A Particle Swarm Optimization on JGEA: implementation and comparison against other optimization techniques on different problems

Samuele Lippolis

November 2023

Contents

1 Project description

One of the possible exam project of the course Global and Multi-object Optimization (GMO) is applying a discussed algorithm to an existing problem. I chose to apply a standard version of Particle Swarm Optimization (PSO). It is implemented to be able to face problems of different dimensions. Moreover, it exploits the advantages of the parallel computation. I implemented PSO in Java within the Java General Evolutionary Algorithm (JGEA) framework [3]. It is a modular Java framework for experimenting with Evolutionary Computation.

Once I understood the JGEA structure, I was able to implement PSO as an extension of the previous existing structure in the framework. So that, I was able to benefit from parallelism automatically. Furthermore, I declare that the professor Eric Medvet helped me to understand the JGEA structure and to implement the code. For this reason, in the report I used both the first person singular and the first person plural. When I use the plural, it means that I did that stuff together with Eric Medvet.

2 Particle Swarm Optimization

2.1 Overview of the PSO algorithm

PSO is a stochastic optimization technique. It is an optimization technique, in fact we have a fitness function and our goal is finding its argmin. In particular, in the PSO case, the fitness function is defined as

$$f : \mathbb{R}^p \to \mathbb{R}$$

and the goal is getting the

$$r_{\min} \in \mathbb{R}^p \text{ s.t. } r_{\min} = \arg\min_{r \in \mathbb{R}^p} f(r).$$

It is stochastic since there is a random component within the procedure of evolution from one state of the algorithm to the next.

The PSO algorithm works similarly to an evolutionary algorithm. It means that it is an iterative method that works on a multi-set of evolving solutions. The searching space is $\mathbb{R}^p$.

In the PSO notation, there is a swarm composed by particles (instead of having a population and individuals like the classical evolutionary algorithms). A particle (e.g., the i^{th}) has 3 attributes:

1. A position $x_i \in \mathbb{R}^p$
2. A velocity $v_i \in \mathbb{R}^p$
3. The fittest position found so far by $x_i : x_{i,\text{best}} \in \mathbb{R}^p$

The state of the algorithm, for each generation, is given by

1. The current swarm position and velocity
2. For each particle $x_i \in$ pop, the fittest position found so far by the particle: $x_i^* \in \mathbb{R}^p$
3. For each informants group l, the fittest position found so far by all the informants: x_l^+
4. The fittest position found so far by the entire swarm: $x^!$

The PSO algorithm has 6 parameters:

- The size of the swarm: n_{pop}
- $w =$ proportion of velocity to be retained
- $\phi_{\text{particle}} =$ proportion of personal best to be retained (called *cognitive factor*)
- $\phi_{\text{informants}} =$ proportion of the informants' best to be retained
- $\phi_{\text{swarm}} =$ proportion of global best to be retained (called *social factor*)
- $\epsilon =$ jump size of a particle

and 3 random variables, that changes for each generation, for each component:

1. $r_{\text{particle}} \sim U([0,1]^p)$: random damping factor of the personal term
2. $r_{\text{informants}} \sim U([0,1]^p)$: random damping factor of the informants term
3. $r_{\text{swarm}} \sim U([0,1]^p)$: random damping factor of the swarm term

The procedure works as follow:

Algorithm 1 Particle Swarm Optimization (PSO)

```
1: Initialize swarm of particles with random positions and velocities
2: for each particle do
3:     Initialize particle's position randomly within the search space
4:     Initialize particle's velocity randomly
5:     Evaluate the fitness of the particle
6:     if particle's fitness is better than its personal best then
7:         Update personal best to current position
8:     end if
9: end for
10: Set the global best to the best particle in the swarm
11: Set the inertia weight ($w$)
12: Set other coefficients (the $\phi$ s)
13: while not converged or maximum iterations not reached do
14:     for each particle do
15:         Update velocity and position using the following equations:
16:         velocity = $w \times$ velocity + $\phi_{\text{particle}} \times r_{\text{particle}} \times$ (personal_best − current_position) + $\phi_{\text{informants}} \times r_{\text{informants}} \times$ (informants_best − current_position) + $\phi_{\text{swarm}} \times r_{\text{swarm}} \times$ (global_best − current_position)
17:         position = current_position + $\epsilon *$ velocity
18:         Evaluate the fitness of the particle at the new position
19:         if new position is better than personal best then
20:             Update personal best to the new position
21:         end if
22:         if new position is better than global best then
23:             Update global best to the new position
24:         end if
25:     end for
26: end while
27: return the fittest individual
```

