Homework 5: Artificial Bee Colony

Alberto Guerrini

May 30, 2025

1 Introduction

In the realm of computational optimization, many real-world problems are characterized by features like non-convexity, non-linearity, the presence of constraints, making traditional deterministic optimization methods less effective or resource intensive.

Stochastic optimization algorithms, which incorporate randomness into the search process, have emerged as powerful tools for tackling such challenging optimization problems. These algorithms are particularly adept at escaping local optima and exploring the solution space more broadly, which is essential for finding global optima in complex landscapes. A broad subgroup of such algorithms comprises iterative and population-based methods. While iterative identify that the solution is found in multiple steps to evolve the state, population-based refers to an algorithm working with a set of solutions and trying to improve them.

This homework focuses on a method that models the specific intelligent behaviours of honey bee swarms, namely the Artificial Bee Colony (ABC) algorithm, first proposed by Karaboga and Basturk [2]. Furthermore, since constraints are encountered in numerous optimization applications ranging from physics and engineering to economics and management, their handling is a key feature in modern optimization techniques. This homework thus extends the ABC algorithm to solve Constrained Optimization Problems (COPs) as proposed in [3].

2 Background

This section defines the problem of optimization and how it is tackled in the stochastic optimization realm, specifically with the method of our choice. We first provide the formalization of the problem both without and with constraints, then we provide its modelling to be then solved using the ABC method.

2.1 Problem

An Optimization Problem (OP) aims to find the optimal solution $\boldsymbol{x} \in \mathbb{R}^n$ that minimizes (maximizes) an objective function $f : \mathcal{D} \to \mathbb{R}$, where $\mathcal{D}$ is the domain (also called search space) on which the objective function is defined and s.t. $\mathcal{D} \subseteq \mathbb{R}^n$.

When introducing constraints, thus extending this problem to a COP, we formally require the optimal solution to be in a feasible region $\mathcal{F}$, i.e. $\boldsymbol{x} \in \mathcal{F} \subseteq \mathcal{S}$.

2.2 Model

In the field of mathematical optimization, OPs are modelled with the assumption that the domain $\mathcal{D}$ is defined as a n-dimensional rectangle, thus, it can be characterized by a lower bound and upper bound vectors $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$. This assumption eases the handling of candidate solutions, especially in population-based methods where random vectors have to be drawn within the search space $\mathcal{D}$.

Thus, a generic OP can be formulated as:

$$\begin{aligned} \text{minimize} \quad & f(x), \quad x = (x_1, \ldots, x_n) \in \mathbb{R}^n \\ \text{in} \quad & l_i \leq x_i \leq u_i, \quad \forall i \in [1, n]. \end{aligned} \tag{1}$$

In the case of COPs, an additional assumption is usually made, without loss of generality, to efficiently model the feasible region. The latter is defined by a set of m additional constraints, where each can be either an inequality $g(\boldsymbol{x}) \leq 0$, or equality $h(\boldsymbol{x}) = 0$ constraint. Given the above considerations, the problem in (1) is extended to a COP as:

$$\begin{aligned} \text{minimize} \quad & f(x), \quad x = (x_1, \ldots, x_n) \in \mathbb{R}^n \\ \text{in} \quad & l_i \leq x_i \leq u_i, \quad \forall i \in [1, n] \\ \text{subject to} \quad & g_j(\boldsymbol{x}) \leq 0, \quad \text{for } j = 1, \ldots, q \\ & h_j(\boldsymbol{x}) = 0, \quad \text{for } j = q+1, \ldots, m. \end{aligned} \tag{2}$$

When handling constraints, the considered COP is reformulated so as to take the form of optimizing two functions, the objective function $f(\boldsymbol{x})$ and the constraint violation function $c : \mathcal{D} \to \mathbb{R}$, defined as

$$c(\boldsymbol{x}) = \sum_{j=1}^{q} \max(0, g_j(\boldsymbol{x})) + \sum_{j=q+1}^{m} |h_j(\boldsymbol{x})|. \tag{3}$$

2.3 Method

In this work, the ABC algorithm is the method of choice to find a solution for the problem in (2). The ABC algorithm simulates the intelligent foraging behaviour of a honey bee swarm. The algorithm consists of three types of bees: *employed bees*, *onlooker bees*, and *scout bees*, which work together to find the optimal solution.

At each iteration (cycle), each *employed bee* produces a modification on the position in her memory with probability MR (modification rate): bee i selects a new random position x_i^{new} based on a randomly chosen bee k with the position update expression: $x_{i,j}^{new} = x_{i,j} + \text{rand}(0,1)(x_{i,j} - x_{k,j})$, where $j \in [1, n]$ is randomly chosen. This step can be formalized as:

$$x_{ij}^{new} = \begin{cases} x_{ij} + \beta_{ij}(x_{ij} - x_{kj}), & \text{if } R_j < MR \\ x_{ij}, & \text{otherwise.} \end{cases}$$

The bee's position is then updated based on a Greedy rule: if the fitness in the new position $f(\boldsymbol{x_i^{new}})$ is higher than the current fitness, the bee moves.

