience*, vol. 25, no. 10, pp. 1410–1426, 2013.

[89] Y. Vargas, S. Chen, and J. Otero, "Estrategias para mejorar el balance entre exploración y explotación en optimización de enjambre de partículas," *Universidad de la Habana, Facultad de Matemticas y Computacin, Diciembre de*, 2011.

[90] F. Olivas, F. Valdez, O. Castillo, and P. Melin, "Dynamic parameter adaptation in particle swarm optimization using interval type-2 fuzzy logic," *Soft Computing*, vol. 20, pp. 1057–1070, 2016.

[91] H. Wang, H. Sun, C. Li, S. Rahnamayan, and J.-s. Pan, "Diversity enhanced particle swarm optimization with neighborhood search," *Information Sciences*, vol. 223, pp. 119–135, 2013.

[92] S. Cheng and Y. Shi, "Diversity control in particle swarm optimization," in *2011 IEEE symposium on swarm intelligence*, pp. 1–9, IEEE, 2011.