National Technical University of Athens
School of Mechanical Engineering
Parallel CFD & Optimization Unit
Joint Postgraduate Program
«Computational Mechanics»

PROGRAMMING AND IMPLEMENTATION OF THE CHICKEN SWARM OPTIMIZATION (CSO) ALGORITHM IN OPTIMIZATION PROBLEMS

Master's Thesis

Nikolaos Fykatas

Supervisor: Kyriakos C. Giannakoglou, Professor NTUA

Athens, June 2025

Acknowledgments

With this master's thesis, I complete my studies in the postgraduate program ≪Computational Mechanics≫ at the National Technical University of Athens (NTUA). For this reason, I would like to extend my gratitude to the people who contributed in various ways to my successful completion of both this thesis and my postgraduate studies as a whole.

First of all, I would like to thank my thesis supervisor, Professor Kyriakos Giannakoglou of NTUA, for assigning me this exceptionally interesting topic. His guidance was crucial in my decision to accept and successfully complete this thesis. I must also acknowledge the knowledge he shared with me, not only during the period of this thesis but also throughout my studies in the three courses I had the privilege to take under his instruction.

I would also like to thank Dr. Varvara Asouti for her time and guidance, both on a theoretical level and in terms of code and algorithms. Her clarification of my questions and provision of educational material were invaluable.

Finally, I must express my gratitude to my parents, Athanasios and Maria, for their continuous emotional support throughout my studies. I am also deeply thankful to my friends, classmates, and certain colleagues who helped ease my daily routine during an exceptionally demanding period. I thank them all warmly.

Athens, June 2025

Fykatas Nikolaos

National Technical University of Athens
School of Mechanical Engineering
Parallel CFD & Optimization Unit
Joint Postgraduate Program
«Computational Mechanics»

PROGRAMMING AND IMPLEMENTATION OF THE CHICKEN SWARM OPTIMIZATION (CSO) ALGORITHM IN OPTIMIZATION PROBLEMS

Master's Thesis

Nikolaos Fykatas
Supervisor: Kyriakos H. Giannakoglou, Professor NTUA

Abstract

The aim of this Master's thesis is to develop and assess an optimization software based on the cooperative behavior of individual members of the three subgroups (roosters, hens, and chicks) within a chicken swarm as they search for food. The performance of the Chicken Swarm Optimization (CSO) algorithm is compared with Evolutionary Algorithm (EA) and Metamodel-Assisted Evolutionary Algortihm (MAEA), which are implemeted through the EASY software, developed by the Parallel CFD & Optimization Unit of the School of Mechanical Engineering of NTUA. In the future, an overarching objective is to potentially integrate promising elements of the proposed algorithm into EASY, to further improve its performance.

As part of the Swarm Intelligence family of algorithms, the CSO technique manages populations of chickens that move through the design space in search of optimal solutions. This thesis investigates the interactions among the three aforementioned subpopulations that comprise the total swarm, the parameter settings related to their rearrangement, and the overall performance of CSO compared to EA and MAEA. This algorithm effectively handles both Single-Objective (SOO) and Multi-Objective Optimization (MOO) problems. New features have also been introduced by the author beyond what is described in the literature, aimed at further improving the algorithm's

performance. These new features concern the movement of the roosters and the handling of penalized individuals in both SOO and MOO problems with constraints.

Specifically, the shortcomings of the original algorithm—particularly in relation to the movement of roosters and chicks—were effectively addressed through the introduction of a stochastic mechanism that aids in avoiding local optima. Furthermore, the capability to solve MOO problems was developed/programmed using the SPEA-II method. Constraint handling in MOO problems was strengthened through a mechanism that enhances the foraging ability of non-penalized individuals, potentially leading to a greater number of solutions. Lastly, a "revival" mechanism was implemented to restore individuals previously penalized with elimination, in a manner that improves the algorithm's performance rather than merely replacing them without substantial enhancement.

The algorithm is developed and validated on well-known mathematical benchmark problems, and further tested on two pseudo-engineering problems: the Vibrating Platform Design and the Two-Bar Truss Design problems. Although CSO is capable of solving CFD problems as well, these are not presented in this thesis due to the different background of the author. Overall, the results are very promising, suggesting that CSO is a powerful optimization algorithm capable of being applied to a variety of problems. However, further tuning and investigation are necessary for it to consistently reach the performance levels currently exhibited by EASY, particularly in more complex optimization cases.

Last but not least, it is important to highlight the significant role of Metamodel-Assisted techniques in modern stochastic optimization algorithms. The EASY framework incorporates this which is utilized in this thesis. In contrast, the CSO algorithm does not currently support metamodel assistance. Introducing such a mechanism into CSO could represent a promising direction for future work, potentially enhancing its performance.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Μηχανολόγων Μηχανικών
Μονάδα Παράλληλης Υπολογιστικής Ρευστοδυναμικής & Βελτιστοποίησης
Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών «Υπολογιστική Μηχανική»

ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΚΑΙ ΧΡΗΣΗ ΑΛΓΟΡΙΘΜΟΥ ΣΜΗΝΟΥΣ ΚΟΤΟΠΟΥΛΩΝ (Chicken Swarm Optimization- CSO) ΣΕ ΠΡΟΒΛΗΜΑΤΑ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ

Μεταπτυχιακή Εργασία

Φύκατας Νικόλαος

Επιβλέπων Καθηγητής: Κυριάκος Χ. Γιαννάκογλου, Καθηγητής ΕΜΠ

Περίληψη

Σκοπός της μεταπτυχιακής αυτής εργασίας είναι η ανάπτυξη λογισμικού για την εφαρμογή μιας μεθόδου βελτιστοποίησης με απώτερο σκοπό να βρει εφαρμογή σε προβλήματα μηχανικού. Ο αλγόριθμος βασίζεται στο πώς ένα σμήνος κοτόπουλων, αποτελούμενο από τρεις υπο-πληθυσμούς (κόκορες, κότες, κοτοπουλάκια), συνεργάζονται στην αναζήτηση τροφής, πραγματοποιώντας, με αυτόν τον τρόπο, βελτιστοποίηση στα προβλήματα αυτά. Τα αποτελέσματα που προέκυψαν, συγκρίνονται με τον Εξελικτικό Αλγόριθμο (ΕΑ) και τον αλγόριθμο Βοηθούμενης Βελτιστοποίησης μέσω Μεταμοντέλων(ΜΑΕΑ), που είναι υλοποιημένοι στο λογισμικό EASY της Μονάδας Παράλληλης Υπολογιστικής Ρευστομηχανικής & Βελτιστοποίησης του Ε.Μ.Π., με απώτερο στόχο την ενσωμάτωση υποσχόμενων στοιχείων του πρώτου στο δεύτερο, στο εγγύς μέλλον.

Στο πλαίσιο των αλγορίθμων Swarm Intelligence, ο αλγόριθμος Chicken Swarm Optimization (CSO) διαχειρίζεται πληθυσμούς κοτόπουλων που κινούνται στον χώρο σχεδιασμού αναζητώντας βέλτιστες λύσεις. Η μεταπτυχιακή εργασία εξετάζει τις αλληλεπιδράσεις μεταξύ των τριών προαναφερθέντων υπο-πληθυσμών που απαρτίζουν το

συνολικό σμήνος, τις παραμέτρους που σχετίζονται με την αναδιάταξή τους, καθώς και τη συνολική απόδοση του CSO σε σύγκριση με τους αλγόριθμους EA και MAEA. Επιπλέον, ο αλγόριθμος έχει τη δυνατότητα να χειρίζεται αποτελεσματικά τόσο προβλήματα βελτιστοποίησης ενός Single Objective Optimization - SOO) όσο και πολλών Multi-Objective Optimization - MOO στόχων. Επίσης, έγιναν μερικές σημαντικές προσθήκες στον αλγόριθμο από τον συγγραφέα, πέραν όσων αναφέρονται στη βιβλιογραφία, με στόχο την περαιτέρω βελτίωση της απόδοσής του. Αυτές αφορούν την κίνηση των κοκόρων καθώς και τη διαχείριση των ατόμων που τιμωρούνται λόγω παραβίασης περιορισμών σε προβλήματα με περιορισμούς είτε ενός είτε πολλών στόχων.

Συγκεκριμένα, τα μειονεκτήματα του αρχικού αλγορίθμου—ιδίως σε ό,τι αφορά την κίνηση των κοκόρων και των μικρών κοτόπουλων—αντιμετωπίστηκαν επιτυχώς μέσω της εισαγωγής ενός στοχαστικού μηχανισμού, που βοηθά στην αποφυγή τοπικών ακροτάτων. Επιπλέον, η δυνατότητα επίλυσης προβλημάτων βελτιστοποίησης πολλών στόχων αναπτύχθηκε με τη χρήση της μεθόδου Strength Pareto Evolutionary Algorithm II (SPEA-II). Η διαχείριση περιορισμών σε προβλημάτων πολλών στόχων ενισχύθηκε μέσω ενός μηχανισμού που βελτιώνει την ικανότητα αναζήτησης τροφής των μη τιμωρημένων ατόμων, οδηγώντας ενδεχομένως σε περισσότερες λύσεις. Τέλος, εφαρμόστηκε ένας μηχανισμός «αναβίωσης» για την επαναφορά ατόμων που είχαν υποστεί ποινή θανάτου, με τρόπο που ενισχύει την απόδοση του αλγορίθμου, αντί να την επιβαρύνει ή να περιορίζεται απλώς στην αντικατάστασή τους χωρίς ουσιαστική βελτίωση.

Ο αλγόριθμος αναπτύχθηκε και δοκιμάστηκε σε γνωστά μαθηματικά προβλήματα αναφοράς και, στη συνέχεια, εφαρμόστηκε σε δύο προβλήματα ψευδο-μηχανικής: τον Σχεδιασμό Πλατφόρμας Δόνησης (Vibrating Platform Design) και τον Σχεδιασμό Δικτυώματος Δύο Ράβδων (Two-Bar Truss Design). Αν και ο CSO μπορεί να εφαρμοστεί και σε προβλήματα Υπολογιστικής Ρευστοδυναμικής (CFD), τα συγκεκριμένα δεν παρουσιάζονται στην εργασία λόγω του διαφορετικού γνωστικού αντικειμένου του συγγραφέα. Συνολικά, τα αποτελέσματα είναι ιδιαίτερα ενθαρρυντικά, υποδεικνύοντας ότι ο CSO αποτελεί έναν ισχυρό αλγόριθμο βελτιστοποίησης με δυνατότητα εφαρμογής σε ευρύ φάσμα προβλημάτων. Ωστόσο, απαιτείται περαιτέρω ρύθμιση και μελέτη για να φτάσει σε επίπεδα απόδοσης αντίστοιχα με αυτά του EASY, ειδικά σε πιο σύνθετα προβλήματα βελτιστοποίησης.

Τέλος, αλλά εξίσου σημαντικό, είναι ουσιώδες να υπογραμμιστεί ο καθοριστικός ρόλος των τεχνικών Βοηθούμενης Βελτιστοποίησης μέσω Μεταμοντέλων (Metamodel-Assisted) στους σύγχρονους στοχαστικούς αλγορίθμους βελτιστοποίησης. Το λογισμικό EASY ενσωματώνει αυτή τη δυνατότητα, η οποία χρησιμοποιείται και στην παρούσα εργασία. Αντιθέτως, ο CSO δεν υποστηρίζει επί του παρόντος τη χρήση μεταμοντέλου. Η ενσωμάτωση ενός τέτοιου μηχανισμού στον CSO θα μπορούσε να αποτελέσει μια πολλά υποσχόμενη κατεύθυνση για μελλοντική έρευνα, ενισχύοντας ενδεχομένως την απόδοσή του.

Acronyms

SOO : Single-Objective Optimization
MOO : Multi-Objective Optimization
EA : Evolutionary Algorithms
GA : Genetic Algorithms
ES : Evolutionary Strategies
CSO : Chicken Swarm Optimization
DEA : Distributed Evolutionary Algorithm
CFD : Computational Fluid Dynamics
EASY : Evolutionary Algorithm SYstem
SPEA : Strength Pareto Evolutionary Algorithm
GEA : Generalized Evolutionary Algorithm
PCOpt/NTUA : Parallel CFD & Optimization Unit of the School of Mechanical
: Engineering, National Technical University of Athens
MAEA : Metamodel Assisted Evolutionary Algorithm
LCPE : Low-Cost Pre-Evaluation
HVI : Hyper-Volume Indicator

Contents

Chapter 1

Introduction

1.1 Introduction to Optimization

The term *optimization* refers to the process of seeking the most suitable solution or set of solutions to any problem, the performance of which is described by one or more mathematical functions. These mathematical functions are known as *objective functions* or *target functions* or *design functions* [1]. The optimal solution is defined as the value-set of the *design variables* in the *design* space where these variables are defined. Furthermore, the solutions that emerge must respect the *constraints* imposed, if any.

Problems aiming to optimize a single objective function (Single Objective Optimization - SOO) [2] seek the global extremum of this function. If this is the minimum, then we have a minimization problem; if it is the maximum, then it is a maximization problem. However, real-world problems are much more complex than that. Constraints make them more challenging. Additionally, there might be many local optima where the solution could get "trapped".

The challenges multiply if solving the problem requires satisfying many objective functions (Multi-Objective Optimization - MOO) [3] [4]. With MOO methods, the so-called Pareto front is sought. Among the members of the Pareto front, one objec-

tive requires sacrificing performance in another, necessitating a careful balance and compromise. There is a whole scientific field, called decision-making, that guides users in selecting the most appropriate solutions among those found. However, this is beyond the scope of this thesis.

Various methods exist to solve optimization problems. These are divided into two categories: stochastic and deterministic (or gradient-based).

1.2 Deterministic Optimization Methods

Deterministic optimization methods (or gradient-based methods) rely on the generalized concept of the gradient of the objective function, with their primary goal being to calculate or at least approximate it. Computing the gradient is not a trivial task. Several methods have been introduced and developed for this purpose, including the Finite Differences Method [5], [6], the Complex Variables Method [7], [8], the Direct Differentiation (continuous and discrete) [9], and the Adjoint Variables (continuous and discrete) methods [2].

Apart from precision, one of the most critical considerations for an engineer is the cost to compute the gradient of the objective function. In real-world problems, the number of design variables, denoted as N, is typically much larger than the number of objectives, M. Most optimization methods computing the gradient have a computational cost that scales proportionally with N. However, the Adjoint Variables Method stands out due to the fact that its cost does not scale with N [2]. This unique characteristic makes it particularly advantageous and places it in a distinguished position among other optimization techniques.

The Deterministic algorithms are highly dependent on initialization. Since the solution is randomly initialized, it may fall into local optima. Furthermore, the implementation of deterministic algorithms often requires greater development time compared to stochastic algorithms. Extending and generalizing these methods to solve similar problems can be particularly challenging.

1.3 Stochastic Optimization Methods

In contrast to deterministic methods, stochastic methods do not use the gradient of objective functions. They adapt easily to various problems for which they are designed and are capable of locating the global extremum in each case. In other words, they are independent of initialization. However, they have higher runtime cost compared to deterministic methods. The implementation of stochastic algorithms involves the use of a separate software to calculate objective functions and evaluate candidate solutions.

There is a plethora of stochastic algorithms well described in the literature, such as Genetic Algorithms (GA)[10], [11], [12], Evolutionary Strategies (ES) [13], [14], [15], [16], [17], [18], Genetic Programming (GP)[19], [20], [21], [22], [23], and other bio-inspired algorithms that simulate the behaviors of natural systems, animals, insects, and more. Particle Swarm Optimization (PSO) [24] is a well-known representative of the latter category. A derivative of PSO is the Chicken Swarm Optimization (CSO) algorithm [25], which is programmed, used and assessed in this thesis.

1.4 Basic Concepts in Optimization

To provide a foundation for understanding stohastic optimization, it is essential to first discuss a basic optimization problem. This approach will help illustrate the fundamental characteristics of optimization and shed light on the challenges that may arise during the process. By analyzing a simple problem, the reader can gain a clearer understanding of the core principles and intricacies involved in optimization. Evolutionary optimization employs nature-inspired strategies such as mutation, crossover, and elitism to iteratively refine solutions. Given the vast scope of stochastic methods, this thesis focuses on providing a concise introduction to the field, outlining its key principles while acknowledging that it cannot encompass all topics within the domain.

In out notation, objective functions are denoted by f_i, while inequality constraint functions by g_i must be less than or equal to zero ($g_i \leq 0$) and equality constraint functions by h_i. However, alternative notations may also be used depending on the

context. The variables upon which both the objective functions and constraints depend are referred to as *design variables*, denoted by x_i. The collection of these design variables forms a vector, known as the *design vector*. Each design variable is confined by its own bounds within the $\mathbb{R}^N$ space, collectively referred to as the *design space*. This design space defines the region where optimization occurs. So, the aim in an optimization problem is to find the design vector:

$$\vec{X}^* = \begin{Bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_{N-1}^* \\ x_N^* \end{Bmatrix} \tag{1.1}$$

which minimizes or maximizes the objective function in SOO problems, or provides a compromise set of solutions in MOO problems. Additionally, the solution must respect all constraints, if any.

As mentioned before, the design variables are bounded. Therefore, there is a need to define a minimum and a maximum value for every design variable, i.e.:

$$x_i^{\min} \leq x_i \leq x_i^{\max}, i = 1, \ldots, N \tag{1.2}$$

In SOO, the problem is typically defined by a function $f(\vec{X})$ that needs to be maximized or minimized. The optimization process involves exploring the search space to identify the solution $\vec{X}^*$ that maximizes or minimizes $f(\vec{X})$ while adhering to the specified constraints.

While the problem may appear straightforward, it's quite challenging since there is an, theoritically, infinity number of candidate solutions to be evaluated. Moreover, challenges emerge when the objective function is non-linear, has many local minima or maxima, or is subject to complex, often nonlinear constraints. In such cases, finding an optimal solution can be computationally demanding.

A simple approach to solving such optimization problems is to start with a random initial solution $\vec{X}$, evaluate its objective function, and then iteratively make small

changes to it. After each modification, the solution is reevaluated, and if the new solution improves the objective function, is kept. This process is repeated until the changes in the objective function value become negligible over successive iterations, indicating that either further improvements are minimal or a local optimum has been reached.