Subsequently, each *onlooker bee i* chooses a solution from the candidate solutions from the *employed bee* phase, based on their fitness. The probability of selecting the candidate position $\boldsymbol{x_j}$ is given by $p_j = \frac{f(\boldsymbol{x_j})}{\sum f(\boldsymbol{x_j})}$. As in the previous phase, the bee's position $\boldsymbol{x_i}$ is then updated with the selected $\boldsymbol{x_j}$ based on the Greedy rule.

Finally, the *scout bees* handle bees subject to starvation. If a solution x_i is not updated for ACN (abandonment cycle number) steps, it is considered as abandoned. Each abandoned solution is re-initialized in the initial domain $\mathcal{D}$. This process repeats until a stopping criterion is met. In this implementation, the loop stops after MCN (maximum cycle number) iterations.

The ABC algorithm balances exploration (global search) and exploitation (local search) through the different behaviour among bees' types, and the cooperative behaviour within the swarm. It dynamically adapts the search process based on feedback from the environment: the pairwise distance between two bees decreases as iteration count increases, thus the step size is adaptively reduced, improving its ability to find optimal solutions.

The version implemented in this project extends the above algorithm to solve COPs by replacing the Greedy selection method with the Deb's rule [1]. The latter is a technique to select the best solution, by applying an adaptive relaxation of constraints in a tournament of pairwise selections. Given two candidate solutions, it consists of three steps:

1. A feasible solution is preferred to an infeasible solution.
2. Among two feasible solutions, the one having better objective function value is preferred.
3. Among two infeasible solutions, the one having smaller constraint violation is preferred.

This technique is preferred to the common penalty rule as it does not require additional parameters, thereby preserving the self-adaptive property of the ABC algorithm and not requiring knowledge to tune such parameter.

The pseudocode of the ABC algorithm for COPs solution is proposed in Algorithm 1.

3 Implementation Details

In this section we provide an overview on our implementation of the ABC algorithm for COPs solution, highlighting relevant implementation choices and

Algorithm 1 Pseudo-code of the ABC algorithm for COPs

```
Require: $SN$ population size, $MR$ modification rate, $MCN$ max cycle number, $ACN$ abandonment cycle number
 1: Initialize the population of solutions $\boldsymbol{x_i} \in \mathbb{R}^n,\ i = 1 \dots SN$
 2: Evaluate the population's fitness and total constraint violation
 3: cycle = 1, count$_i$ = 0 $\forall i$
 4: repeat
 5:     Employed bees phase ($\forall i$ with probability $MR$):
 6:         Produce new solutions $x_{i,j}^{new} = x_{i,j} + \text{rand}(0,1)(x_{i,j} - x_{k,j})\ \forall i$
              $k \in [1, SN],\ j \in [1, n]$ randomly selected.
 7:         Evaluate the bees' fitness and total constraint violation ($\forall i$)
 8:         Apply selection process based on Deb's method ($\forall i$)
 9:     Onlooker Bees Phase ($\forall i$):
10:         Calculate probabilities $p_{i,j} = \frac{f(\boldsymbol{x_i})}{\sum_{n=1}^{SN} f(\boldsymbol{x_n})}$
11:         Draw the new solutions $x_{i,j}^{new}$ from the current solutions $x_{i,j}$,
              depending on $p_{i,j}$
12:         Apply selection process based on Deb's method ($\forall i$)
13:         Increment count if solution is not improved, reset count otherwise
14:     Scout bees phase ($\forall i$ if count$_i \geq ACN$):
15:         Re-initialize the abandoned solution with a random solution $x_{i,j}^{new} \in \mathcal{D}$
16:     Memorize the best solution achieved so far
17:     cycle = cycle + 1
18: until cycle = $MCN$
```

details on the parallelization of the algorithm.

The presented algorithm has been implemented with the C++17 programming language, leveraging the `Eigen 3.3` linear algebra library for the low-level handling of n-dimensional vectors. Templates are used to define at compile-time the dimensionality of the search space, allowing the resource-efficient static storage of vectors instead of dynamic heap-based memory allocation.