2.2 Our PSO algorithm choice

In our implementation we chose a simpler version where

$$\phi_{\text{informants}} = 0,$$

$$\epsilon = 1.$$

For this reason, the velocity update is

$$v(t+1) = w*v(t)+r_{\text{particle}}*\phi_{\text{particle}}*(x(t)-x_{\text{particle best}})+r_{\text{swarm}}*\phi_{\text{swarm}}*(x(t)-x_{\text{swarm best}})$$

for each single component of the velocity of each particle. The first term represents the inertia so the tendency to maintain its own motion. The second term is about the tendency to follow the fittest known position by the single particle, while the last is about the tendency to follow the best known position by the swarm. The update of the velocity is simply:

$$x(t+1) = x(t) + v(t).$$

2.3 PSO parameters

A great method to solve an optimization problem is the Gradient descent. However, there are some situations where the gradient is too laborious or even impossible to derive. Then, PSO overcomes this problem and works independently from the gradient. Nevertheless, a price must be paid. Following the observations made in the paper [4], the PSO algorithm relays on many parameters and their choice has a large impact of the optimization performance. Unfortunately, the setting for PSO is problem-dependent. Despite this, there are some good choices that make the algorithm work well in a lot of different cases. We will discuss the parameter involved in the PSO version we implemented.

Consistent with the observations in [1], to prevent a huge increase of the velocity the inertia weight must be smaller then 1. There is no a specific rule for the choice of the other two parameters. However, many analysis show that a good choice is constraining the cognitive factor and the social factor within $[1, 3]$.

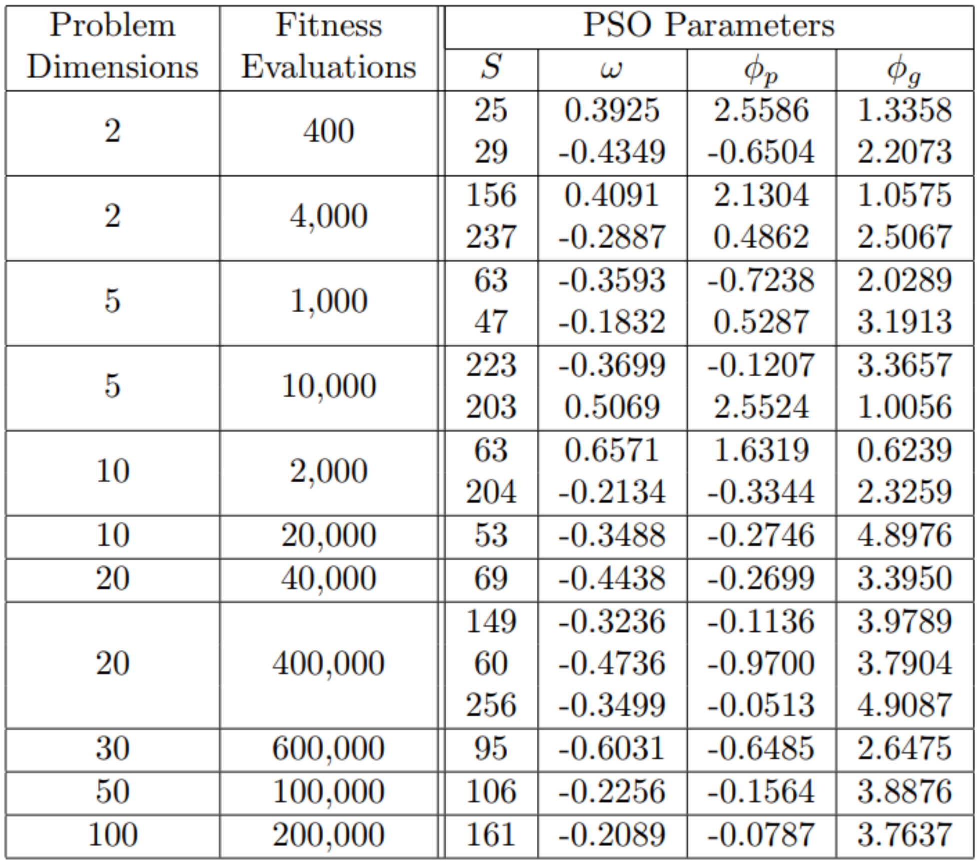
As mentioned in the GMO course [2], a usual choice for the inertia parameter w is a function that linearly decrements from 0.9 to 0.4. We fix