A second approach involves taking a set of design vectors $\vec{X_i}$ (i.e a population), and evaluating all of them. The best solution, $\vec{X_i^*}$, is then kept and so forth.

These two approaches correspond to different classifications of stochastic algorithms: the individual-based methods and the population-based methods, respectively. A basic population-based evolutionary algorithm (EA), with terminology borrowed from the (μ, λ) EA /EASY framework, consists of λ offspring and μ parents, where usually $\mu \leq \lambda$. The λ offspring are evaluated and based on a selection procedure, μ parents are chosen. Then, using a procedure different from the one before, the selected parents generate λ new offspring.

1.5 Introduction to Swarm Intelligence

The CSO algorithm is thoroughly discussed in Chapter 3. However, as CSO belongs to the family of swarm intelligence algorithms, it is essential to first get familiar with the ideas behind swarm intelligence algorithms, in order to better understand the CSO when it is introduced.

Intelligent meta-heuristic algorithms have the capability to learn and provide effective solutions to highly complex problems. Within this category, swarm intelligence computing is gaining prominence, as these algorithms mimic the adaptive and learning behavior of biological organisms. Their appeal lies in their ability to address increasingly complex problems, navigate vast multi-dimensional solution spaces, adapt to the dynamic nature of constraints, and manage incomplete, probabilistic, or imperfect information in decision-making processes.

However, the rapid advancements in this field have made it increasingly difficult for researchers to stay up to date, as new algorithms are introduced at an accelerated

pace. Over the years, numerous swarm intelligence algorithms have been developed, including particle swarm optimization (PSO) [24] [26], ant colony optimization (ACO) [27], [28], [29], artificial bee colony optimization (ABCO) [30], [31], bacterial foraging optimization (BFO) [32], [33], [34], firefly optimization (FO) [35], [36], [37], leaping frog optimization (LFO) [38], [39], bat optimization algorithms (BOA) [40], [41], and CSO [25], [42], among others. To build a stronger foundation in swarm intelligence algorithms and set the stage for understanding CSO, the fundamental concepts of PSO is introduced next.

1.6 Particle Swarm Optimization

PSO[24] [26] holds a prominent position among swarming theory and bio-inspired algorithms. It serves as a broader category that encompasses the CSO algorithm. For this reason, it is crucial to introduce and briefly present PSO before delving into the specifics of CSO. The PSO [24] [26] algorithm is inspired by the collective behavior of swarm animals, such as fish schools or bird flocks, among others. Fundamentally, the primary objective of the PSO algorithm is to search for parameters that optimize a given objective function.

In PSO, each particle represents a potential solution within the search space. Each particle in the swarm has a position, velocity, and fitness value. The position represents a potential solution, the velocity indicates the direction and speed of movement, and the fitness value measures how good the solution is. Particles keep track of their personal best positions and the global best position, which guide the swarm's movement towards optimal solutions. The algorithm updates each particle's velocity and position based on a combination of its current velocity, the distance to its personal best, so-far, position, and the distance to the global best, so-far, position.

1.7 Chicken Biology

To better understand the CSO algorithm, it's helpful to first learn about real chickens and how their flocks behave. According to [25], which provides comprehensive information about chicken biology, chickens are among the most common domesticated animals worldwide. These social creatures typically live in groups and display a surprisingly advanced level of intelligence. They can recognize and remember over a hundred individual chickens, even after being apart for long periods. Their vocal communication is rich, consisting of more than 30 distinct sounds such as chirps, crows, and squawks, used to convey messages about nesting, food, mating, and threats. Chickens don't rely solely on trial and error to learn; they also draw on past experiences and observe the behavior of others to guide their actions.

A structured social ranking, often referred to as a pecking order, shapes their interactions. More assertive birds hold dominant positions, staying close to the leading roosters, while less dominant individuals tend to keep to the edges of the group. Any changes to the group, like adding or removing members, temporarily disrupt this structure until a new balance is formed.

High-ranking chickens generally get first access to food. Sometimes, a rooster that discovers food will even call over the rest of the group before eating. This kind of considerate behavior is also seen in hens as they care for their young. However, such cooperation is mostly limited to members of the same group. When unfamiliar chickens enter the area, roosters often respond with loud vocalizations to defend their territory.

Behavioral patterns also differ between roosters and hens. The leading rooster actively seeks out food and defends the group's area. Dominant hens follow roosters while foraging and chicks typically feed near their mothers.

While individual chickens may appear simple, collectively they exhibit a kind of coordinated behavior. Working together, they follow their internal social structure to locate food efficiently. This group-level cooperation, or "swarm intelligence," can be viewed as a natural problem-solving process, one that has inspired the creation of new optimization algorithms. This fascinating behavior inspired scientists and engineers

to develop a novel bio-inspired algorithm: the CSO algorithm.

1.8 Real-World Optimization Problems

The concepts discussed in the previous sections lay the groundwork for solving basic SOO problems. However, real-world challenges are often far more complex, requiring modifications to the foundational stohastic algorithms introduced earlier in the chapter. Engineers frequently face problems involving constraints, and many of these problems also require addressing multiple objectives. It is important to note that the ideas presented in this section apply not only to the previously discussed algorithms but also to other stochastic and bio-inspired algorithms. Sorting operation and comparison make use of penalized fitness values, rather than the raw fitness values themselves.

1.9 Goal of Thesis

The CSO algorithm has been gaining significant interest over the last decade since its introduction. Numerous researchers have tested and applied the algorithm in various fields, such as data mining, robotics, and more [42]. The promising results have intrigued the Parallel CFD & Optimization Unit of NTUA to explore and design enhancements for the algorithm.

The team has previously developed the (μ, λ) EA (as in the EASY software) [2], that merges characteristics from GA and ES. The primary focus of this thesis is to present the characteristics of CSO, a population-based stochastic algorithm, the relationship between the three populations it comprises, the comparison with (μ, λ) EA, its integration of ideas drawn from EASY and the potential application of CSO concepts to EASY. The contribution of this work aspires to bridge the gap between a swarm intelligence algorithm and the (μ, λ) EA.

1.10 Structure of the Thesis

This thesis is structured into the following chapters:

- **Chapter 2 - Evolutionary Optimization Algorithms:** This chapter provides an analysis of the basic principles of optimization, the (μ, λ) EA and an introduction to swarm intelligence algorithms and PSO.
- **Chapter 3 - The CSO Algorithm:** A detailed description of the CSO algorithm is presented, including its structure, operation, and application to optimization problems.
- **Chapter 4 - Applications - Discussion:** An analysis about the parameters of CSO is presented on how the parameters RN, MN, G affect the solution of the SOO problem of the shifted Rastrigin function. Two pseudo-engineering problems, the Vibrating Platfrom Design and the Two Bar Truss Design, are tested and are compared to EA and MAEA. This shows how well can the CSO perform, compared to a well-developed algorithm.
- **Chapter 5 - Conclusions and Future Work:** This chapter presents the conclusions drawn from the performed tests and suggests directions for future work in the field.

Chapter 2

EA-based Optimization

EAs represent a wide class of stochastic algorithms. The notable rise in the use of such algorithms followed the rapid increase in computational power. Consequently, the corresponding wall clock time was drastically reduced. In addition, they gained significant interest due to their non-mathematical foundation, their ease of adaptation to various problems, and their flexibility in comparison to deterministic algorithms. The only thing needed is an evaluation software to assess each individual. Their greatest advantage is that they do not fall in local optima. However, the time required to find the optimal solution can be quite large, as it demands a significant number of evaluations to reach it. The concept of EAs is not new; they were proposed in the 1960s [43] [44] [45]. Their foundation stems from Darwin's theory of evolution of species.

With the term "evolution", we describe the automatic process of adaptation to each system to the environment. The term "environment" includes all the external conditions that affect the system. EAs are computational models that, through a process analogous to the evolutionary adaptation of individuals in the environment, solve problems. For this purpose, they utilize stochastic mechanisms of evolution derived from nature, and they are based on principles of the evolution of species, which were initially developed by Darwin in the 1960s. EAs resemble, in a simplified

way, the evolution of natural populations. According to Darwin's theory, individuals in a population are engaged in the struggle for resources such as shelter, food, and survival. The successful individuals have a higher probability of reproducing and passing their traits to future generations. This natural selection means that the offspring of successful individuals will have better-adapted traits to the environment and, as a result, improve the population. The combination of these favorable traits from different successful individuals results in the population evolving and adapting to the environment.

A fundamental characteristic of EAs is that they are population-based methods (with a few exceptions, ES may have a population of one individual) and involve an iterative process, similar to other stochastic methods (e.g., CSO). While EAs were initially developed for solving single-objective problems, with appropriate modifications, they can, also, address MOO problems to provide the Pareto front of non-dominated solutions.

A special place among EAs is held by the GA[10], [11], [12], which is the most widely recognized EA. Two other prominent categories with extensive applications are ES [13], [14], [15], [16], [17], [18] and GP[19], [20], [21], [22], [23]. Below is a concise overview of the three main categories of EAs and their applications.

2.1 The Generalized EA - (μ, λ) EA

In this section, a Generalized EA is introduced, combines elements of GA and ES. This algorithm, also referred to as the (μ, λ) EA, was developed by the PCOpt /NTUA, as mentioned earlier. Further details can be found in [2]. However, since the developed software, EASY, is based on this algorithm and is used for comparison with the CSO algorithm presented in this thesis, the (μ, λ) algorithm is also briefly presented herein.

The (μ, λ) EA handles populations of solutions. This iterative process is repeated until a convergence criterion is satisfied. The convergence criteria can be one or any combination of the following:

- The best solution is no longer improving,

- The population becomes homogeneous,
- The computational time limit provided by the user is reached.

Before proceeding with the detailed steps of the algorithm, it is essential to introduce the three groups of individuals that the algorithm operates with. The first group is $G^{t,\mu}$, the parent group, which consists of μ individuals. The parents are selected by applying the parent selection operator and the offspring by applying the crossover, mutation and elitism operators. The second group is $G^{t,\lambda}$, the offspring group, which consists of λ individuals. Finally, the third group is $G^{t,e}$, the elite group, consisting of the elite individuals. The superscript t is the generation counter, indicating that these groups evolve over successive generations. The elite group holds the best solutions identified up to the current generation, ensuring that these superior solutions are preserved throughout the evolutionary process. Further details can be found in [2].

2.1.1 Parent Selection

One of the most critical parameters influencing the performance of the algorithm is the parent selection operator. Its objective is to exploit the best characteristics of high-quality candidate solutions, refining and improving them across generations. Ideally, this process guides the EA toward convergence on a satisfactory and acceptable solution for the optimization problem at hand.

The literature proposes several parent selection operators. Despite decades of research, no universal guidelines or theoretical frameworks exist to determine the most suitable parent selection operator for a given problem. This is a significant challenge, as an inadequate parent selection operator can result in premature convergence and inefficiency.

Six different Selection Operators (SOP) [46] are considered the most common ones: roulette wheel selection (RWS)[47], stochastic universal sampling (SUS)[48] [49], linear rank selection (LRS) [49], exponential rank selection (ERS) [49], tournament selection (TOS) [49] [50], and truncation selection (TRS) [49]. From this list of op-

erators, only TOS is presented here, as it is the one used for the simulations in this thesis.

2.2 Constrained Optimization

Optimization problems with physical significance include, usually, constraints. For instance, when designing a transportation system with minimal cost, the objective function could represent the total cost. However, the design must also satisfy various physical and safety regulations to ensure that it can handle the required loads without failure. Therefore, the strength limits must be considered as constraints and must not be exceeded under any circumstances. It is important to note that there is no need to impose constraints on the objective functions, such as a maximum allowable cost in this example, because the solutions are optimized based on these functions, making it unnecessary. A constraint may involve a subset of the design variables, or even all of them simultaneously. In a black-box approach the evaluation software must be able to provide the EA not only with the values of the objective functions $f_i(\vec{X})$, but also with the values of the constraints $g_i(\vec{X})$ for each candidate solution $\vec{X}$, since the solution must be tested in every aspect of the problem.

The infeasible individuals in the population that do not satisfy one or more of the constraints should never be ignored by the EA. If they are allowed to reproduce freely, the population will acquire unsuitable characteristics over time, and optimization will fail.

There are two categories of constraints: equality and inequality constraints. In this thesis, the analyzed problems involve inequality constraints of the form $g_i(\vec{X}) \leq 0$. Therefore, only this category is discussed in detail. If the constraint value is greater than the threshold but below the nominal value, the individual receives a "soft" penalty. Otherwise, it receives a "death" penalty. A penalty is calculated for each individual based on the constraints it does not satisfy. The total penalty is the sum or product of all penalties for each constraint the individual violates. In some cases, it may be more appropriate to use the product of penalties, depending on the developer's

choice. The penalized fitness is given by the following equations:

If all the constraints are respected,

$$f_p(\vec{X}_i) = f(\vec{X}_i) \tag{2.1}$$

else,

$$f_p(\vec{X}_i) = f(\vec{X}_i) + Penalty \tag{2.2}$$

where $f_p(\vec{X}_i)$ is the penalized fitness of the i^{th} individual and Penalty is given by

$$\text{Penalty} = \sum_{j=1}^{N_c} \text{penalty}_j \quad \text{or} \quad \text{Penalty} = \prod_{j=1}^{N_c} \text{penalty}_j \tag{2.3}$$

where nc is the number of constraints and penalty_j is an exponential function depending on the value of each constraint j, the nominal threshold C_j^{THR} (which is zero, since the the form of constraints is $g_i \leq 0$), above which an individual receives the "soft" penalty, and the relaxed penalty value C_j^{D}, above which an individual receives a "death" penalty. For every constraint, both values are selected by the user. In summary:

$$\text{If } g_j(\vec{X}) \leq C_j^{\text{THR}} \rightarrow \text{Penalty}_j = 0 \tag{2.4}$$

$$\text{If } C_j^{\text{THR}} < g_j(\vec{X}) < C_j^{\text{D}} \rightarrow \text{Penalty}_j = \exp\left[\frac{g_j(\vec{X}) - C_j^{\text{THR}}}{C_j^{\text{D}} - g_j(\vec{X})}\right] \tag{2.5}$$

$$\text{If } g_j(\vec{X}) \geq C_j^{\text{D}} \rightarrow \text{Penalty}_j = \text{"Death" Penalty} \tag{2.6}$$

Figure 2.1 shows how the penalty value increases as the constraint value approaches the C^{D} value. In some cases, researchers may assign a "death" penalty to a solution when the penalty becomes extremely high, even though the constraint value is still lower than the nominal value. This "death" penalty value could be something like 10^{20}, which is considered sufficiently large to discard a solution.

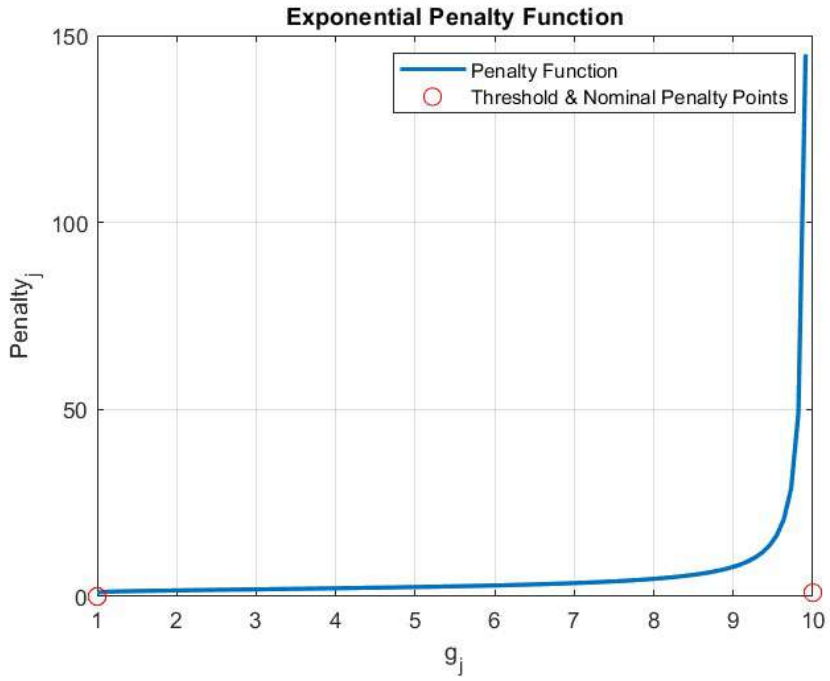

Figure 2.1: Penalty function between the threshold C_j^{THR} and the nominal penalty value C_j^{D}.

2.3 MOO Problems

Consider a MOO minimization problem with M objectives and N design variables, where there are M objective functions $f_1(\vec{X}), f_2(\vec{X}), \ldots, f_{M-1}(\vec{X}), f_M(\vec{X})$ that depend on the same n common design variables. The vector-valued objective function to be minimized, $\vec{f}(\vec{X})$, is defined as:

$$\vec{f}(\vec{X}) = \begin{bmatrix} f_1(\vec{X}) \\ f_2(\vec{X}) \\ \vdots \\ f_{M-1}(\vec{X}) \\ f_M(\vec{X}) \end{bmatrix} \tag{2.7}$$

The primary issue that arises based on the theory of EAs presented in Section 2.2 lies in the area of parent selection. All methods proposed in for SOO, ranked solutions,

from best to worst, use the value of $f(\vec{X})$ as a criterion. However, in MOO, there are now many equally important and almost always conflicting objective functions $f_i(\vec{X})$, which makes it far from obvious which offspring should be selected as parents.

In MOO, unlike SOO, there are many optimal solutions instead of just one. A vector $\vec{X}_P$ is referred to as a Pareto optimal solution if, in a minimization problem, no other vector $\vec{X}_R$ exists in the design space such that $f_i(\vec{X}_R) \leq f_i(\vec{X}_P)$ for every $i = 1, 2, \ldots, m$, and simultaneously $f_j(\vec{X}_R) < f_j(\vec{X}_P)$ for at least one objective j. Conversely, a vector $\vec{X}_D$ is said to be dominated by $\vec{X}_R$ if $f_i(\vec{X}_R) \leq f_i(\vec{X}_D)$ for every $i = 1, 2, \ldots, m$, and $f_j(\vec{X}_R) < f_j(\vec{X}_D)$ for at least one objective j. The set of all non-dominated solutions generated by the algorithm forms the *Pareto front*, which represents the trade-offs between the objectives. The Pareto front can be visualized effectively only in 2D or 3D.