The parallelization is implemented both at thread-level where it is based on `OpenMP 3.1` APIs and at process-level by the `MPI 3.0` APIs. The parallel implementation is based on the idea that each particle can handle its update with minimal (or null in some cases) interaction with other particles. Each thread or process handles the update of a statically-assigned subgroup of the SN bees, sharing only the essential information, consisting of the bees' position at the end of the *employed bee* phase. Parallel-reduction operations are implemented to extract the best solutions at the end of each cycle. We note that synchronization among threads/processes is only required before such information sharing and reduction, minimizing the non-negligible overhead associated with synchronization constructs.

Problem	Optimal	Best	Mean	Worst	Std. Dev.
Gomez-Levy	-1.0316	-1.0316	-1.0316	-1.0316	0
Townsend	-2.0239	-2.0237	-1.993	-1.8760	0.0278
Griewank	0	0	0	0	0
G7	24.306	24.449	24.5316	24.666	0.0529
G10	7049.2	7106.6	7284.8	7464.6	23.228

Table 1: Optimization results for 5 test problems commonly used in the literature, over 20 independent runs each with 80 particles and 8000 cycles.

4 Experimental Evaluation

In this section we present our findings on the evaluation of the proposed implementation. First, the correctness of the optimizer is evaluated through its application on different test functions commonly proposed in the literature. We assess its convergence to the actual solution and the correct behaviour of the constraint handling technique. Then we analyse the performances of the parallel implementation through scaling and speed-up studies.

The experimental setup for all the tests consists of a local machine with an Intel Core i7-13700H CPU (20 logical threads and 8 performance + 4 power-efficient physical cores) and a 16GB RAM.

Optimization error The error and constraint violation are represented as function of the iteration count in Figure 1 to assess the correct behaviour of the optimization procedure. These results are based on the optimization of the G7 test problem in a 10-dimensional space, with a colony of $SN = 60$ bees.

We observe that the initial solutions are characterized by an high violation, as Deb's rule implicitly relaxes constraints, and bad fitness as expected from the random initialization of bees' position. Then, as a consequence to the Deb's rule, the algorithm first tries to minimize the constraint violation disregarding the relative error, until it reaches a potential solution with null constraint violation (achieved around iteration 10 in Fig. 1). In these first iterations the constraint violation is decreasing, while the relative error might not. Once it finds a feasible solution, infeasible solutions are no longer accepted even if they have a better fitness. This is clearly represented by the constraint violation remaining null after iteration 10. In this phase, however the error is strictly decreasing and converging to 0, as seen in the top plot in Fig. 1. The total number of feasible particles is increasing from 0 until convergence to the total number of bees.

Further testing has been carried out on a total of 5 test problems for constrained and unconstrained optimization commonly used in the literature and defined in search spaces of dimensionality from 2 to 30. Results are provided in Table 1, confirming the correctness of the implementation and the versatility of the proposed algorithm.

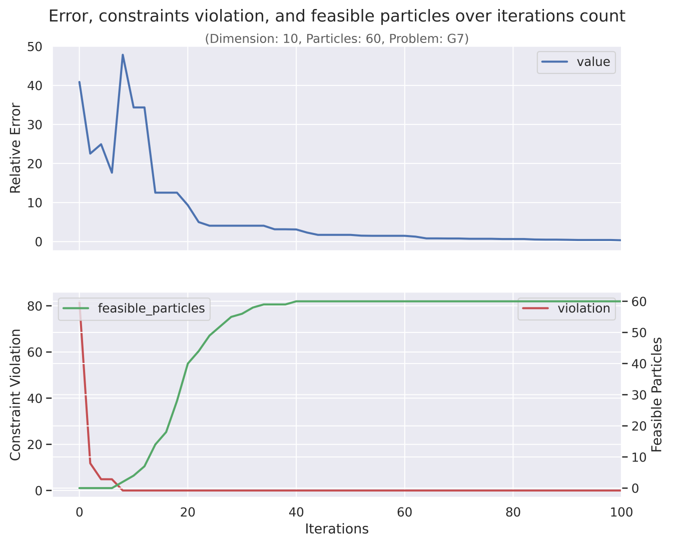
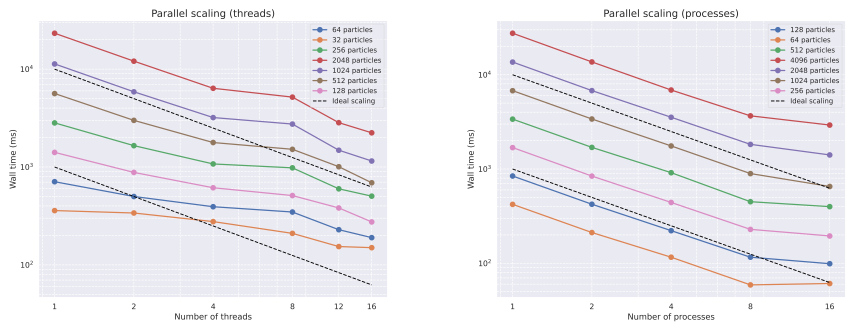


Figure 1: Optimization of the G7 function with the implemented ABC solver. Above: relative error from the optimal solution at each iteration; Below: constraint violation of the best solution and number of feasible particles at each iteration.