$$w = 0.8.$$

The usual choice for the cognitive factor and the social factor is

$$\begin{aligned} \phi_{\text{particle}} &= 1.49445, \\ \phi_{\text{swarm}} &= 1.49445. \end{aligned}$$

Due to the problem dependency of the parameters, it would be a great to know which parameters use depending on the problem we are tackling. In accordance with the study [4], there is a table that shows which parameters use depending on the problem. In particular, if you know the dimension of the problem and which is the maximum number of fitness evaluations you will do, then the following table suggests the parameters:



Problem Dimensions	Fitness Evaluations	PSO Parameters			
		S	ω	ϕ_p	ϕ_g
2	400	25	0.3925	2.5586	1.3358
		29	-0.4349	-0.6504	2.2073
2	4,000	156	0.4091	2.1304	1.0575
		237	-0.2887	0.4862	2.5067
5	1,000	63	-0.3593	-0.7238	2.0289
		47	-0.1832	0.5287	3.1913
5	10,000	223	-0.3699	-0.1207	3.3657
		203	0.5069	2.5524	1.0056
10	2,000	63	0.6571	1.6319	0.6239
		204	-0.2134	-0.3344	2.3259
10	20,000	53	-0.3488	-0.2746	4.8976
20	40,000	69	-0.4438	-0.2699	3.3950
20	400,000	149	-0.3236	-0.1136	3.9789
		60	-0.4736	-0.9700	3.7904
		256	-0.3499	-0.0513	4.9087
30	600,000	95	-0.6031	-0.6485	2.6475
50	100,000	106	-0.2256	-0.1564	3.8876
100	200,000	161	-0.2089	-0.0787	3.7637

Figure 1: PSO parameters for various problem configurations. The practitioner should select the PSO parameters where the dimensionality and allowed number of fitness evaluations most closely match those of the optimization problem at hand. For some problem configurations multiple parameters are listed as they had almost the same optimization performance.

3 Overview of JGEA

3.1 What is JGEA

JGEA is an optimization framework that is very general.

3.2 Interface Approach

JGEA is based on interfaces.

A Java interface is an abstract type used to designate a set of abstract methods for classes to implement. As mentioned before, when a class implements an interface (an interface can be implemented by more classes), it must inherit all of the abstract methods declared within, as though signing into a contract and carrying out the agreement.

The most significant similarity between interfaces and classes is that they both contain methods. They are more different than they are the same, however: an interface lacks constructors, contains exclusively abstract methods (no method implementation), and exclusively final and static fields.

3.3 The Most General Interface

JGEA is a structure of nested interfaces. The most general in which we are interested here is the *Solver* interface. It takes a problem P in input and returns a collection of solutions collection(S). It has the capability to solve a wide range of problems (due to its generality and abstraction). It needs some more concrete interfaces (extensions) in order to be able to actually solve a problem.

Within the *Solver* interface, we can see that there is the sequence `P extends Problem`. It means that P is at least a problem. In other words, P has all the features of a Problem **and** may have more of them.

3.4 The Most General Problem

The most general problem is the interface *Problem*. It is structured like this

```
public interface Problem<S> extends PartialComparator<S> {}
```

That means that the *Problem* interface is at least a partial order operation. That means that there is a way to know if a solution is better than another **and** not all the solutions are comparable. In fact, you cannot define an optimization problem if you cannot compare two elements.