Two members of the Pareto front cannot be compared to each other without involving extra criteria. In a two-objective problem, it is generally expected that one solution will perform better in one objective and worse in the other, compared to another solution that behaves inversely. For this reason, a family of optimization methods that has started to find significant applications (especially in optimization problems in aerodynamics) is those based on the concept of the Pareto front. These methods do not compute a single solution but instead a Pareto front.

The solution to a MOO problem does not conclude with the calculation of the Pareto front. Instead, it marks the beginning of a new phase: selecting a preferred solution among all the optimal solutions. However, this is not a topic relative to this study, so it won't be detailed further. In order to compute the Pareto front, a utility function, denoted as Φ, is introduced. Φ can represent a combination of all the objective functions f_m. For example, it can be expressed as:

$$\Phi = \sum_{m=1}^{M} w_m f_m, \tag{2.8}$$

where w_m are the corresponding weights of f_m, indicating the prioritization or importance of each objective. Figure 2.3 illustrates how a Pareto member is not dominated by any other individual, as shown by its dominance boundaries, and how a non-Pareto

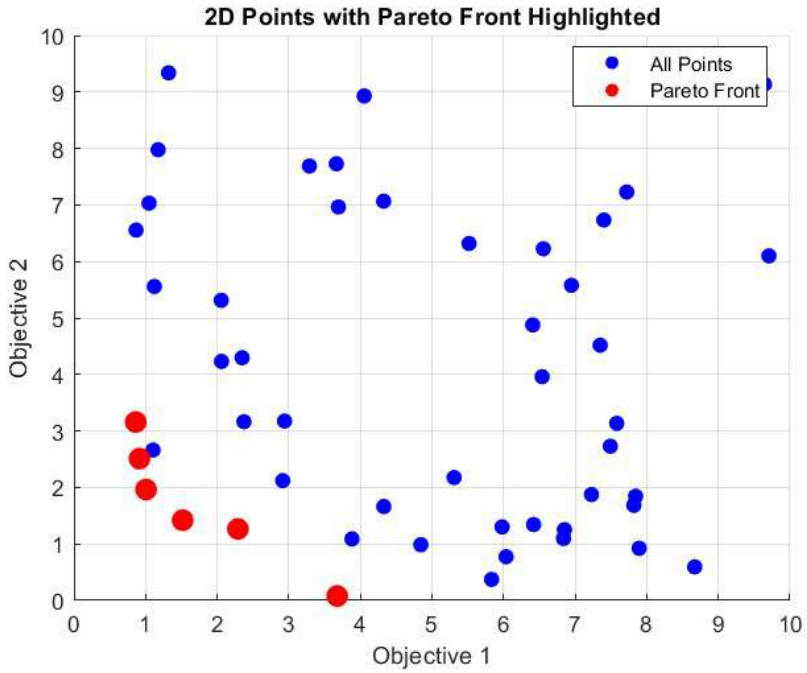

Figure 2.2: The red dots denote the Pareto front in a two-objective minimization problem, while the blue dots present the some dominated solutions by the Pareto front.

individual dominates others, indicated by its extended dominance lines.

Calculating Φ using weighted values for the objective functions is a basic approach that often leads to suboptimal results. To address this issue and deliver improved outcomes in computing the Pareto front, various algorithms have been developed, such as NSGA (Nondominated Sorting Genetic Algorithm)[51], SPEA (Strength Pareto Evolutionary Algorithm)[52], and their evolved alternatives. Since SPEA-II is utilized in the CSO algorithm this thesis is dealing with, further details on it can be found in this chapter.

2.3.1 SPEA

Strength Pareto Evolutionary Algorith (SPEA) [52] is a many objective optimization algorithm and belongs to the field of evolutionary multiple objective algorithms.

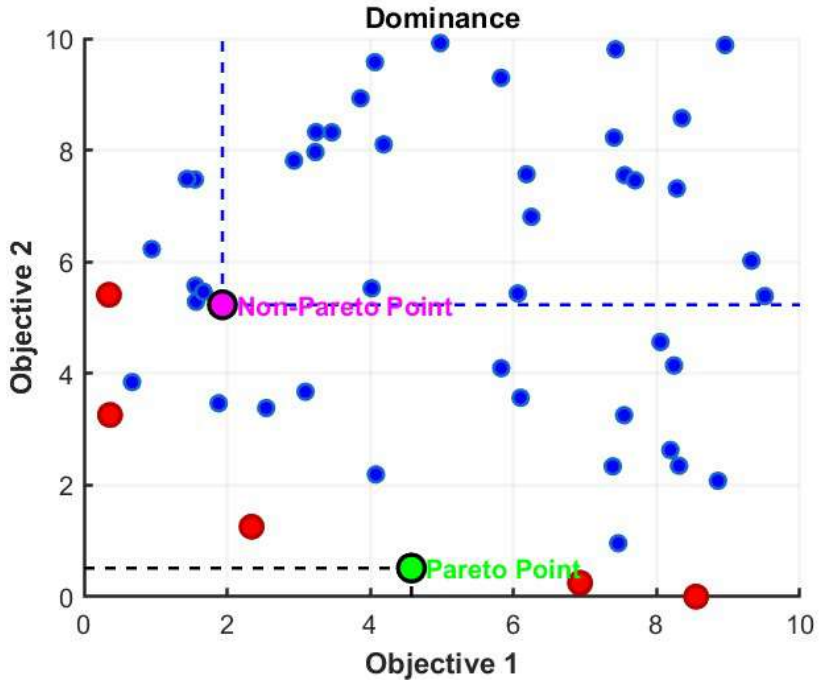

Figure 2.3: Non-Dominated solutions in red denote the Pareto front with green dot being a random solution on the Pareto, while the blue are some domianted solutions and the magenta dot present a random dominated solution by the Pareto front.

The "Strength Pareto" concept plays a critical role in SPEA. The objective of the algorithm is to identify and preserve a set of non-dominated solutions, ideally forming a Pareto optimal set. All Pareto optimal solutions are part of the "Pareto optimal set", which consists of the best non-dominated solutions in the objective space.

Two main parameters are considered for each solution:

1. **Strength Pareto ($S(i)$)**:

$$S(i) = \frac{n}{\mu + \lambda + 1}, \tag{2.9}$$

where n is the number of individuals (each representing a solution vector) that are dominated by or are equal to individual i. Dominated solutions have lower strength values compared to others according to Eq. (1).

2. **Fitness Value ($\Phi(j)$)**:

$$\Phi(j) = \sum_{\forall i<j} S(i), \tag{2.10}$$

where $\Phi(j)$ for individual j in the population is calculated by summing the strength values $S(i)$ of all population members that dominate ($\succ$) or are equal to individual j.

This fitness assignment implies that solutions with lower fitness values are better.

2.3.2 SPEA-II

Similar to the steps presented in Subsection 2.3.1 for the basic SPEA algorithm, the enhanced version, SPEA-II [53], is presented in the following paragraphs. This improved algorithm incorporates mechanisms to better preserve diversity and ensure convergence toward the Pareto front.

Let $G^{t,\lambda}$ denote the main population at generation t. Similarly, let $G^{t,e}$ represent the archive of elite individuals at generation t, which contains the best non-dominated solutions found so far.

2.3.2.1 Steps of SPEA-II

(1) Initialization: Set $t = 0$ and generate the initial population $G_0^{t,\lambda}$ and empty archive $G_0^{t,e}$.

(2) Calculate Fitness Values: For each individual i in both the archive elite group $G^{t,e}$ and the offspring group $G^{t,\lambda}$, the strength value $S(i)$ is calculated using the following equation:

$$S(i) = \left|\{j \mid j \in G^{t,\lambda} + G^{t,e} \& i > j\}\right|, \tag{2.11}$$

where the symbol $\cup$ stands for the multiset union, $\succ$ corresponds to the Pareto dominance relation, and & means AND.

In SPEA-II, the fitness $F(i)$ is defined as:

$$\Phi(i) = R(i) + D(i), \tag{2.12}$$

where the raw fitness of individual i can be calculated by the following equation:

$$R(i) = \sum_{j \in G^{t,\lambda} \cup G^{t,e}} S(j). \tag{2.13}$$

However, if the optimization goal is to minimize the objective $f(i)$, the raw fitness should be minimized, i.e., $R(i) \to \min$, corresponding to a non-dominated individual.

The individual's density $D(i)$, for distinguishing those with the same raw fitness values, is calculated using the κ-nearest neighbor method, using the following equation:

$$D(i) = \frac{1}{\sigma_i^{\kappa} + 2} \tag{2.14}$$

where σ_i^{κ} represents the objective-space distance between the i-th individual and the κ-th nearest neighbor. The κ-th nearest neighbor is defined as $\kappa = \frac{\mu+\lambda+\epsilon}{c}$, where ϵ is the number of elites and c is a constant integer value on the order of 2–3. In this way, the κ-th nearest neighbor dynamically changes according to the number of elites in each generation.

2.4 Metamodel-Assisted Evolutionary Algorithm (MAEA)

In a lot of optimization problems, EAs require a large number of evaluations to reach high-quality solutions. The computational cost per generation is dominated by the evaluation of the objective functions, especially when these evaluations involve time-consuming simulations or external software. Typically, this cost scales with the population size λ, leading to a total of λg evaluations over g generations, excluding those that already exist in the database. When λg becomes very large, the total runtime may exceed the limits of what is feasible using conventional CPU resources.

To address this generic issue, the concept of Metamodel-Assisted EAs (MAEAs)[54], [55], [56] has been introduced. A metamodel significantly reduces the overall computational cost while maintaining good convergence behavior. MAEAs operate using the already discussed selection and reproduction techniques, but with the addition of metamodels that can predict the objective function values of candidate solutions. In simple terms, a metamodel can be seen as a function with adjustable weights, which are tuned to approximate the objective function's value. To ensure a good approximation, a database is required containing pairs of previously evaluated individuals and their responses. The metamodel uses this data to train itself on-line, separately for each new offspring to be evaluated, and approximate its objective functions, much faster, thus dramatically reducing the overall optimization cost.

MAEA Functionality

The operation of a MAEA can be broken down into two key phases [54], [55]. The first phase (which runs as a conventional EA) involves filling the database with sufficient data to ensure that the metamodels built during the process are effective. If previous exact evaluations have already been performed using the same software, these can directly be incorporated into the database before running the MAEA. If these evaluations are deemed adequate for tuning the metamodels, this phase can be skipped. However, if not, the MAEA behaves like a standard EA during the initial generations. This means each new candidate solution is evaluated using the evaluation software and then added to the database, along with its corresponding objective function value.

Once this initial phase is completed, the second phase begins. During reproduction, a local metamodel is created for each offspring, trained using the data from nearby evaluated solutions in the database. This results in the creation of λ personalized metamodels, each capable of providing an approximate evaluation of the λ offspring. The population is then ranked based on the metamodel predictions, and the top individuals are sent for exact evaluation by the software to verify their characteristics. The user can specify the number of individuals to be re-evaluated on the problem-

specific model, denoted as λ_e, with the requirement that $\lambda_e \ll \lambda$ so that the use of metamodels still offers a significant reduction in computational cost. The exactly evaluated individuals are then added to the database to further enrich it for future usage.

This process is known as Low-Cost Pre-Evaluation (LCPE) and has proven to be highly efficient, especially when Radial Basis Function Networks (RBFNs) are used as the metamodels [54], [55], [56]. It is also particularly efficient when applied to MOO problems [57].

Chapter 3

Chicken Swarm Optimization

Among other optimization methods, bio-inspired algorithms have become a prominent area of research, demonstrating their potential across various domains. These algorithms draw inspiration from natural phenomena and the behavior of living organisms, offering innovative approaches to solving complex optimization problems. In particular, swarm optimization algorithms, which replicate the collective behavior of groups of organisms, have shown exceptional efficiency and effectiveness in tackling challenges in fields such as machine learning and engineering.

This chapter delves into a specific algorithm in this category: the CSO Algorithm [25]. Over the last decade, CSO has emerged as a promising meta-heuristic algorithm, attracting substantial interest from researchers[42] due to its simplicity, adaptability, and robust performance. By modeling the social behaviors and hierarchical structures observed in chicken swarms, the CSO algorithm offers a unique and effective approach to optimization.

The chapter is structured as follows: It begins with an overview of the fundamental principles underlying the CSO algorithm, providing a conceptual understanding of its mechanics. Subsequently, the implementation of the algorithm, as developed by the author, is detailed.

3.1 CSO Algorithm

A simplified version of the CSO algorithm is presented in Table 3.1. In this table, TN is the total number of chickens, RN is the number of roosters, HN is the number of hens, CN is the number of chicks, MN is the number of mother hens, G is the population re-arrangement factor and Max_Generations is the maximum number of generations.

Chicken Swarm Optimization Algorithm
Initialize a population of TN total number of chickens and define the related parameters (RN, HN, MN, CN, G, Max_Generations, Number of Objectives, boundaries for each design variable).
Evaluate the fitness values of the TN chickens, $t = 0$.
While $t <$ Max_Generation:
If $t \quad \% \quad G = 0$:
Rank the chickens' fitness values and establish a hierarchical order in the swarm.
Divide the swarm into different groups and determine the relationship between the chicks and mother hens in a group.
End if
For $i = 1 : TN$:
If i is a rooster, update its solution/location using equation 3.3.
If i is a hen, update its solution/location using equation 3.6.
If i is a chick, update its solution/location using equation 3.7.
Evaluate the new solution.
If the new solution is better than its previous one, update it.
Set the elite individuals.
The whole is added to the database.
End for
End while

Table 3.1: Framework of the CSO.

Implementation Details

The CSO algorithm, discussed in this study, is developed for minimization and operates in a more straightforward manner than what has been explained previously, specifically, it employs the following rules. Within a flock, there are distinct populations of roosters, hens, and chicks. Each of them is structured into units that are referred to as groups whereby each unit has one rooster, several hens and chicks. The structure of the groups is defined from top to bottom as the rooster which sits at the top, hens in the middle and chicks at the bottom. The rooster's count determines the number of groups in the flock. The user defines a number of parameters for the algorithm including the number of roosters and the number of hens and chicks. So, since the total population is accounted for by the sum of these three, it does not need to be defined explicitly by the user. The user also specifies the new generation factor G which essentially determines when the population rearrangement mechanism is triggered, the maximum allowable generations, and the number of design variables and their limits. More detailed information about the input file can be found in Appendix 5.1.

The mechanism of constraint handling and penalty assignment is discussed in more detail in a later section. However, it is important to note that, in this approach, the nominal (C_i^{THR}) and relaxed (C_i^{D}) threshold values are user-defined. The individuals exceeding C_i^{D} are considered to have incurred a "death" penalty and must be reintroduced into the population through a resurrection mechanism.

The initial positions of all chickens are generated randomly using a uniform real distribution, ensuring they lie within the boundaries of the design variables. The hens are assigned to their rooster mates as follows: First, the total number of hens is divided by the number of roosters. Each rooster is initially assigned the quotient number of hens randomly. Any remaining hens (those left over after the division) are then distributed one by one to the roosters, continuing serially, until all hens in the population are assigned to the roosters in the flock. For example, if there are two roosters and seven hens in the flock, then each rooster is assigned with three hens and the one remaining is assigned to the rooster with the best fitness. So, the first/best

rooster has four hens in its group and the second has three hens. The same rule applies to the assignment of chicks to their mother hens. Note that the hens that are declared as mothers are randomly selected. Once the initial population (Generation 0) has been constructed, its evaluation is ready to be performed.

After initialization and the first evaluation, the main loop of the algorithm begins. At each generation, the position of each individual is updated according to mathematical formulas that will be detailed later in this chapter. Once the new position is computed, it is checked against the predefined boundaries. If the position exceeds these boundaries, the position is assigned the value of the boundary. If it is iteratively modified by adding or subtracting values to ensure it lies within the boundaries, until the solution is within the acceptable range, the algorithm loses its sequence, as the hens may not follow their mate rooster and the chicks their mother hen. This process not only ensures that the solution stays within the specified range but also minimizes the likelihood of multiple individuals lose their connection with other related individuals, which can occur when the solution is adjusted to lie on the boundaries.

While the function for evaluating individuals is running, the algorithm calls an external evaluation software, a separate program responsible for assessing the individuals. The necessary information for this process is retrieved from specific files. More details about the input file can be found in Appendix 5.2.

The algorithm, then, determines whether the chickens are in a better position with respect to the constraints. First, the fulfillment for constraints, if any, is checked. If all constraint values are below their respective thresholds, the solution is considered to respect the constraints. However, if the value of at least one constraint exceeds the C^{THR} but remains below the C^{D}, the chicken incurs a penalty in its fitness values, calculated as follows:

$$f_{\mathrm{penalized}} = f + Penalty \tag{3.1}$$

where

$$Penalty = \sum_{i}^{N_c} e^{3\frac{g_i - C_i^{\mathrm{THR}}}{C_i^D - g_i}} f, \tag{3.2}$$

where N_c is the number of constraints, c_i is the computed constraint, C_i^{THR} is the nominal threshold, and C_i^D is the relaxed threshold value. From this point onward, every chicken sorting operation is performed using $f_{\mathrm{penalized}}$. Naturally, if there are no constraints or if the chickens respect the constraints, the values of f and $f_{\mathrm{penalized}}$ are identical. In the case that more than one constraint is not respected, the higher penalty value is used for the penalty computation. If the value exceeds the C^{D}, the chicken incurs a "death" penalty, meaning that the penalized fitness $f_{\mathrm{penalized}}$ becomes the largest possible value, representing "death". Since the algorithm has no reason to continue in this state, a new chicken takes its place from the database. This new chicken may be penalized or not, but certainly will not carry a "death" penalty. If the database is empty (which occurs in Generation 0) or if all chickens suffer a death penalty (an extreme case), a completely random position is assigned to the new chicken. The role of the chicken (whether rooster, hen, or chick) remains unchanged.

The penalized fitness value determines the classification of the chickens. Those with the smallest fitness values, since it is a minimization algorithm, are classified as roosters, while those with the largest fitness values become chicks, and those in between are classified as hens. Fitness values are calculated after the evaluation of each individual by the evaluation software. Each individual's position and fitness are updated only if the calculated fitness is better than the previous one and if it does not carry a "death" penalty. Otherwise, the position of the chicken is not updated, even if it respects the constraints. However, a chicken may still be updated even if its fitness value is worse, as long as it respects the constraints, since solutions that respect the constraints are the primary goal.