Scalability In order to compare how each of our algorithms scale, timing data has been collected varying the number of threads (and processes when possible) and the problem size. Scaling experiments have been performed solving the heavy G10 problem (8-dimensional, 6 constraints) in obtain reliable results. Figure 2 reports the scalability for the thread-level and process-level parallel implementation. Tests are executed varying both the problem size SN and the number of threads(processes). Along the presented lines problem size is constant. When doubling the number of threads(processes), one expects a halving of the computational time, indicated by the black dashed ideal scaling lines. This behaviour is known as strong scaling.

It is possible to evaluate the weak scaling behaviour from Figure 2 by considering that each line differs by a factor of 2 in the number of particles and that an ideal weak scaling is obtained when total time remains constant when doubling both the number of particles and the number of threads(processes).

The ABC algorithm benefits significantly from parallel execution, achieving substantial reductions in execution time. Optimal strong scaling is almost achieved for all problem sizes in process-level parallelism, while for the thread-level parallelism the implementation suffers for smaller problem sizes, i.e. below $SN = 512$ bees. Similarly, for lower number of threads the scaling is closer to the ideal one compared to higher number of threads. This is likely due to synchronization overhead and shared-memory management that involves higher



(a) (b)

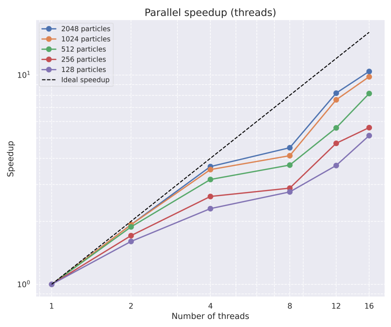
Figure 2: Execution time as a function of the number of (a) threads and (b) processes (b) to evaluate the scaling of the (a) OMP thread-level and (b) MPI process-level parallelization.

resource consumption with higher thread count. Both thread-level and process-level parallelism show a performance degradation after 12 threads(processes), i.e. the number of physical cores. Thus, a suboptimal behaviour is seen especially for smaller population sizes and for higher number of threads (bottom and right hand side areas) as expected. The synchronization overhead is not negligible and has a stronger impact when execution time is quite low (i.e. less than 1 second). MPI process-level parallelization maintains closer adherence to the ideal scaling line across various problem sizes.

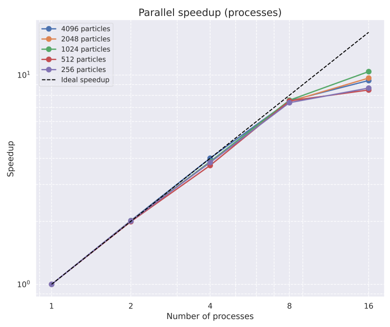
Parallel Speedup The parallel speedup for the execution on n threads or processes is defined as $s_n = \frac{t_1}{t_n}$, where t_k is the execution time with i threads(processes). We plot the speedup as a function of the number of threads(processes) in Figure 3. We observe that the implemented algorithm achieves a maximum speedup of around 10 with 16 threads and $SN = 2048$ bees. Thread-level speedup shows a degradation after 4 threads, while process-level parallelization achieves an optimal speedup only limited by hardware constraints (i.e. 12 physical cores). Results are coherent with the parallel scalability study proposed above.

5 Conclusion

This project successfully provides a correct and complete implementation for the ABC algorithm that is extended to target the optimization of both unconstrained and constrained problems. A parallel implementation has been developed as well showing close-to-optimal speedup and scalability. The algorithm performs well on common optimization problems, providing solutions with low variance and close to the optimum.



(a)



(b)

Figure 3: Speedup as function of the number of (a) threads and (b) processes (b).

References

[1] Kalyanmoy Deb. "An efficient constraint handling method for genetic algorithms". In: *Computer Methods in Applied Mechanics and Engineering* 186.2 (2000), pp. 311–338. ISSN: 0045-7825. DOI: `https://doi.org/10.1016/S0045-7825(99)00389-8`.

[2] Dervis Karaboga and Bahriye Basturk. "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm". In: *Journal of global optimization* 39.3 (2007), pp. 459–471.

[3] Dervis Karaboga and Bahriye Basturk. "Artificial Bee Colony (ABC) Optimization Algorithm for Solving Constrained Optimization Problems". In: *Foundations of Fuzzy Logic and Soft Computing*. Ed. by Patricia Melin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 789–798. ISBN: 978-3-540-72950-1.