3.5 Decrease the Abstraction

Solver is the most abstract interface we saw. There exists a sequence of interfaces where each of them is an extension of the previous one, and the more you go forward, the more concrete interfaces you get. In mathematical terms, there exists a sequence of interfaces $i_1, i_2, \ldots, i_r$ such that

$$i_r \text{ extends } i_{r-1} \text{ extends } \ldots \text{ extends } i_1 \text{ extends } \textit{Solver}.$$

IterativeMethod is an interface that is just one abstraction level below *Solver*. In fact, we have

```
IterativeMethod extends Solver.
```

It is an interface with methods to solve a problem iteratively.

Next, there is *AbstractPopulationBasedIterativeSolver*, an extension of *IterativeSolver*. This is the first level where

$$G \neq S.$$

It means that we can derive a genotype space G from the phenotype space S. This is done since population-based methods have functions that must be applied on the genotype space (e.g., evolutionary operators) **and** functions that must be applied on the phenotype space (e.g., the fitness evaluation).

The next level is *AbstractStandardEvolver*. It is even more concrete, with methods like

- select
- evolutionary operators
- etc.

In interface terms, this is the most concrete level I will report. Now, there are classes that implement this interface.

One such class is *StandardEvolver*. It contains

```
public class StandardEvolver<G, S, Q> extends AbstractStandardEvolver ...
```

Another class that extends *AbstractStandardEvolver* is *EvolutionaryStrategies*. It is an interface used to address problems where

- $G = \mathbb{R}^p$
- Order = Total order
- Mutation is: $x_{i,\text{new}} = x_i + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$

4 PSO Implementation on JGEA

We implemented the PSO algorithm within Jgea. To be more specific, we implemented the ParticleSwarmOptimization class in the branch *develop*.

The abstraction level of PSO is the same as *SimpleEvolutionaryStrategies*, where

$$G = \mathbb{R}^p = \text{list of double}.$$

Note that PSO works on $\mathbb{R}^p$. We implemented PSO to work on any searching space S that allows to define a function f_{map} s.t.

$$f_{\text{map}} : S \to G \text{ where } (G = \mathbb{R}^p).$$

For this case, as an evolutionary algorithm, we apply the fitness evaluation on $s = f_{\text{map}}(g) \in S$, while the position update on $g \in \mathbb{R}^p$.

4.1 The State of the Algorithm

Each algorithm has a state. With state, we mean a set of information that we can have at a fixed iteration. The state of a genetic algorithm (called *Standard evolver* in Jgea) is

$$\text{state} = \text{population}.$$

In the PSO case, given a population P, the state is

$$(p_{\text{pos}}, p_{\text{vel}}, p_{\text{particle best}}), \forall p \in P$$

For this reason, we need to declare the state of PSO as something more than the state of GA, i.e., `T extends State`.

4.2 Elements

Variables

- T = state type
- P = the problem
- I = an individual

Functions

- *solutionMapper:* $G \rightarrow S$
- *init:* it generates a population in G (in PSO, $G = \mathbb{R}^p$)

4.3 The Individual

In Jgea, there exists an interface called *Individual.* It has functionality related to what is an individual of the population in a population-based method. It has only one field, that is the *phenotype.*

In PSO, two further fields are needed:

- velocity: the velocity of the individual
- particle_best: the fittest position found so far by the particle

So, we defined inside the class *ParticleSwarmOptimization* an interface

```
PSOindividual extends Individual
```

Therefore, *PSOIndividual* is at least an *Individual.* Moreover, we added the *velocity* and the *particle_best.*

4.4 Initialization

1. **Initialize standard positions**
   We used a pre-implemented function that builds n random positions in $[-1,1]^p \subset \mathbb{R}^p$.