The evaluation of the entire population is carried out at every generation. However, the rearrangement of populations is conducted only after every G generations. The process of assigning rooster mates, mother hens, and chicks is performed in the same manner as described above. After every iteration, every chicken that has been evaluated is stored in the database.

The database only keeps the essential information: the chicken's ID, its design vector, the fitness values (for MOO, this includes all fitness values; for SOO, there

is just one fitness value), penalized fitness values, constraint values, and whether the chicken is penalized or not. A chicken is considered non-penalized if all of its constraint values meet the constraints, otherwise it is penalized. Also, in unconstrained optimization, the fitness values and the penalized fitness values are identical. As for the algorithm's implementation, all data is stored in C++ objects for processing and it uses real encoding by default.

3.1.1 Movement of the chickens

The mathematical formulas for the calculation of chicken positions are presented. The position of the i-th individual in the j-th dimension at the t-th generation is denoted by $x_{i,j}^t$. The movement of chickens within the design space follows stochastic rules based on their hierarchy: **roosters** lead the group, followed by **hens**, and finally, **chicks**. The movement of the chickens is described in hierarchical order. First, the movement of the **roosters** is described, followed by that of the **hens**, and finally, the movement of the **chicks**.

It is important to note that the formulas used to compute each individual's position have been modified. The basic structure presented in [25] is followed; however, the original algorithm is quite sensitive and prone to getting trapped in local minima. To address this issue, a later study presented in [58] introduces a modification to the chicks' position update formula. This idea can be further expanded to enhance the overall performance of the algorithm. Therefore, an additional modification is proposed by the author, which involves changes to the position update formulas for both roosters and chicks.

3.1.1.1 Movement of roosters

Each rooster serves as the leader of its groups. Those with better fitness values are given priority access to food over those with lower ones. Consequently, roosters with superior fitness values should move across a wider area in the design space, reflecting their dominant role and enhanced capability to explore resources. According to [25], this can be formulated as:

$$x_{i,j}^{t+1} = x_{i,j}^{t} \cdot (1 + \mathcal{N}(0, \sigma^2)) \tag{3.3}$$

$$\sigma^2 = \begin{cases} 1, & \text{if } f_i \leq f_k, \\ \exp\left(\frac{f_k - f_i}{|f_i| + \epsilon}\right), & \text{otherwise} \end{cases} \tag{3.4}$$

where $k \in \{1, N\}$ and $k \neq i$. When there is only rooster in the flock $k = i$ and $\sigma^2 = 1$. The term $\mathcal{N}(0, \sigma^2)$ refers to a Gaussian distribution with mean value 0 and variance σ^2. The constant ϵ is used to avoid division by zero and represents the smallest positive constant in the computer system. The variable k denotes the index of a rooster, which is randomly selected from the group of roosters. The function f represents the fitness value associated with the corresponding position x.

There some cases, however, where roosters may be trapped in local minima, causing the group to get trapped as well. In order to overcome this drawback, a simple idea is proposed. The rooster with the worst fitness is forced to take a random position in the design space with a 50% probability. In case this happens, there two different formulas that can be chosen with equal probability each, the first is a complete random position in the design space and the second is given by this formula:

$$x_{i,j}^{t+1} = x_{\min,j} + (x_{\max,j} - x_{\min,j}) \cdot \text{Rand}(0, 1) \tag{3.5}$$

where $\text{Rand}(0, 1)$ is a uniformly distributed random value in the range $[0, 1]$. This mechanism acts as a strong mutation operator, similar to the one employed by EASY. For example, in the case where only one rooster forages in the flock and it might mislead the group, this mechanism is applied to it, as it is considered to be the "worst" individual at that time.

Another modification occurs when updating the position of a rooster. In SOO problems, if the best solution does not improve after two generations, the standard deviation is set to 0.01. This adjustment assumes that the current region might contain the best solution, and thus intensifies the search in a smaller, more focused area. Similarly, in MOO problems, the same value is assigned to the standard deviation

if the number of solutions on Pareto front remains unchanged for two consecutive generations.

3.1.1.2 Movement of hens

The hens follow their group mate rooster to search for food. Furthermore, they may stand in a better position and steal food from other hens in a different group. The position of a hen in the flock is calculated as follows.

$$x_{i,j}^{t+1} = x_{i,j}^{t} + S_1 \cdot \text{Rand}(0,1) \cdot \left(x_{r_1,j}^{t} - x_{i,j}^{t}\right) + S_2 \cdot \text{Rand}(0,1) \cdot \left(x_{r_2,j}^{t} - x_{i,j}^{t}\right) \quad (3.6)$$

where $S_1 = \exp\left(\frac{f_i - f_{r_1}}{|f_i| + \epsilon}\right)$, $S_2 = \exp\left(\frac{f_{r_2} - f_i}{|f_{r_2}| + \epsilon}\right)$, Rand is a random number sampled from a uniform distribution over $[0, 1]$, and the indices $r_1 \in \{1, 2, \ldots, TN\}$ the index of the rooster that is a group-mate of the i-th hen, and $r_2 \in \{1, 2, \ldots, TN\}$. The index of a randomly chosen chicken (rooster or hen) from the entire swarm. The constraint $r_1 \neq r_2$ ensures that the indices are distinct and can never be equal. As for the fitness values f_i, f_{r_1}, f_{r_2}, it is expected that $f_i > f_{r_1}$ since a rooster is expected to have a lower fitness value than a hen. Additionally, $f_i > f_{r_2}$ if r_2 is a rooster or a hen in the first half of the cumulative population of hens and roosters, and $f_i < f_{r_2}$ if r_2 is a hen in the second half of the cumulative population. When $f_i > f_{r_1}$, $S_1 > 1$, which indicates that the i-th hen is influenced more by its rooster group-mate. It is important to note that S_2 can take very high values, leading to delays. Therefore, it has been decided to normalize the exponent, rather that keep the exponent as the difference of $f_{r_2} - f_i$ [25].

3.1.1.3 Movement of chicks

As mentioned above, chicks forage for food only around their mother hen. This is formulated as:

$$x_{i,j}^{t+1} = x_{i,j}^{t} + FL(0,2) \cdot (x_{m,j}^{t} - x_{i,j}^{t}) \quad (3.7)$$

where $x^t_{m,j}$ denotes the position of the i-th chick's mother ($j \in \{1, 2, \ldots, N\}$).

FL ($FL \in \{0, 2\}$) is a parameter indicating that the chick would follow its mother to forage for food. Considering individual differences, the FL of each chick is randomly chosen between 0 and 2.

To address the early convergence problem of the algorithm, [58] proposed a modification whereby chicks are allowed to update their positions using an alternative strategy with a probability of 25%, instead of strictly following Equation 3.7. To promote a broader exploration of the design space, when a chick is selected to update its position differently (with the probability of 25%), it either follows the rule defined in Equation 3.8 with a probability of 50%, or is assigned a completely random position—analogous to the position update strategy employed by roosters—with the same probability.

$$x^{t+1}_{i,j} = x_{\min,j} + (x_{\max,j} - x_{\min,j}) \cdot \text{Rand}(0, 1) \tag{3.8}$$

3.2 Single-Objective CSO

In SOO, the problem is typically defined by a function $f(\vec{X})$ to be minimized. The optimization process involves exploring the search space to identify the solution $\vec{X}^*$ that minimizes $f(\vec{X})$ while adhering to the specified constraints.

In this thesis, the solution $\vec{X}^*$ is determined through the movement of chickens within the design space. Each solution is required to respect the boundaries and constraints defined by the problem. Understanding the dynamics of chicken movement in the search space is key to solving optimization problems effectively, as it mimics natural processes that help in exploring the solution space efficiently. By leveraging the movement patterns of different types of chickens (roosters, hens, and chicks), we aim to find the best possible solution for the given optimization problem.

After the evaluation of a chicken, the algorithm checks whether the new position yields a better fitness value. If it does, the penalty status check is performed. If any of the chicken's constraint functions exceed the nominal death value, it receives

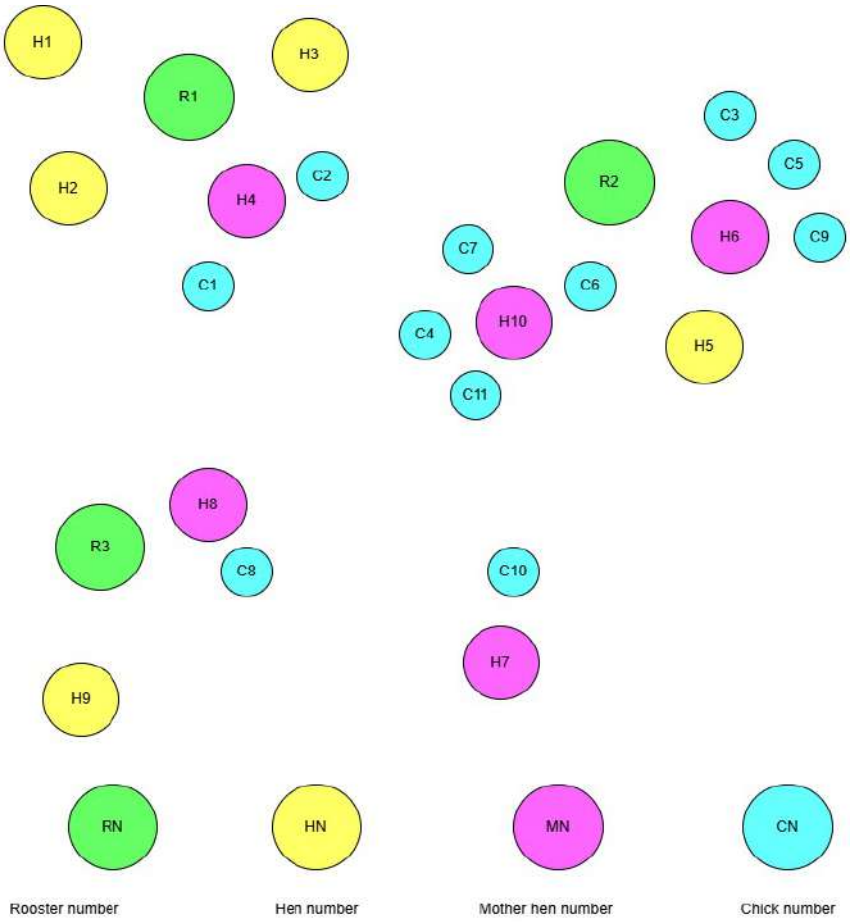

Figure 3.1: Example of a flock in groups

a death penalty and is resurrected this actions is detailed later in Subsection 3.4.4). If the constraints do not exceed the C^{D} but are still above the C^{THR}, a penalty is applied and the chicken returns to its previous position. Otherwise, it successfully updates its position.

If the chicken does not improve its fitness value, it returns to its previous position, as described earlier, unless it is forced to update its position according to the mechanism detailed in Subsection 3.1.1.

3.3 Multi-Objective CSO

The concepts discussed above are specific to SOO problems. For MOO, additional computations are incorporated. In this thesis, the algorithm employs the SPEA-II to calculate the fitness (Φ) of each individual. A key difference in the implementation of SPEA-II within the CSO framework is that the κ-nearest neighbor is defined as the closest individual to the examined individual, based on their actual distance in the objective space. It is not computed using the formula $\kappa = \frac{\mu+\lambda+\epsilon}{c}$. This approach ensures the effective handling of multiple conflicting objectives by balancing trade-offs and maintaining a diverse set of Pareto-optimal solutions.

For the elite population, the algorithm stores only those individuals that belong to the Pareto front, even if the user requests a larger number of individuals. Conversely, if the user requests fewer individuals than the total number of elites, the algorithm stores and returns only the specified number of individuals, selected in the sequence they were calculated.

3.4 Elite Selection

The selection of elites plays a significant role in the effectiveness and performance of the optimization process. As mentioned earlier, after evaluation, a chicken updates its position only if it achieves a better fitness value while still respecting all constraints. In SOO problems, this process is relatively straightforward. However, in MOO problems, additional challenges arise due to the need to balance multiple conflicting objectives, making the selection criteria more complex.

3.4.1 Elites in SOO problems

In SOO problems, the solution is checked to determine whether the penalized fitness f_p is better than the previous one. If f_p is better and the chicken is not penalized at all, then the solution is updated as indicated by f_p, and in such a case the penalized fitness is equal to the normal fitness, $f_p = f$.

Otherwise, if the chicken's "death" status and previous "death" status are both *true*, then the chicken undergoes a resurrection procedure, which is discussed later. If the chicken's "death" status is *true* but was *false* in the previous state, then the chicken returns to its previous state so that it can continue searching for a better solution without being penalized. If the solution is not improved, then once again, the chicken resurrects if both the current and previous "death" statuses are *true*. Otherwise, it returns to its previous state.

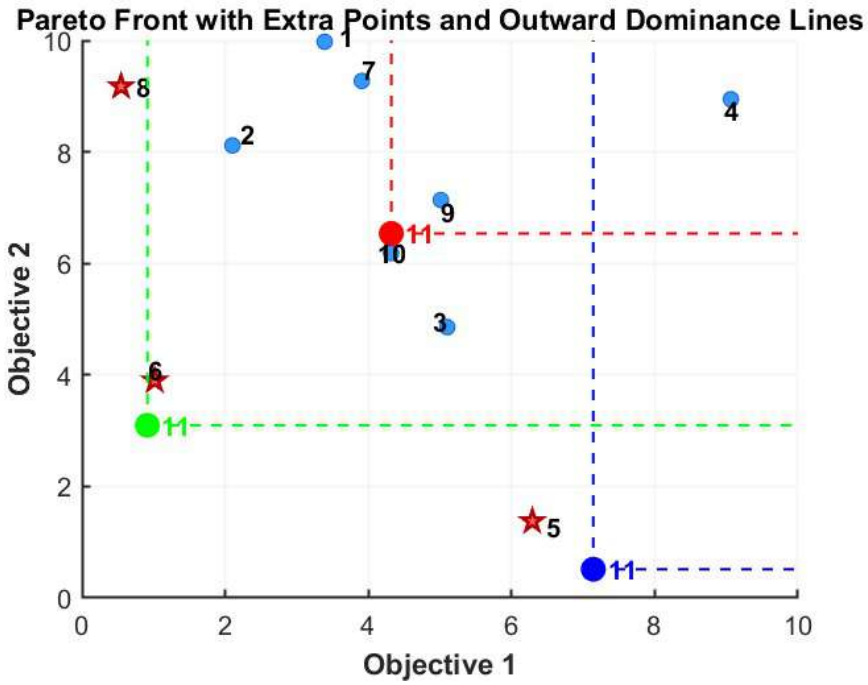

Figure 3.2: Elite Selection in MOO. The new individual, if it is in the green or in the deep blue position, is added in the elites group. In contrast, the one in the red position does not.

3.4.2 Elites in MOO problems

In MOO, things become more complicated. To fully comprehend the method, let us consider a simple example of a MOO problem with two objective functions. In this example, 11 individuals are used, and the result shown in Figure 3.2 is taken after

t generations. Individuals 1, 3, and 7 form the Pareto front as it was before the t^{th} iteration.

Suppose the eleventh individual has moved within the search space and obtained objective function values that lead to three possible conditions, as illustrated in Figure 3.2. If this position is the one denoted by the red dot, then it does not belong to the Pareto front. If it takes the position denoted by the intense blue dot, then it is not dominated and it does not dominate any other individual on the Pareto front; thus, this individual is added to the Pareto front. In the last case, denoted by the green dot, this particular individual dominates one individual on the Pareto front and is not dominated by any other individual in the front. Consequently, the sixth individual is replaced by the eleventh.

Back to the CSO now, having in mind the discussion above, the algorithm takes into account not only the raw fitness calculated by the SPEA-II algorithm to compare individuals for the elite selection, but also the mechanism discussed in the previous example. Specifically, if an individual takes a position corresponding to the intense blue dot illustrated in Figure 3.2, it is added to a container holding all possible elite individuals. The actual elite selection occurs later. In case the position is as denoted by the red dot, the following may happen:

Red Dot Conditions
Current "death" status **true** and Previous "death" status **false** → return to previous state.
Current "death" status **false** and Previous "death" status **true** → update to current state.
Current "death" status **true** and Previous "death" status **true** → chicken resurrects.
Current penalty status **false** and Previous penalty status **true** → update to current state.
Current penalty status **true** and Previous penalty status **false** → return to previous state.
Current penalty status **true** and Previous penalty status **true** → if the raw fitness is better, update to current state; otherwise, return to the previous state.

If the position is as denoted by the green dot, the following cases apply:

Green Dot Conditions
Current penalty status **false** → update to current state.
Current "death" status **true** and Previous "death" status **false** → return to previous state.
Current "death" status **true** and Previous "death" status **true** → chicken resurrects.
Current "death" status **true** and Previous "death" status **false** → return to previous state.
Current penalty status **true** and Previous penalty status **true** → update to current state (this happens because the individual is in a better position; although it is penalized, it is allowed to continue since it might improve and eventually respect the constraints in a later state).

It is important to note that, every time a chicken updates its state, even if it is penalized, the rationale is to allow it to continue moving, with the expectation that it will eventually reach a better state.

Later, every solution found during the current iteration—including the entire population, the already existing elites, and the possible elites—is gathered together. All these solutions are then compared with each other to separate dominated from the non-dominated ones. The solutions that are not dominated by any others form the newly created elite group.

Moreover, when the search for elites is performed, if an individual is found to be identical to one in the elite group, a possible replacement occurs. This means that the identical chickens acquire a noise in their position and are reinjected into the population. As a result, they replace chickens with the worst fitness while also inheriting the attributes associated with their specific ID to maintain connection in the population. Specifically, roosters retain their assigned hens, hens keep their mate rooster and chicks, and chicks inherit their mother hen.

Last but not least, an additional mechanism is implemented to handle penalized chickens in MOO problems. During population rearrangement, each penalized chicken's position is replaced by that of a randomly selected elite individual, with added noise. This happens with a probability of 75%.

When the maximum number of elites has not yet been reached, the noise is con-

strained to a smaller region around the selected elite, increasing the likelihood of discovering and adding another elite individual to the elite set. Otherwise, the noise is applied over a broader area, promoting exploration of different regions in the design space. This broader search comes with the risk of placing individuals in areas where they may be penalized. This mechanism is introduced to address situations where only a few non-penalized individuals are available to form the elite pool. By forcing penalized individuals to relocate near elite ones, the pool of potentially high-performing solutions is expanded, thereby enhancing the overall algorithm performance.

3.4.3 Thinning

The user of the algorithm has the freedom to select how many elites can be on the Pareto front. This means that the algorithm must handle the case when the maximum elite number requested by the user has reached its limit and one more asks to enter the elite group. The simplest way to handle such a case, albeit quite ineffective, is to discard any chicken beyond that limit. This solution may satisfy the maximum limit set by the user, but it may lead to a Pareto front with sparse and dense regions in several spots. Consequently, the quality of the Pareto front is reduced, making it more challenging for the user to understand and accurately evaluate the results.

In this thesis, to make the process smoother distances, as measured by the penalized fitness of each chicken, are calculated. The two individuals with the minimum distance are kept as candidates for replacement when a new elite candidate arrives. The distance between every chicken and the new elite candidate is then calculated, and if the minimum of these distances is greater than the minimum distance between the two old candidates, one of the old candidates is replaced, with the selection happening randomly. No specific criteria are applied for this action.

3.4.4 Chicken Resurrection

As mentioned previously in this section, some chickens according to some criteria are resurrected in order to play again their roles in the algorithm. This mechanism is presented here.

When a chicken dies, a spot gets empty in the population. If no action was taken to replace this spot, then the total size of the population would vary, and the algorithm would lose control. In order to preserve and control the algorithm, the total population must be fixed.

At first, the algorithm checks the ID of the chicken to determine whether it is a rooster, hen, or chick. A search is performed in the database, where every previously calculated individual is stored, and a random individual with the same ID is returned. Then, its position is modified with a random noise, and a new chicken is reintroduced into the population.

Except for the position, the newly created chicken is assigned the same relationships it had before its death. The rooster is assigned its hens, the hen is assigned its mate and its chicks, and the chick is assigned its mother hen. If the database is empty, which happens during the first iteration, the chicken is initialized with the highest possible fitness and a random position. In the extreme case where in the Generation 0 all chickens are about to die, then the same action is performed for 5 times. If their "death" status is true every time, the algorithm terminates and the user should try again.

3.5 DataBase

The database is used to store all evaluated individuals and is enriched in every iteration. If an individual already exists in the database, it is discarded. In this thesis, the implementation does not follow the structure of a regular database, like MySQL etc.. Instead, it is a C++ structure containing objects with specific values. The values stored in the database include the positions of the chickens, their fitness values, penalized fitness values, constraints, raw fitness values, and their IDs.

Every time a chicken's position is calculated, it is checked against the database to determine if it already exists. If it does, the stored chicken from the database is used instead of the newly calculated one. This approach reduces the number of evaluations performed. Additionally, the database is utilized for chicken resurrection as seen in Subsection 3.4.4.

A database may also serve other purposes. One of them is to train metamodels to enhance the effectiveness of stochastic algorithms, as done in a MAEA.

3.6 Hyper-Volume Indicator

A quantitative measure is required to assess the closeness and quality of an estimated set of solutions relative to the Pareto front. One of the most widely used and effective indicators for this purpose is the *Hypervolume Indicator* (HVI).

The hypervolume indicator measures the volume of the objective space that is weakly dominated by the estimated Pareto front and bounded by a predefined reference point. It simultaneously captures both convergence—how close the solutions are to the Pareto front, and diversity, how well the solutions are distributed along the front. A higher HVI value indicates that the Pareto front dominates a greater portion of the objective space, assuming the reference points are fixed and appropriately selected, and it is commonly used as an indicator of convergence in MOO problems.

In this study, the HVI is employed to compare the quality of the Pareto fronts produced by the EA or MAEA and CSO algorithms.

Chapter 4

Applications - Discussion

Initially, every optimization algorithm is tested using a series of well-known benchmark problems. These problems are mathematical functions, commonly referred to as test functions. Widely recognized examples include the Rastrigin function, the Ackley function, and others, which are characterized by many local optima and a single global minimum, typically at zero (0, 0, 0,...0).

Since these problems are inherently challenging, optimization algorithms often risk getting trapped in local optima. To better evaluate an algorithm's performance and robustness, it is beneficial to shift these functions so that the global optimum be located at a value other than zero. This allows for a more thorough assessment of the algorithm's ability to explore the search space effectively.

Moreover, the complexity of the problem increases when a larger number of design variables are introduced, making the optimization process even more demanding. This provides an advantage when testing an algorithm compared to pseudo-engineering problems, which are also commonly used. It is important to note that these test functions do not include any constraints.

Pseudo-engineering problems, on the other hand, represent scenarios that are closer to real-world applications, providing a more practical approach for testing optimization algorithms. Unlike test functions, the number of design variables in pseudo-engineering problems is fixed, making them easier in this regard. However, many of these problems include constraints, which significantly increase their complexity.

In this thesis, the shifted Rastrigin function is used to examine the parameters of the algorithm for SOO problems. Additionally, EAs or MAEAs and CSO are compared using two pseudo-engineering problems. For each problem, the population size in every generation and the encoding method have been kept the same for both algorithms to ensure the most absolute comparison possible. All other setup paremeters are selected in order to achieve the best possible performance for each algorithm.

4.1 CSO Parameters Analysis

The Rastrigin function is defined as:

$$f(\vec{X}) = 10n + \sum_{i=1}^{N} \left[x_i^2 - 10\cos(2\pi x_i)\right] \tag{4.1}$$

where N is the number of dimensions or design variables. Each dimension corresponds to a variable x_i, where $i = 1, 2, \ldots, N$ and $x_i \in [-5.12, 5.12]$.

The function has many local minima, making it difficult for optimization algorithms to find the global minimum. The global minimum occurs when $x_i = 0$ for all dimensions, and the function value is $f(\vec{X}) = 0$. The shifted Rastrigin function used for this analysis has its global minimum at $x_i = 3$, $i = 1, \ldots, N$. The Rastrigin function is shifted away from the origin because CSO easily finds the global minimum at zero; shifting the function makes the optimization task more challenging. To achieve this, the function is defined as:

$$h(\vec{X}) = f(\vec{X} - \begin{Bmatrix} 3 \\ 3 \\ \vdots \\ 3 \end{Bmatrix}) \tag{4.2}$$

4.1.1 CSO Parameters

This analysis will focus on three parameters of the algorithm: the number of roosters, the number of mother hens, and the regeneration parameter G. The study will examine how these parameters affect the solution of the problem and potentially provide guidance on their optimal usage. The balance remains due to the adjustment of the chicks population, when changing the number of roosters. After several runs, a population size of 60 individuals appears to be sufficient for the algorithm to perform both effectively and efficiently.

Moreover, for the plots and tables presented in this subsection, data from 5 independent runs with different random seeds (initialiazation of the first population) were used. Specifically for the plots, the respective objective function values from each run were used individually, rather than computing average values across runs. For each x-axis value (i.e., the number of function evaluations), if an objective function value was not available in a particular run, the most recent previous objective function values was carried forward. For example, in a case of evaluations 14585 to 14696, if an objective function value of 0.055 was recorded at evaluation 14585 and no new value was available until evaluation 14696, the value 0.055 was used for all intermediate evaluations. At evaluation 14696, when a new objective function value (e.g., 0.026) appeared, the curve changed accordingly.

4.1.1.1 Number of Roosters RN

The number of roosters plays a significant role in the overall swarm mechanism. Roosters are the only individuals that explore the design space independently, without being influenced by the positions of other members of the flock, whether roosters, hens, or chicks. As a result, the performance of the algorithm can be heavily influenced by the quality of the rooster leaders. A poorly performing rooster may misguide its group, negatively affecting convergence. Conversely, a competent rooster leader can significantly improve the search process and help the algorithm find the optimal solution more efficiently. Therefore, the number of roosters warrants discussion, not only to highlight this aspect of the algorithm's design but also to explain how the

algorithm mitigates the risk of poor leadership within the swarm. Table 4.1 shows the constant parameters used for studying the RN parameter. Table 4.2 presents the values and the number of evaluations of every run, of every case.

CSO
$HN = 20$
$MN = 16$
$CN = 39\,/\,37\,/\,35$
$G = 5$
$G\_Max = 350$

Table 4.1: Shifted Rastrigin function. CSO algorithm parameters for RN analysis.

Before diving into the analysis of RN, it is important to highlight an interesting observation. The RN parameter bears resemblance to the number of demes used in the (μ, λ)-EA, known as Distributed EA (DEA). This functionality divides the population into groups corresponding to the demes. These demes can cooperate and exchange information with one another, much like the groups in the CSO algorithm, which interact through the hen population.

RN = 1

When $RN = 1$, there is only one group in the flock, and every chicken follows a single rooster. As a result, the algorithm's convergence toward the best solution becomes highly dependent on the performance of this individual rooster. If the rooster is not effective, the algorithm may struggle to find the optimal solution.

From Tables 4.2 and 4.1, it is evident that the algorithm's performance is significantly worse compared to the other two RN configurations. It requires more evaluations, and the objective function achieves higher values. A likely explanation for this is the rooster's movement mechanism: it moves randomly and only updates its position if it discovers a better solution. However, after two generations (see Chapter 3 for algorithm details), it may randomly relocate in the design space with a 50% probability, which disrupts its trajectory. While this mechanism is intended to help the rooster escape local optima if it gets stuck, it also introduces instability, for this

	RN=1		**RN=3**		**RN=5**		**RN=7**		**RN=9**	
Stat	**Eval**	**Val**	**Eval**	**Val**	**Eval**	**Val**	**Eval**	**Val**	**Eval**	**Val**
	20018	0.25306	19844	0.00721	19529	0.00065	19639	0.00432	19613	0.00641
	20286	2.27877	20000	0.00482	19692	0.00242	20816	1.77915	19598	0.05083
	20809	0.55680	19694	0.01595	19770	0.01136	19448	0.01472	19774	0.00426
	19779	0.02490	19652	0.02893	19803	0.00164	20236	0.01116	19523	0.00322
	20914	0.20350	19520	0.00397	19917	0.00303	19495	0.00870	19644	0.00459
Mean	20361.2	0.66340	**19742.0**	0.01218	19742.2	**0.00382**	19926.8	0.36301	19690.4	0.01346
Median	20286.0	0.25306	19694.0	0.00721	19770.0	**0.00242**	**19639.0**	0.00870	19644.0	0.00459
Std Dev	440.12	0.82563	165.32	0.00939	128.84	**0.00385**	508.20	0.65260	**101.39**	0.01816
Min	19779.0	0.02490	**19520.0**	0.00397	19529.0	**0.00065**	19448.0	0.00432	19523.0	0.00322

Table 4.2: Shifted Rastrigin results for RN = 1, 3, 5, 7, and 9 (5 runs each). Bolded values denote best performance per metric.

case, which, most likely, negatively impacts overall performance.

RN = 3, 5, 7, 9

These four selections give much better performance than $RN = 1$, as expected. The mechanism mentioned above is now applied only to the worst rooster—the one with the highest fitness. As a result, the best roosters continue their search uninterrupted, leading their groups toward the optimal solution in each generation. The worst rooster is free to wander when the time comes, which may allow it to eventually become the best. If that happens, its group will follow it to the better solution.

$RN = 5$ yields slightly better results, as shown in Table 4.2. In particular, it appears more stable, as indicated by its lower convergence value, lower mean, lower median, and smaller standard deviation compared to the other RN values. The only exception is the mean and minimum number of evaluations, which is slightly higher than $RN = 3$, though the difference is negligible.

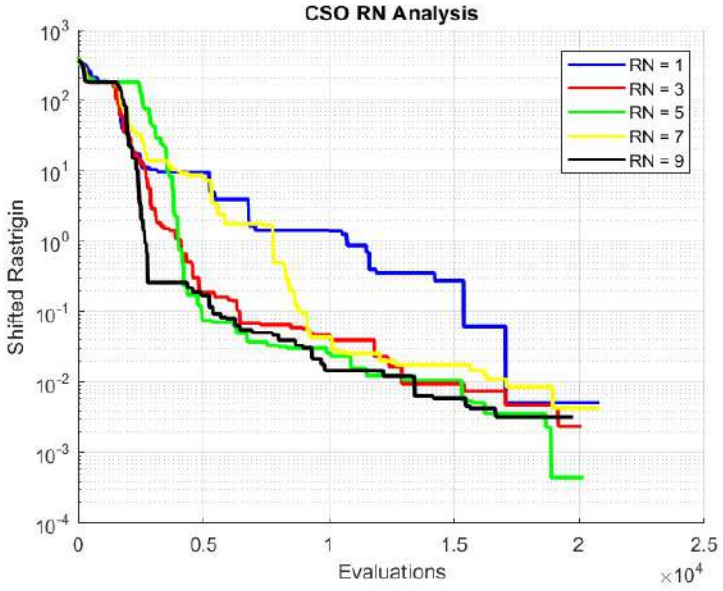

Figure 4.1: Shifted Rastrigin function. Comparison among the mean values of 5 different RN values ($RN = 1, 3, 5, 7, 9$) by using five independent runs with different seeds.

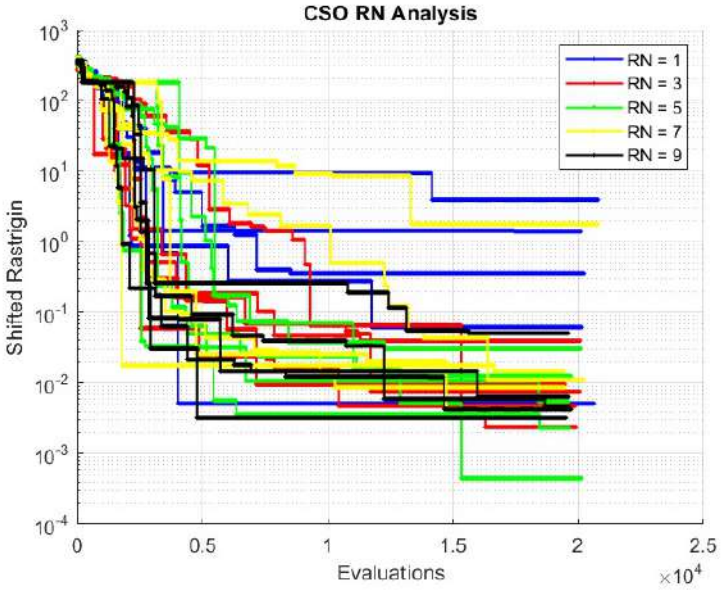

Figure 4.2: Shifted Rastrigin function. This figure presents comparison of all 5 indepedent runs of 5 different RN values ($RN = 1, 3, 5, 7, 9$).

Figure 4.1 provides a graphical representation of the discussion above. In the case of $RN = 1$, it is evident that the algorithm performs worse: it converges more slowly, although it seems to converge a bit faster than $RN = 5$ at first. Also, it reaches a higher final value, and requires slightly more evaluations. However, the comparison between $RN = 3, RN = 5, RN = 7$ and $RN = 9$ reveals an interesting insight. Although earlier analysis suggested that $RN = 5$ yields better overall results, the figure shows that it converges more slowly than $RN = 9$ until approximately 15000 evaluations. Also, $RN = 7$ converges slower, but it reaches almost the same value as $RN = 3$ and $RN = 9$. This behavior highlights the nature of the algorithm when multiple roosters are involved: the worst rooster has the opportunity to become the best relatively quickly, potentially accelerating convergence.

Also, in Figure 4.2, it is evident that for $RN = 3, 5, 7$ and $RN = 9$, the variance is lower compared to the case of $RN = 1$. However, despite the reduced variance, the corresponding curves remain somewhat dispersed.

Based on the presented results, it is evident that having more roosters generally

leads to better convergence, as the underlying mechanism benefits from a larger number of guiding individuals. In the examined case, the best performance was achieved with $RN = 5$, although $RN = 3, 7, 9$ also demonstrated strong results.

4.1.1.2 Number of Mother Hens MN

The number of mother hens is a particularly sensitive parameter that significantly affects the algorithm's performance. It determines which hens will act as mothers and how many chicks will be assigned to each. Since chicks follow only their respective mothers, if a mother explores poorly or inefficiently, her chicks are likely to do the same. At first glance, it may seem risky to use only one mother in the flock. Conversely, if all hens are mothers, there will be fewer chicks per mother, and if some of these mothers are not effective at foraging, many chicks may also perform poorly. Naturally, these assumptions may not be valid for all problem types, so it is advisable for users to experiment with different mother hen configurations to identify the most effective setup for their specific case.

This analysis tries to give the reader a glimpse of how MN works and possibly give some guidance. The table 4.3 presents the constant values of the parameters used for these runs.

CSO
$RN = 5$
$HN = 20$
$CN = 35$
$G = 5$
$G\_Max = 350$

Table 4.3: Shifted Rastrigin function. CSO algorithm parameters for MN analysis.

From Table 4.4, it is easy to observe that, for the shifted Rastrigin problem, selecting $MN = \frac{HN}{2} = 10$ yields the best performance overall. This setting achieves lower minimum and maximum values, along with better mean and median results. Both $MN = 1$ and $MN = 5$ are strong competitors, as they approach the performance of $MN = 10$ with slightly fewer evaluations. In particular, for $MN = 1$, it seems

	MN = 1		MN = 5		MN = 10		MN = 15		MN = 20	
Statistic	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**
	19549	0.00216386	19616	0.00244747	19856	0.00156522	20232	0.0374904	19868	0.00080109
	19582	0.00341881	19607	0.0110887	19570	0.00296417	19854	0.00518246	19950	0.000499899
	19652	0.00110059	20078	0.00002845	19454	0.00000494	19690	0.0728534	19884	0.0815115
	19639	0.0326114	19412	0.00594385	20180	0.00044478	19592	0.00543264	20629	0.0648564
	19433	0.00425258	19726	0.00043761	19531	0.00518246	19678	0.00133934	19530	0.0139208
Mean	**19571.00**	0.00819	19687.80	0.00439	19718.20	**0.00243**	19869.20	0.02406	19972.20	0.03272
Median	19582.00	0.00342	19616.00	0.00594	**19570.00**	**0.00157**	19690.00	0.00543	19884.00	0.01392
Std Dev	**79.41**	0.01285	227.41	0.00409	260.58	**0.00206**	219.04	0.02864	429.33	0.03368
Min	19433.00	0.00110	**19412.00**	0.00003	19454.00	**0.00000494**	19592.00	0.00134	19530.00	0.00050
Max	**19652.00**	0.03261	20078.00	0.01109	20180.00	**0.00518**	20232.00	0.07285	20629.00	0.08151

Table 4.4: Shifted Rastrigin function. Raw values and statistical summary for MN = 1, 5, 10, 15, and 20 after 5 runs. Bolded values denote best performance per metric.

that a single mother hen is sufficient to consistently follow a good rooster in every population rearrangement, achieving good results reliably. On the other hand, when $MN = HN = 20$, the algorithm struggles more in terms of performance. However, this does not imply that choosing $MN = HN$ is always a poor choice; later, it will become clear why, in some problems, setting $MN = HN$ can actually improve the algorithm's performance.