2. **Define the interval**
   For all arguments i, we take the min and the max values of the generated position (let's call them $x_{i,\min}$ and $x_{i,\max}$)

3. **Initialize the velocity**
   For all individuals x, for all arguments i, extract a random value

$$v_{x,i} \sim U[-|x_{i,\max} - x_{i,\min}|, |x_{i,\max} - x_{i,\min}|]$$

4.5 Update

4.5.1 The Update

At each iteration, we update the position of each particle x of the swarm. Firstly, we update the velocity

$$v_{\text{new}} \leftarrow \text{update}(v)$$

After, we can update the position simply by

$$x_{\text{new}} \leftarrow x + v_{\text{new}}$$

The update of the velocity of a particle depends on:

- The best position found so far by the particle
- The global best position found so far by the swarm
- Some parameters
- Some random variables

To be more accurate, for each argument j

$$v_{\text{new},j} = w \cdot v_j + \phi_{\text{particle}} \cdot r_{\text{particle}} \cdot (b_{\text{particle},j} - x_j) + \phi_{\text{swarm}} \cdot r_{\text{swarm}} \cdot (b_{\text{swarm},j} - x_j)$$

where w, ϕ are the parameters, r are the random variables, b_j are the values of the j-th argument of the fittest position found so far (by the particle or by the swarm).

Remember that PSO is a population-based optimization algorithm, where each particle represents a potential solution in a multidimensional search space. The algorithm iteratively updates the positions and velocities of particles to explore the solution space and find optimal solutions.

4.5.2 The Stopping Criterion

The algorithm stops the iterations and returns the fittest particle when a certain number of fitness evaluations is reached. This is a parameter that can be selected.

4.6 Parameters

There are multiple choices for the parameters. The literature shows how the choice of the parameters highly affects the results. Moreover, it seems that the parameter selection should depend on the dimension of the problem **and** on the number of fitness evaluations that will be done.

We set, as default,

$$\begin{aligned} w &= 0.8 \\ \phi_{\text{particle}} &= 1.5 \\ \phi_{\text{swarm}} &= 1.5 \end{aligned}$$

However, the user can specify different parameter values.

5 PSO code application

5.1 Tested problems

`ea.p.s.ackley()`:
Ackley's function is a non-convex function. It has many local minima and a single global minimum. The function is characterized by its smooth, bowl-shaped appearance.
`ea.p.s.sphere()`:
The Sphere function is a simple and convex optimization problem. It is defined as the sum of the squares of the variables and has a unique global minimum at the origin.
`ea.p.s.pointAiming()`:
Points Aiming defines an optimization problem where the objective is to find a solution that minimizes the distance to a set of target points in Euclidean space.
`ea.p.s.circularPointsAiming()`:
Circular Points Aiming extends the Points Aiming problem by specializing it for a circular arrangement of target points. The goal of optimization is to find a solution that minimizes the distance to these circularly arranged targets.
`ea.p.s.rastrigin()`:
The Rastrigin function is a non-convex, multimodal function. It is characterized by a large number of local minima and is used to test the robustness of optimization algorithms.

5.2 Compared models

- `pso`: Particle Swarm Optimization:
  the method we implemented.

- `cmaEs`: Covariance Matrix Adaptation - Evolutionary Strategies.
  It is known for its ability to handle complex, ill-conditioned, and high-dimensional optimization problems by adapting the covariance matrix during the optimization process.

- `doubleStringGa`: Genetic algorithm for binary string representation.
  It is a population-based optimization algorithm designed for problems with solutions represented as lists of real numbers. It provides a flexible and customizable approach to optimization.

- `simpleEs`: Evolution Strategies.
  It leverages mutation to explore the search space, and elitism helps in preserving the best solutions.

- `differentialEvolution`: Differential Evolution.
  It is simple and effective. The differential mutation helps explore the search space, and the crossover operation aids in sharing information between individuals.

6 Results and discussion

I tested each of the reported model with each of the shown problems. For each pair (solver, problem), I tested the solver for different dimensions (2,10 and 100) and for different parameter choice. The first parameter choice was the "good enough for all" choice, i.e.,