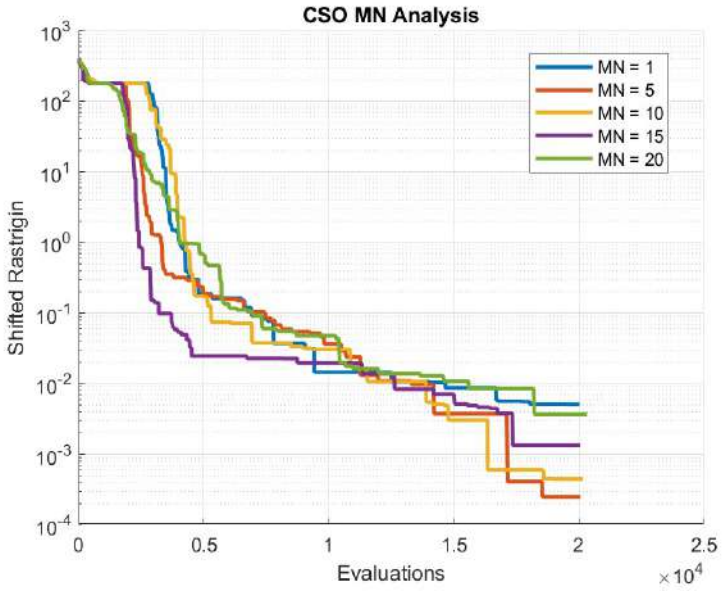

Figure 4.3: Shifted Rastrigin function. This figure presents comparison of mean values of five different MN values ($MN = 1, 5, 10, 15, 20$) by using five independent runs with different seeds.

Figure 4.3 is a good example of the discussion above. The $MN = 5$ case appears to converge to the best value compared to all other cases, although its convergence is slower than the rest. Good performance, both in terms of convergence speed and final value, is achieved with $MN = 10$, followed by $MN = 15$. On the other hand, the worst performance is observed for the $MN = 20$ and $MN = 1$ cases. Based on the above results, for such problems, an intermediate value of MN, neither too small nor too large, should be selected to achieve better performance. For such problems, values around $\frac{1}{4}HN$ or $\frac{1}{2}HN$ are usually sufficient.

4.1.1.3 Population Re-arrangement Parameter G

The purpose of this parameter is to re-arrange the population (Table 3.1 Step 4) . This means it determines when the best chickens become roosters, the worst become chicks, and the middle-ranked ones become hens, with the frequency of these updates dictated by G. It also governs when some of the hens are promoted to mothers. The goal of this analysis is to verify the claims made in [42], which state that the G parameter should be neither too low nor too high. If G is too low, the algorithm may quickly become trapped in local optima, whereas if G is too large, the overall effectiveness of the algorithm declines.

The table 4.5 presents the constant values of the parameters used for these runs.

CSO
$RN = 5$
$HN = 20$
$MN = 10$
$CN = 35$
$G\_Max = 350$

Table 4.5: Shifted Rastrigin function. CSO algorithm parameters for G analysis.

The analysis presented here verifies the conclusions from [42] and both table 4.6 and figure 4.4 confirms it. When $G = 1$, it means that in every generation population re-arrangement occurs and the best chickens become roosters. This means that every time the rooster starts searching from an initial position, the groups change in every generation as well, and a rooster cannot lead its group properly. So, the exploration ability reduces, and this causes the algorithm to eaily gets stuck in local optima.

On the other hand, when $G = 50$, which is quite large, only 7 re-arrangements occur over the course of 350 maximum generations. In this case, the algorithm allows the roosters to lead their groups for a significant amount of time. However, if an individual other than a rooster finds a better position and gets a better fitness value, it retains the position without becoming a rooster and thus cannot properly lead the group. It must wait until the next re-arrangement, which might occur quite late. As a result, the algorithm's performance can be heavily impacted at times. The metrics

	G = 1		G = 2		G = 5		G = 10		G = 20		G = 50	
Statistic	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**	**Evaluations**	**Value**
	20702	0.0519527	19984	0.000852488	19856	0.00156522	19850	0.0180468	20060	0.0087543	20078	0.0294945
	19243	0.00515743	19430	0.0202553	19570	0.00296417	19786	0.0688959	19791	0.0141988	19825	0.0040859
	20094	0.0827796	19753	0.00220689	19454	0.00000494	19683	0.00518246	19981	0.00400069	20258	0.0500479
	19727	0.0612729	19411	0.0000679273	20180	0.000447815	19785	0.00420632	20144	0.00112761	20157	0.173257
	20792	0.688652	20036	0.00389567	19531	0.0058246	19960	0.0632412	—	—	20107	0.014053
Mean	20111.60	0.17736	19722.80	0.00506	**19718.20**	**0.00256**	19812.80	0.03112	19994.00	0.00727	20085.00	0.05459
Median	20094.00	0.06127	19753.00	**0.00221**	**19570.00**	0.00296	19785.00	0.01805	19981.00	0.00400	20107.00	0.02949
Std Dev	648.44	0.27434	238.52	0.00768	260.58	**0.00200**	**109.00**	0.02941	142.99	0.00555	168.69	0.06486
Min	**19243.00**	0.00516	19411.00	0.00007	19454.00	**0.00000494**	19683.00	0.00421	**19791.00**	0.00113	**19825.00**	0.00409
Max	20792.00	0.68865	20036.00	0.02026	20180.00	**0.00582**	**19960.00**	0.06890	20144.00	0.01420	20258.00	0.17326

Table 4.6: Shifted Rastrigin function. Raw values and statistical summary for G = 1, 2, 5, 10, 20, and 50 after 5 runs. Bolded values denote best performance per metric.

results clearly indicate this, as the algorithm struggles somewhat in this setting.

Therefore, a compromise between these two extremes tends to allow the algorithm to perform better. In the examples presented, $G = 5$ shows a slight advantage, achieving the lowest fitness value and overall better metric results.

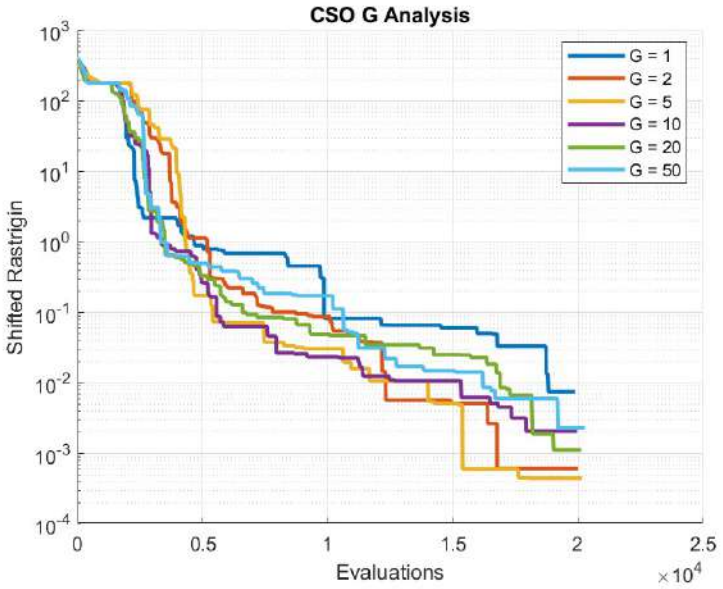

Figure 4.4: Shifted Rastrigin function. This figure presents comparison of mean values of six different G values ($G = 1, 2, 5, 10, 20, 50$) by using five independent runs with different seeds.

Also, for $G = 2$, $G = 10$, and $G = 20$, the results show quite good performance. However, the user of the algorithm must experiment to find the best setup for each specific problem. For the case examined, a value different from $G = 1$ or one that is too large could potentially provide better performance.

4.2 Pseudo-Engineering Problems

In this study, two pseudo-engineering design problems are selected to evaluate and compare the performance of the two algorithms. These benchmark problems are the Vibrating Platform Design and the Two-Bar Truss Design, both of which have been

sourced from the comprehensive suite introduced in [59]. These problems were chosen due to their relevance in structural optimization and their ability to reflect real-world constrained MOO scenarios, making them suitable for a meaningful performance assessment. In particular, two comparisons are conducted: one with EA and another one with MAEA.

For every run of each algorithm the fitnesss assignement mode is SPEA2. The number of elites requested is 50 for each run. Also, for each algorithm 5 independent runs with different initializations were conducted to ensure a more robust and meaningful comparison, not affected by stochasticity. All comparisons are performed under the same computational budget.

4.2.1 Vibrating Platform Design Problem

The Vibrating Platform Design problem is a constrained MOO problem used as a benchmark to evaluate the effectiveness of various evolutionary algorithms in solving complex real-world problems.

This problem was notably studied in [60], where has been proposed several enhancements to standard Multi-Objective Genetic Algorithms (MOGAs). However, in their study, the vibrating platform problem was used solely as a representative test case due to its inclusion of mixed-discrete variables, which introduces additional complexity to the optimization process.

In this work, the same problem helps comparing the performance of the proposed CSO algorithm against the EA and MAEA algorithms, and to evaluate how effectively CSO handles MOO with mixed-variable types. The Vibrating Platform Design is a constrained MOO problem that seeks to optimize the dynamic and cost performance of a mechanical structure. The problem consists of two objective functions that need to be minimized:

$$f_1 = -\frac{\pi}{2L^2}\sqrt{\frac{EI}{\mu}} \tag{4.3}$$

$$f_2 = 2bL\left(c_1 d_1 + c_2(d_2 - d_1) + c_3(d_3 - d_2)\right) \tag{4.4}$$

Objective function f_1 represents the inverse of the dynamic performance of the platform, where L is the length of the platform, EI is the moment of inertia, related to the platform's cross-sectional geometry and μ is the material's density per unit length.

Objective function f_2 represents the cost associated with the platform, which is proportional to its material and structural dimensions, where b is the breadth of the platform, L is the length of the platform, d_1, d_2, d_3 represent the different thicknesses of the platform at various segments and c_1, c_2, c_3 are cost coefficients that reflect the material or structural cost at different thickness intervals.
It, also, involves the following inequality constraints:

$$g_1 = \mu L - 2800 \leq 0 \tag{4.5}$$

$$g_2 = d_1 - d_2 \leq 0 \tag{4.6}$$

$$g_3 = d_2 - d_1 - 0.15 \leq 0 \tag{4.7}$$

$$g_4 = d_2 - d_3 \leq 0 \tag{4.8}$$

$$g_5 = d_3 - d_2 - 0.01 \leq 0 \tag{4.9}$$

The mass per unit length μ is given by:

$$\mu = 2b\left(\rho_1 d_1 + \rho_2(d_2 - d_1) + \rho_3(d_3 - d_2)\right) \tag{4.10}$$

The bending stiffness EI is given by:

$$EI = \frac{2b}{3}\left(E_1 d_1^3 + E_2(d_2^3 - d_1^3) + E_3(d_3^3 - d_2^3)\right) \tag{4.11}$$

The material properties are as follows:

$$\rho_1 = 100, \quad \rho_2 = 2770, \quad \rho_3 = 7780$$
$$E_1 = 1.6, \quad E_2 = 70, \quad E_3 = 200$$
$$c_1 = 500, \quad c_2 = 1500, \quad c_3 = 800$$

The design variables and their respective bounds are:

$$0.05 \leq d_1 \leq 0.5$$
$$0.2 \leq d_2 \leq 0.5$$
$$0.2 \leq d_3 \leq 0.6$$
$$0.35 \leq b \leq 0.5$$
$$3 \leq L \leq 6$$

To effectively study and solve the problem, it is necessary to define two threshold values for each constraint: the C^{THR}, as specified by the constraint equation itself, and a C^{D}, defined by the user. If an individual exceeds the C^{D}, a "death" penalty is applied. Every parameter value is denoted by the table 4.7.

CSO	**EA & MAEA**
$RN = 3$	$\mu = 20$
$HN = 32$	$\lambda = 90$
$MN = 32$	Crossover probability = 97%
$CN = 45$	Crossover Mode = 2-point /var
$G = 5$	Mutation probability = 5%
$G\_Max = 400$	Parents of one offspring = 3
	Tournament size = 3
	Tournament probability = 90%
	max_evaluations = 30000
	encoding = Binary-Gray

Table 4.7: Vibrating Platform Design Problem. CSO and EA /MAEA algorithm parameters.

The only way to control the maximum number of evaluations in CSO is by multiplying the total population size by the maximum number of generations, before the algorithm starts. In this case, the product yields 28000 evaluations. However, some evaluations are skipped because certain updated individuals coincide with previously evaluated ones stored in the database. When this occurs, their evaluation is bypassed, and the corresponding stored solutions are reused instead. As a result, for the run presented, the actual number of evaluations performed by CSO was around 25000. The number of total evaluations for EA remains at 30000 evaluations.

The results presented in Figure 4.5 are quite interesting and highlight the potential of CSO. The front of non-dominated solutions (computed with the same computational budget) obtained by CSO appears to be better distributed than that of EA. In particular, most of EA's solutions are quite spread, but they do not reach the edges and seem to be dominated by CSO's solutions. The objective function values for EA are primarily within the ranges of $[-12, -4] \times 10^{-3}$, and within $[250, 650]$ for f_2.

CSO, on the other hand, provides a clearer approximation of the Pareto front. Although the bottom-right corner of the front of non-dominated solutions appears more densely populated compared to other regions, the overall distribution of solutions is better and spans a wider area than that of EA. Moreover, dominance of the front of non-dominated solutions is clearly in favor of CSO. The number of total solutions after 5 runs are 107 for CSO and 32 for EA.

The results produced by HVI shows a well produced quality of CSO's front of non-dominated solutions comparing to that of EA. The HVI value of EA starts from very low values close to 0.25, and it is increased till the value 0.72. While CSO starts from the value of 0.3 till the value 0.75, in lesser evaluations. Both the Pareto front and the HVI obtained by CSO appears to be superior in terms of solution quality.

Two key factors contribute to these results more than any others. The first is the algorithm's design, which replaces penalized individuals with elite ones, adding random noise to their updated positions. This mechanism helps keep individuals within non-penalized regions for foraging, and is applied every G generations. As a result, the algorithm maintains a larger and more effective pool of individuals for discovering the front of non-dominated solutions (computed at the same computational budget).

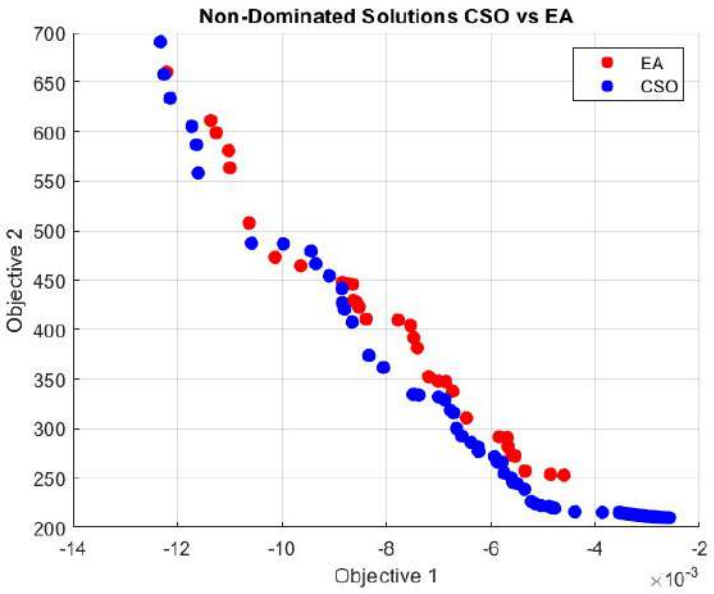

Figure 4.5: Vibrating Platform Design Problem. Front of non-dominated solutions (computed at the same computational budget) comparison CSO and EA. Both algorithms display the fronts of non-dominated solutions obtained from 5 separate runs with different initializations. The number of individuals on the Pareto is 107 for CSO and 32 for EA.

The second factor is the selection of the MN and G parameters. When MN is set equal to HN, the algorithm ensures that if all hens are penalized, a greater number of individuals can be replaced by elite ones. Regarding the choice of G, the value of 5 strikes a balance—it is neither too small nor too large. A smaller G could cause the algorithm to repeatedly reset into non-penalized zones, potentially limiting exploration and making it harder to discover better regions. On the other hand, a larger G might delay the algorithm's response to penalization, making it less effective at escaping poor or infeasible regions. Thus, the selected value of $G = 5$ allows the flock to forage and function as intended while maintaining adaptability.

As for the results produced by comparing CSO to MAEA, the outcomes show much more similarity. The settings used for the metamodel—listed in Table 4.8—are the default parameters recommended by EASY software.

With 30000 evaluations, MAEA is able to produce better results than in previous

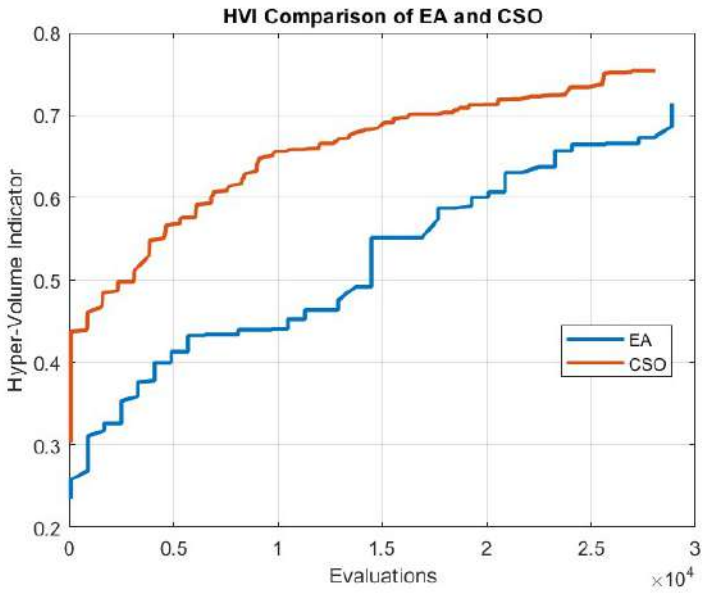

Figure 4.6: Vibrating Platform Design Problem. Convergence of CSO vs EA using Hyper-Volume Indicator. CSO starts from a higher value and converges to a higher value than EA.

Parameter	**Value**
Metamodel type	RBF
Exact evaluations (min)	10
Exact evaluations (max)	15
LCHE pause generations	5
Minimum DB entries	500
Not failed	50
Training patterns (min)	20
Training patterns (max)	40

Table 4.8: LCHE Settings.

runs in terms of dominance, as expected. By the end, the number of elite individuals reaches 42 in total, as shown in Figure 4.7. The solutions are now better distributed across the front of non-dominated solutions. CSO completes around 29000 mean evaluations and performs comparably to MAEA, but has more solutions (107). Notably, MAEA's solutions are, now, better distributed.

Figure 4.8 illustrates the convergence behavior of both algorithms. MAEA demon-

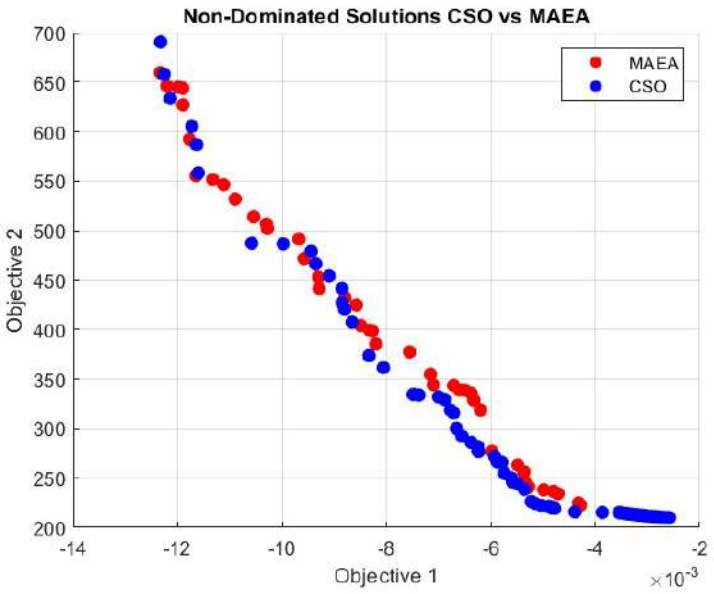

Figure 4.7: Vibrating Platform Design Problem. Front of non-dominated solutions (computed at the same computational budget) of CSO vs MAEA. Both algorithms display the fronts of non-dominated solutions obtained from 5 separate runs with different initializations. The number of individuals on the front of non-dominated solutions is 107 for CSO and 42 for MAEA.

strates faster convergence, as indicated by the lower HVI values. After 21000 evaluations, both algorithms appear to converge at a similar rate. However, CSO reaches an objective value of 0.75, while MAEA converges to a higher value of 0.85, in slightly more ecaluations. Overall, the HVI graph demonstrates similar performance of CSO and MAEA, which is fully consistent with the results observed in the front of non-dominated solutions analysis (Figure 4.7).

4.2.2 Two Bar Truss Design Problem

The two-bar truss is a type of truss structure consisting of two bars connected at the ends by either pins or joints. This type of truss is commonly used in construction and engineering applications. It is a simplified constrained MOO problem that aims to minimize both the volume (mass-related) and the stress of a simple truss structure. The problem consists of two objective functions to be minimized:

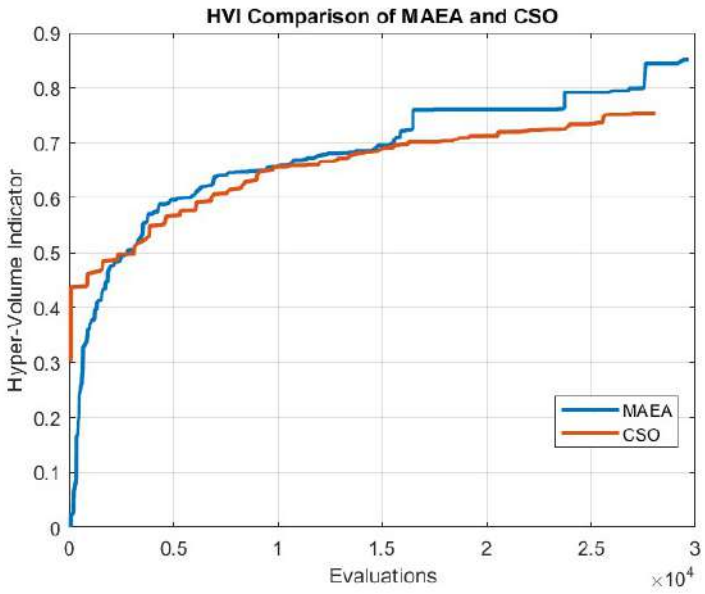

Figure 4.8: Vibrating Platform Design Problem. Convergence of CSO vs MAEA with RBFN using Hyper-Volume Indicator. CSO starts from a higher value and converges to a lower value than MAEA in fewer evaluations. MAEA reaches similarly high values as CSO, although it requires slightly more evaluations to do so.

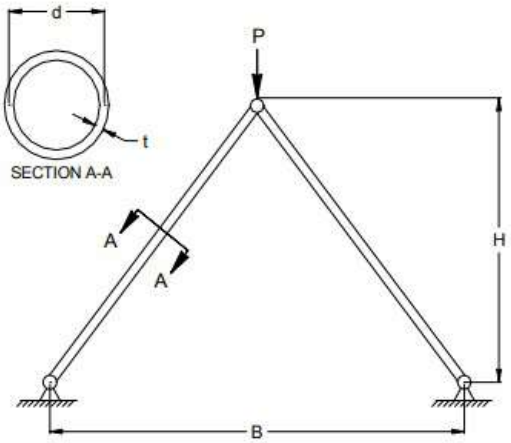

Figure 4.9: Schematic representation of the Two-Bar Truss Design problem [61].

$$f_1 = x_1\sqrt{16 + x_3^2} + x_2\sqrt{1 + x_3^2} \tag{4.12}$$

$$f_2 = \frac{20\sqrt{16 + x_3^2}}{x_3 x_1} \tag{4.13}$$

Objective function f_1 represents a performance or cost metric related to the geometry and load-bearing behavior of the truss, where x_1 is a design parameter typically related to material volume or size, x_2 is a geometric parameter affecting stiffness and deformation and x_3 is a scalar factor (e.g., shape coefficient or slenderness ratio).

Objective function f_2 reflects stress performance under load, inversely related to the product of a design variable and a shape /stiffness parameter. The square root and division imply inverse proportionality to stiffness and direct impact on stress.
The design must satisfy the following nonlinear inequality constraints:

$$g_1 = f_1 - 0.1 \leq 0 \tag{4.14}$$

$$g_2 = f_2 - 10^5 \leq 0 \tag{4.15}$$

$$g_3 = \frac{80\sqrt{1 + x_3^2}}{x_3 x_2} - 10^5 \leq 0 \tag{4.16}$$

ensuring that performance objectives do not exceed permissible limits for weight, stress, and deformation.
The variables x_1, x_2, x_3 are subject to the following bounds:

$$10^{-5} \leq x_1 \leq 100$$
$$10^{-5} \leq x_2 \leq 100$$
$$1 \leq x_3 \leq 3$$

reflecting feasible ranges for geometric and physical design parameters, maintaining the structural integrity and realism of the design space.

Two Bar Truss Design Problem with constraints is examined, now, as it is more realistic and representative of the challenges an engineer must address. To effectively study and solve the problem, it is necessary to define two threshold values for each

constraint again, as done before. Every parameter value is denoted by the table 4.9.

CSO	**EA & MAEA**
$RN = 3$	$\mu = 20$
$HN = 32$	$\lambda = 80$
$MN = 32$	Crossover probability = 97%
$CN = 45$	Crossover Mode = discrete
$G = 5$	Mutation probability = 5%
$G\_Max = 250$	Parents of one offspring = 2
	Tournament size = 3
	Tournament probability = 90%
	max_evaluations = 20000

Table 4.9: Two Bar Truss Problem. CSO and EA /MAEA algorithm parameters.

For this problem, less total evaluations than the Vibrating Platform Design problem (around 20000 were allowed). So the maximum selected number of evaluations from CSO is $80 \cdot 250 = 20000$ and for EA is 20000.

The results presented in Figure 4.10 demonstrate promising performance of the CSO algorithm; however, it does not perform as well as in the previously solved problem. The front of non-dominated solutions obtained by the EA appears to be better distributed compared to that of CSO. Specifically, the EA identified 138 solutions that are more widely spread within the same number of evaluations, whereas CSO produced only 88 solutions. In the region corresponding to f_1 values in the range $[0.01, 0.02]$, EA outperforms CSO, as CSO's corresponding f_2 values are higher and thus dominated by those found by EASY. In the left-most portion of the front of non-dominated solutions, CSO was unable to find as many solutions as EA.

In general, CSO provides a clearer approximation to EA regarding the front of non-dominated solutions. It is distributed quite well along the objective with small gaps on it. This could be considered as a proof the mechanism that helps keep individuals within non-penalized regions for foraging, can lead to a pretty good performance comparable to a much more tested algorithm as EA. This is a useful asset of the algorithm and its user.

The results obtained from the HVI analysis indicate that CSO produces a front of

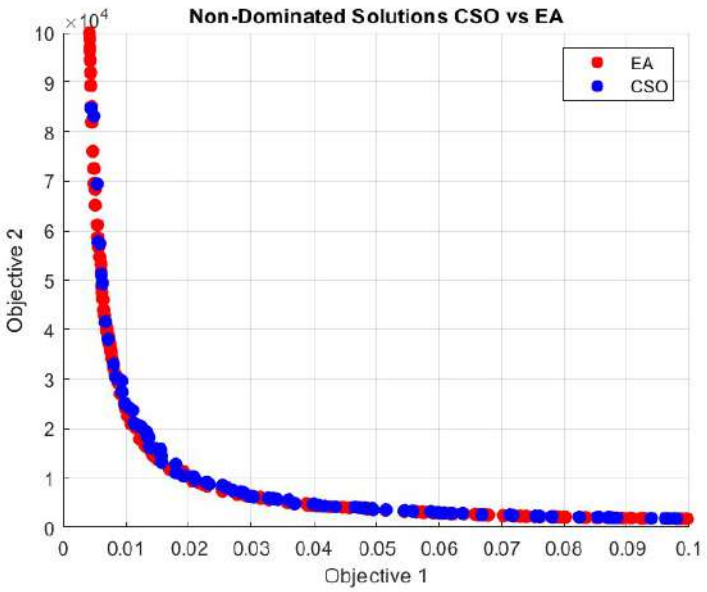

Figure 4.10: Two Bar Truss Design Problem. Front of non-dominated solutions (computed at the same computational budget) of CSO vs EA. EA has a wider front of non-dominated solutions solutions. The number of individuals on the Pareto is 88 for CSO and 138 for EA.

non-dominated solutions comparable to the EA. EA's HVI starts from relatively high values, close to 0, due to its the fact that in the first generations did not manage to find elite individuals. By the end, it managed to reduce the HVI value a bit higher than 0.9. However, it converges a bit slower, till 10000 evaluations, than CSO, which begins with a HVI of around 0, as it initially struggles, as well, to identify high-quality solutions. At the end, it reduces its HVI value to 0.9 faster than EA. As a conclusion, CSO performs better than EA in this case. After 10000 evaluations, both algorithms converge with similar rate.

As for the results produced by comparing CSO to MAEA, the outcomes are very similar. The settings used for the metamodel, listed in Table 4.8, are still the default parameters recommended by MAEA. The same conclusions can be drawn, as in previously described without the use of metamodel. The only change is that in Figure 4.12 MAEA gives 142 individuals instead of 138.

Figure 4.13 presents the behaviour of both algorithms. MAEA struggles to find

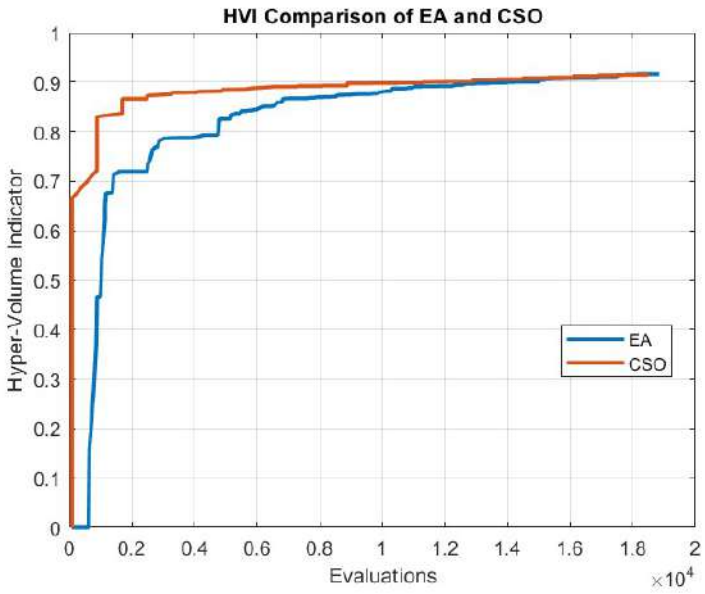

Figure 4.11: Two Bar Truss Design Problem. Convergence of CSO vs EA using Hyper-Volume Indicator. Both algorithms converge with similar rate to the same value.

solutions at the beggining as before. This is indicated by the HVI values which equal to 1. HVI decreases till the HVI value of around 0.1.The use of the RBF metamodel helped MAEA converge faster, with performance nearly matching that of CSO. However, its convergence is slower up to approximately 7000 evaluations. After, it slightly becomes better, and after 13000 evaluations they both converge with similar rate to the same vlaues.

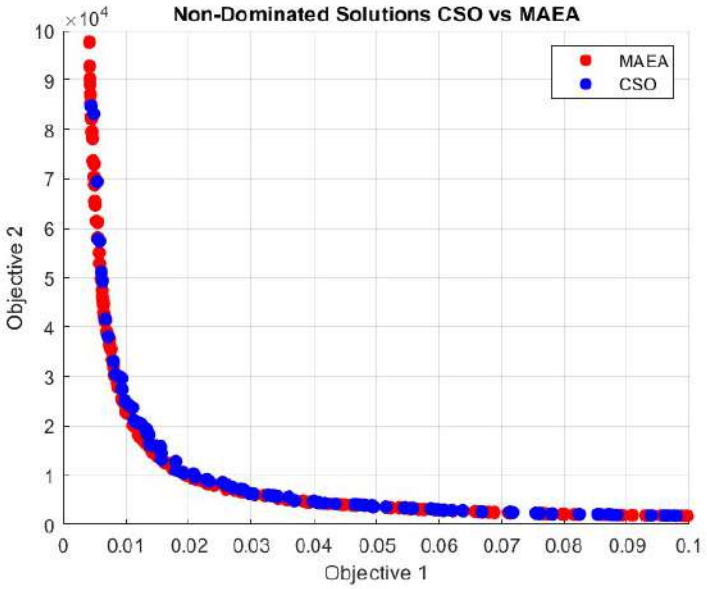

Figure 4.12: Two Bar Truss Design Problem. Fronts of non-dominated solutions (computed at the same computational budget) of CSO vs MAEA. MAEA has a wider front of non-dominated solutions. The number of non-dominated individuals is 88 for CSO and 142 for MAEA.

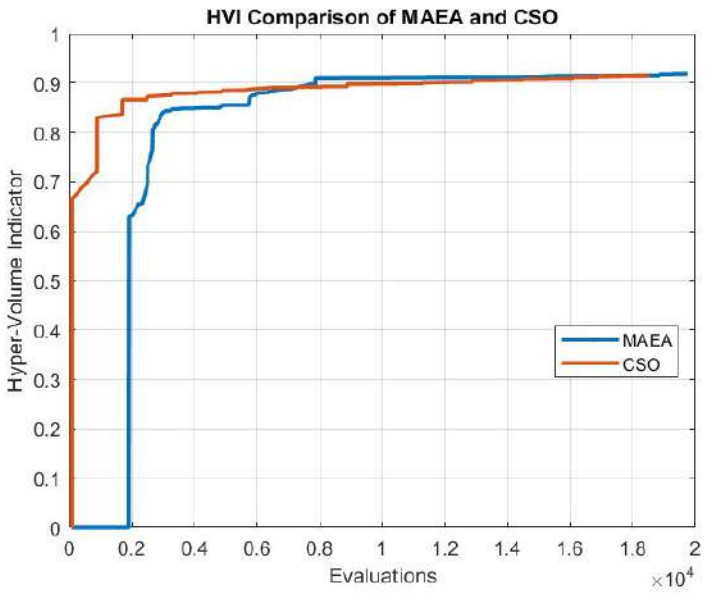

Figure 4.13: Two Bar Truss Design Problem. Convergence of CSO vs MAEA using Hyper-Volume Indicator. Both algorithms converge with similar rate to the same value.

Chapter 5

Conclusions

This thesis examined every aspect of the CSO algorithm as developed and implemented by the author. Prior to the detailed exploration of CSO, the fundamentals of stochastic optimization were introduced, alongside an overview of Swarm Intelligence and the PSO algorithm, since CSO falls within this broader category. Additionally, reference was made to the (μ, λ)-EA and its core principles, as it served as a benchmark for comparing the performance of CSO.

Regarding CSO itself, the algorithm's inspiration from the biological behavior of chicken flocks was analyzed. The structure and interaction among the three primary populations (roosters, hens, and chicks) were thoroughly described. Specifically, roosters lead their respective groups, hens follow their designated rooster, and chicks follow only their mother hen. A systematic rearrangement of population and regrouping of individuals is carried out every G generations.