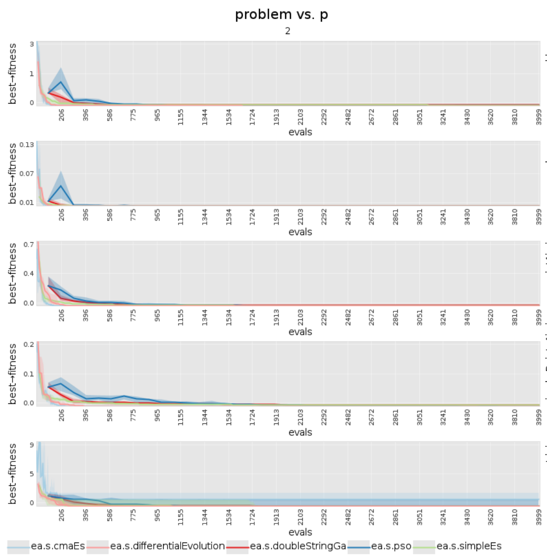
$$\begin{aligned} w &= 0.8, \\ \phi_{\text{particle}} &= 1.5, \\ \phi_{\text{swarm}} &= 1.5. \end{aligned}$$

While, the second parameter choice was the proper one following the results of the paper [4] (so, it depends on the number of fitness evaluations and on the size of the problem).

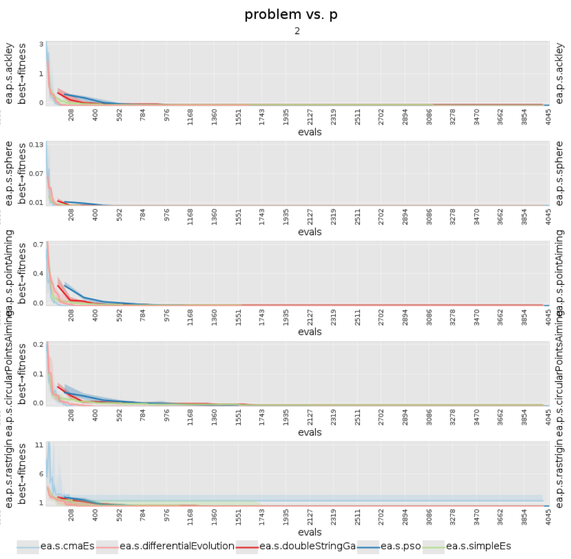
All the plots have the same structure. The plot is divided in two columns. On the right, all the PSO methods are run with the good enough for all parameters. While, on the left, it is the case where PSO has the proper parameters for the studied case. In each of the two parts there are 5 plots one below the other. Each single plot is the result of the application of all the 5 solvers on one the 5 problems. In particular, each plot has as x-axis the number of fitness evaluations and on the y-axis the fitness of the fittest individual. Each different line is the result of a different solver. PSO is the blue one.

6.1 Dimension 2

The first test is for 2-dimension problems.



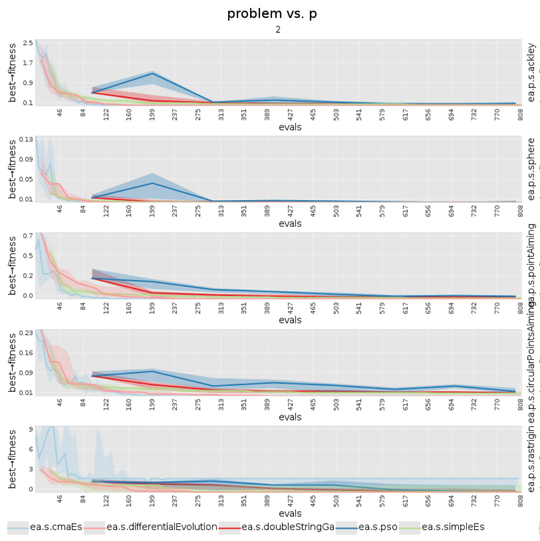
(a) PSO with default parameters



(b) PSO with case-suited parameters

Figure 2: Dimension 2

We can say that all the methods are effective. In fact, it appears that all of them have found a great solution (I would say the global optimum). I did a further test in order to focus on the first evaluations.