The CSO was applied to the shifted Rastrigin function, where the global minimum lies at $(3, 3, \ldots, 3)$ instead of the origin $(0, 0, \ldots, 0)$, in order to increase the problem's complexity and challenge the optimization process. A thorough investigation was conducted into how the algorithm's parameters, RN, MN, and G, influence convergence behavior and final solution quality. To further evaluate the performance and robustness of the implementation, two pseudo-engineering optimization problems were also solved, the Vibrating Platform Design and the Two Truss Bar Design problems. Each problem was run both with and without the use of metamodels, integrated

through the EASY framework, in order to study how well can the CSO perform comparing to a surrogate-assisted algorithm. The constrained handling provided insight into the CSO's adaptability and performance in more practical and computationally demanding scenarios.

Regarding the analysis of CSO's parameters, it is evident that operating with only one rooster, and thus only one group, leads to inconsistent results and deviations in solution quality. Employing a greater number of roosters improves performance, as it reduces the risk of the algorithm being misled by a poorly performing rooster. Increasing the number of roosters allows for broader exploration, as the flock can forage more effectively across the search space. Moreover, this setup provides an opportunity for underperforming roosters to evolve into effective leaders for their respective groups. Figure 4.1, with $RN = 3, 5, 7, 9$, illustrates this behavior clearly.

For the number of mothers, no definitive conclusion can be drawn. When there is only one mother in the flock, if she follows a poorly performing rooster, she will likely lead her chicks to suboptimal regions as well. Conversely, if the rooster is a good leader, the chicks following their mother may produce better results. This results in a high variation in performance, as shown in Figure 4.3. Furthermore, when $MN = HN$, a large number of hens may follow a bad rooster, causing a significant portion of the chicks to forage inefficiently. Based on this, it is advisable to use a value of MN such that $1 < MN < HN$ for unconstrained problems. In contrast, for constrained optimization problems, it is recommended to use a large number of mothers, even $MN = HN$. This is because when many mother hens are penalized, the constraint-handling mechanism forces them to forage in non-penalized regions. A larger MN ensures that more hens, and by extension their chicks, are redirected toward feasible and potentially better regions of the search space.

Regarding the population rearrangement factor G, setting $G = 1$ causes the population to be rearranged at every generation. This frequent reshuffling can lead the algorithm to become easily trapped in local minima, as it prevents the defense mechanisms, such as the stochastic escape behavior of roosters or the random repositioning of chicks, from functioning effectively. On the other hand, when G is set too high, the algorithm's adaptability decreases, resulting in reduced performance. These observa-

tions are consistent with the findings reported in [42]. Therefore, it is recommended to test several values of G, avoiding values that are too small or too large, and tuning the parameter based on the specific characteristics of the problem at hand.

The CSO also demonstrated notable performance when compared to EA and MAEA. For the Vibrating Platform problem, CSO outperformed EA and MAEA, yielding more solutions with better distribution across the front of non-dominated solutions. The results are presented in Figures 4.5 and 4.7. For the Two-Bar Truss Design problem, CSO presented slightly worse results than EA and MAEA. The results are presented in Figures 4.10 and 4.12.

Overall, the CSO algorithm presented in this thesis was thoroughly evaluated and, based on the results, has proven to be a promising optimization approach. Naturally, further testing and validation on a wider range of problems are necessary to fully assess its robustness and general applicability. In some cases, CSO outperformed EA and MAEA, while in others, it performed comparably. Some of the novel ideas introduced in CSO, as it has a swarm-based behavior, are not straightforward to integrate into EASY due to fundamental differences in algorithmic structure. However, there is potential for EASY to benefit from the concepts employed in CSO, possibly through hybridization or algorithmic extensions.

Future Work

There are several ideas that were not explored by the author but could be considered for future development of the CSO algorithm. Given the promising performance of the constraint-handling mechanism used in MOO problems, a similar approach could be investigated for unconstrained problems to further enhance the algorithm's robustness. Additionally, there are instances where the software exhibits delays during execution. As the implementation is written in C++, performance optimization through multithreaded programming could be a valuable enhancement, for example, by assigning a separate thread to each group in the population.

Furthermore, alternative strategies for constraint handling could be explored. The current method is quite strict, and while a very loose approach did not yield strong

results, a balanced compromise between the two may lead to improved performance. Such developments could enhance both the efficiency and adaptability of the algorithm across a wider range of problem types.

In the context of MOO, the function Φ is used to rank and rearrange individuals within the population. According to the values of Φ, individuals are assigned behavioral roles such as roosters, hens, and chicks, following the described hierarchical model. However, this scalar metric Φ may not be the most suitable choice, as it does not take into account the relative positions of individuals on the Pareto front. Since Φ typically aggregates multiple objectives into a single value, it may obscure important trade-offs and overlook the distribution of solutions. A more geometric approach, one that incorporates inter-individual distances in the objective space, could offer a better means of differentiation. For example, metrics based on crowding distance or spatial proximity might help preserve diversity while enhancing convergence guidance.

Last but not least, it is worth noting that the CSO algorithm currently lacks a Metamodel-Assisted feature, a technique that has proven beneficial in many modern stochastic optimization algorithms by reducing computational cost and guiding the search more efficiently. Integrating such a mechanism into CSO could be a valuable direction for future research, potentially improving convergence speed and overall performance, especially in high-dimensional or computationally expensive problems such as CFD problems.

Bibliography

[1] S. S. Rao, *Engineering Optimization: Theory and Practice*, 4th. Hoboken, New Jersey: John Wiley & Sons, Inc., 2009.

[2] K. C. Giannakoglou, *Deterministic and stohastic optimization methods and applications*, NTUA, 2012.

[3] D. A. Van Veldhuizen and G. B. Lamont, «Multiobjective evolutionary algorithms: Analyzing the state-of-the-art», *Evolutionary Computation*, vol. 8, no. 2, pp. 125–147, 2000.

[4] K. M. Miettinen, *Nonlinear Multiobjective Optimization*. Jyvaäskylaä, Finland: Kluwer Academic Publishers, 1999.

[5] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, 2007. DOI: 10.1137/1.9780898717839.

[6] S. Kumar, «Finite difference method: A brief study», 2014, Available at SSRN: https://ssrn.com/abstract=2395968.

[7] L. V. Ahlfors, *Complex Analysis: An Introduction to the Theory of Analytic Functions of One Complex Variable*. McGraw-Hill, 1953.

[8] L. Hörmander, *An Introduction to Complex Analysis in Several Variables*. Van Nostrand, 1966.

[9] ScienceDirect Topics, *Directed differentiation - an overview — sciencedirect topics*, n.d. [Online]. Available: https://www.sciencedirect.com/topics/biochemistry-genetics-and-molecular-biology/directed-differentiation.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[11] L. D. Davis, Ed., *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.

[12] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs* (Artificial Intelligence Series). Berlin: Springer-Verlag, 1992.

[13] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Friedrich Frommann Verlag, 1973.

[14] H.-P. Schwefel, *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Technische Universität, Berlin, 1965.

[15] H.-P. Schwefel, «Collective phenomena in evolutionary systems», in *Problems of Constancy and Change – The Complementarity of Systems Approaches to Complexity*, P. Checkland and I. Kiss, Eds., Int'l Soc. for General System Research, 1987.

[16] H.-P. Schwefel, *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., 1993.

[17] D. V. Arnold, *Noisy Optimization with Evolution Strategies*. Springer, 2002, vol. 8.

[18] H.-G. Beyer, *The Theory of Evolution Strategies*. Springer, 2001.

[19] P. J. Angeline and K. E. Kinnear Jr, Eds., *Advances in Genetic Programming 2*. Cambridge, MA: MIT Press, 1996.

[20] W. Banzhaf *et al.*, Eds., *GECCO-99: Proc. Genetic and Evolutionary Computation Conference*. San Mateo, CA: Morgan Kaufmann, 1999.

[21] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[22] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1994.

[23] J. R. Koza, *Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program.* San Mateo, CA: Morgan Kaufmann, 1995.

[24] J. Kennedy and R. Eberhart, «Particle swarm optimization», in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Conference held 27 November 1995 - 01 December 1995., Perth, WA, Australia: IEEE, 1995, ISBN: 0-7803-2768-3. DOI: 10.1109/ICNN.1995.488968.

[25] X.-B. Meng, Y. Liu, X. Gao, and H. Zhang, «A new bio-inspired algorithm: Chicken swarm optimization», in *Lecture Notes in Computer Science: Proceedings of the International Conference in Swarm Intelligence*, Conference held in October 2014., Location of the conference, if known: Springer, 2014. DOI: 10.1007/978-3-319-11857-4_10.

[26] S. Alam, G. Dobbie, Y. S. Koh, P. Riddle, and S. U. Rehman, «Research on particle swarm optimization based clustering: A systematic review of literature and techniques», *Swarm and Evolutionary Computation*, vol. 17, pp. 1–13, 2014, Available online 17 February 2014. DOI: 10.1016/j.swevo.2014.02.001.

[27] M. Dorigo, V. Maniezzo, and A. Colorni, «Positive feedback as a search strategy», Dipartimento di Elettronica, Politecnico di Milano, Italy, Tech. Rep. 91-016, 1991.

[28] M. Dorigo, «Optimization, learning and natural algorithms (in italian)», Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.

[29] M. Dorigo, V. Maniezzo, and A. Colorni, «Ant system: Optimization by a colony of cooperating agents», *IEEE Transactions on Systems, Man, and Cybernetics—Part B*, vol. 26, no. 1, pp. 29–41, 1996.

[30] D. Karaboga, «An idea based on honey bee swarm for numerical optimization», Erciyes University, Technical Report TR06, 2005.

[31] H. A. A. Bahamish, R. Abdullah, and R. A. Salam, «Protein tertiary structure prediction using artificial bee colony algorithm», in *Proceedings of the Third Asia International Conference on Modelling & Simulation (AMS 2009)*, 2009.

[32] S. M. Tabatabaei and B. Vahidi, «Bacterial foraging solution based fuzzy logic decision for optimal capacitor allocation in radial distribution system», *Electric Power Systems Research*, vol. 81, no. 4, pp. 1045–1050, 2011. DOI: 10.1016/j.epsr.2010.12.002.

[33] H. Chen, B. Niu, L. Ma, W. Su, and Y. Zhu, «Bacterial colony foraging optimization», *Neurocomputing*, vol. 137, pp. 268–284, 2014. DOI: 10.1016/j.neucom.2013.04.054.

[34] M. S. Li, T. Y. Ji, W. J. Tang, Q. H. Wu, and J. R. Saunders, «Bacterial foraging algorithm with varying population», *Biosystems*, vol. 100, no. 3, pp. 185–197, 2010. DOI: 10.1016/j.biosystems.2010.03.003.

[35] I. Fister, I. Fister Jr., X.-S. Yang, and J. Brest, «A comprehensive review of firefly algorithms», *Swarm and Evolutionary Computation*, vol. 13, pp. 34–46, 2013. DOI: 10.1016/j.swevo.2013.06.001.

[36] A. Gandomi, X.-S. Yang, S. Talatahari, and A. Alavi, «Firefly algorithm with chaos», *Communications in Nonlinear Science and Numerical Simulation*, vol. 18, no. 1, pp. 89–98, 2013. DOI: 10.1016/j.cnsns.2012.06.009.

[37] A. H. Gandomi, X.-S. Yang, and A. H. Alavi, «Mixed variable structural optimization using firefly algorithm», *Computers & Structures*, vol. 89, no. 23–24, pp. 2325–2336, 2011. DOI: 10.1016/j.compstruc.2011.08.002.

[38] J. Holm and E. Botha, «Leap-frog is a robust algorithm for training neural networks», *Network: Computation in Neural Systems*, vol. 10, pp. 1–13, 1999.

[39] J. Snyman, «The lfopc leap-frog algorithm for constrained optimization», *Computers & Mathematics with Applications*, vol. 40, no. 8–9, pp. 1085–1096, 2000. DOI: 10.1016/S0898-1221(00)85018-X.

[40] X.-S. Yang, «A new metaheuristic bat-inspired algorithm», in *Studies in Computational Intelligence*, vol. 284, Springer, 2010, pp. 65–74. DOI: 10.1007/978-3-642-12538-6_6.

[41] X.-S. Yang, «Chapter 10 - bat algorithms», in *Nature-Inspired Optimization Algorithms*. Elsevier, 2014, pp. 141–154. DOI: 10.1016/B978-0-12-416743-8.00010-5.

[42] B. Chen, L. Cao, C. Chen, Y. Chen, and Y. Yue, «A comprehensive survey on the chicken swarm optimization algorithm and its applications: State-of-the-art and research challenges», *Artificial Intelligence Review*, vol. 57, p. 170, 2024, Accepted: 5 May 2024 / Published online: 11 June 2024. DOI: 10.1007/s10462-024-10786-3.

[43] L. J. Fogel, *Evolutionary programming*, Foundational work in Evolutionary Computation, San Diego, CA, USA, 1962.

[44] J. H. Holland, *Genetic algorithms*, Early development of Genetic Algorithms, Ann Arbor, MI, USA, 1962.

[45] I. Rechenberg and H.-P. Schwefel, *Evolution strategies*, Pioneering work in Evolution Strategies, Berlin, Germany, 1965.

[46] K. Jebari, M. Madiafi, and A. Elmoujahid, «Parent selection operators for genetic algorithms», *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 11, 2013, ISSN: 2278-0181. [Online]. Available: https://www.ijert.org/parent-selection-operators-for-genetic-algorithms.

[47] Baeldung, *Roulette wheel selection in genetic algorithms*, https://www.baeldung.com/cs/genetic-algorithms-roulette-selection, 2023.

[48] Wikipedia contributors, *Stochastic universal sampling*, https://en.wikipedia.org/wiki/Stochastic_universal_sampling, Accessed: 2025-04-30, 2024.

[49] T. Blickle and L. Thiele, «A comparison of selection schemes used in genetic algorithms», Computer Engineering and Communication Networks Lab TIK, Swiss Federal Institute of Technology ETH, Zurich, Switzerland, Tech. Rep. TIK Report Nr. 1996, Accessed: 2025-04-30.

[50] Wikipedia contributors, *Tournament selection*, https://en.wikipedia.org/wiki/Tournament_selection, 2024.

[51] K. Deb, A. Pratap, S. Agarwal, and T Meyarivan, «A fast and elitist multiobjective genetic algorithm: Nsga-ii», *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[52] E. Zitzler and L. Thiele, «Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach», *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[53] E. Zitzler, M. Laumanns, and L. Thiele, «Spea2: Improving the strength pareto evolutionary algorithm», ETH Zurich, Computer Engineering and Networks Laboratory (TIK), Tech. Rep. 103, 2001.

[54] K. C. Giannakoglou, V. G. Asouti, and D. Kapsoulis, *Low-cost evolutionary algorithms for engineering applications*, Parallel CFD & Optimisation Unit, Laboratory of Thermal Turbomachines, School of Mechanical Engineering, National Technical University of Athens, 2020.

[55] I. C. Kampolis, A. S. Zymaris, V. G. Asouti, and K. C. Giannakoglou, «Multilevel optimization strategies based on metamodel-assisted evolutionary algorithms, for computationally expensive problems», in *Proceedings of the Congress on Evolutionary Computation (CEC)*, Singapore: IEEE, 2007.

[56] K. C. Giannakoglou and M. K. Karakasis, «Evolutionary algorithms with surrogate modeling for computationally expensive optimization problems», in *Proceedings of the Design Optimization International Conference (ERCOFTAC 2006)*, Spain, 2006.

[57] M. K. Karakasis and K. C. Giannakoglou, «Metamodel-assisted multi-objective evolutionary optimization», in *EUROGEN 2005, Evolutionary and Deterministic Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems*, Munich, 2005.

[58] J. Liang, L. Wang, and M. Ma, «An improved chicken swarm optimization algorithm for solving multimodal optimization problems», *Mathematical Problems in Engineering*, vol. 2022, pp. 1–18, 2022. DOI: 10.1155/2022/5359732. [Online]. Available: https://doi.org/10.1155/2022/5359732.

[59] A. Kumar *et al.*, «A benchmark-suite of real-world constrained multi-objective optimization problems and some baseline results», *Expert Systems with Applications*, vol. 213, p. 118 978, 2023. DOI: 10.1016/j.eswa.2022.118978.

[60] S. Narayanan and S. Azarm, «On improving multiobjective genetic algorithms for design optimization», *Structural Optimization*, vol. 18, pp. 146–155, 1999.

[61] A. O. Suite, *Me 575 - two-bar truss design problem*, https://apmonitor.com/me575/uploads/Main/twobar.pdf, Accessed April 11, 2025, 2024.

[62] S. Sanyal, *An introduction to particle swarm optimization algorithm*, https://www.analyticsvidhya.com/blog/2021/10/an-introduction-to-particle-swarm-optimization-algorithm/, 2021.

[63] GeeksforGeeks, *Particle swarm optimization (pso) – an overview*, https://www.geeksforgeeks.org/particle-swarm-optimization-pso-an-overview/, Accessed: 2025-02-09.

[64] Wikipedia contributors, *Rastrigin function*, [Online; accessed DATE], n.d. [Online]. Available: https://en.wikipedia.org/wiki/Rastrigin_function.

Appendices

5.1 Appendix A: Input file

All input parameters are kept in one input file by the name, Data.txt which is located within the same directory as the program. The file is read by the algorithm at the time of its initialization. An example of expected format for Data.txt follows:

Number of roosters 4
Number of hens 25
Number of mothers 10
Number of chicks 46
G 2
max_generations 50
0 10
0 10
1 5
0 6
1 5
0 10

5.2 Appendix B: CSO - Evaluation Software Connection

While the function for evaluating individuals is running, the algorithm generates an ASCII file called task.dat. This file contains the values of the design variables in one column and stores the corresponding data. The evaluation software is then called to perform the calculations for each individual iteratively. Specifically, a loop begins, iterating through each chicken separately. For each chicken, the task.dat file is written, and the evaluation of the chicken is performed. The evaluation software reads the task.dat file for that chicken and generates two more ASCII files: task.res, which stores the calculated objectives of the chicken, and task.cns, which holds the values of the constraints, if any exist. Afterward, control returns to the CSO, which reads the task.res and task.cns files and stores their values. This process continues until all chickens have been evaluated.