(a) PSO with default parameters

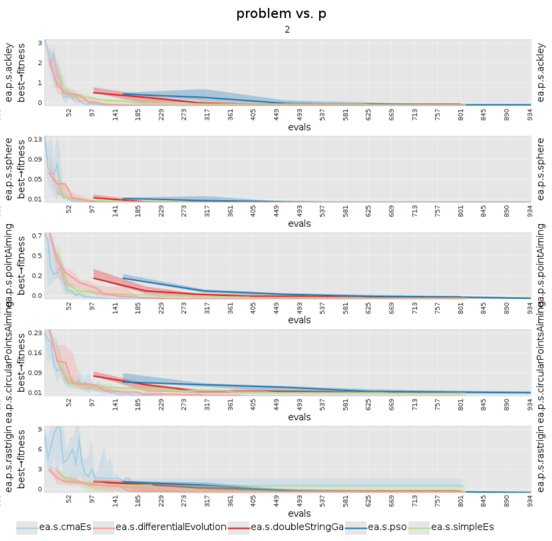
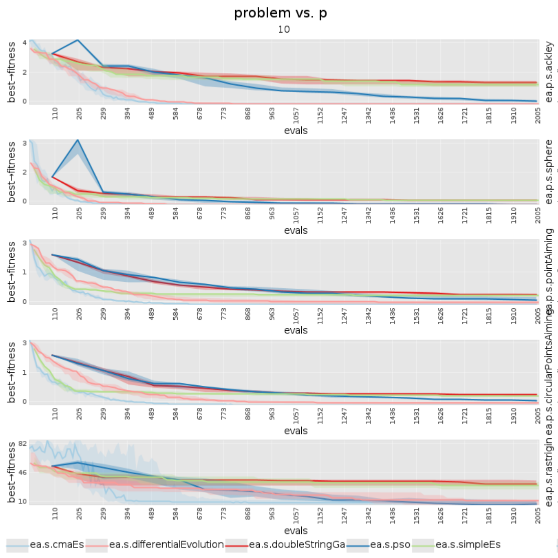

(b) PSO with case-suited parameters

Figure 3: Dimension 2, focus on the first evaluations

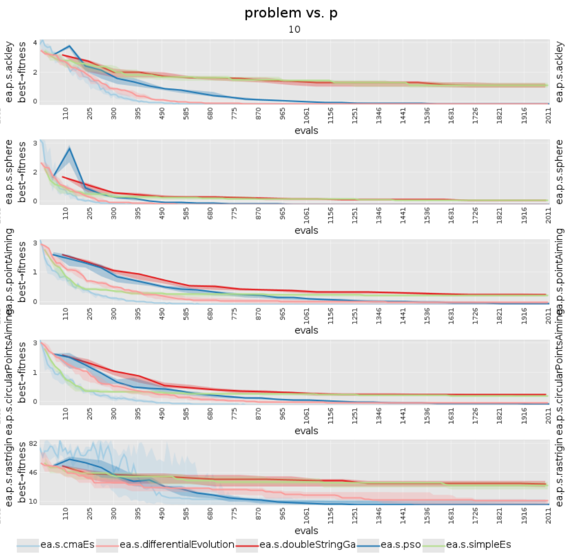
As illustrated by the plots, with the case-suited parameters the PSO algorithm is faster.

6.2 Dimension 10

Then, I repeated the same tests with the same 5 problems in 10 dimensions.



(a) PSO with default parameters

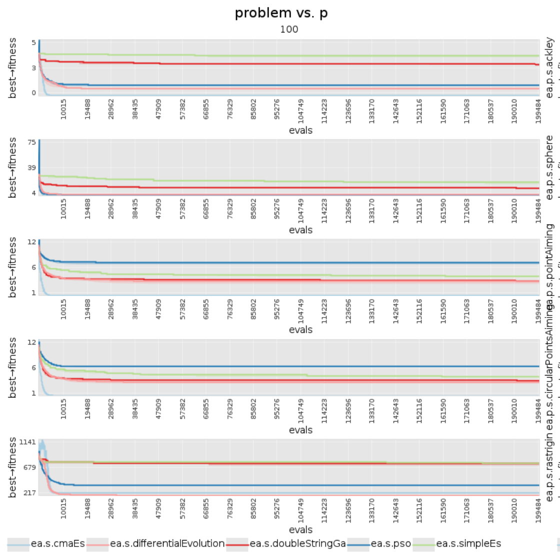


(b) PSO with case-suited parameters

Figure 4: Dimension 10

As shown in the plot, it appears that the case-suited parameters make the PSO algorithm slightly faster.

6.3 Dimension 100


(a) PSO with default parameters

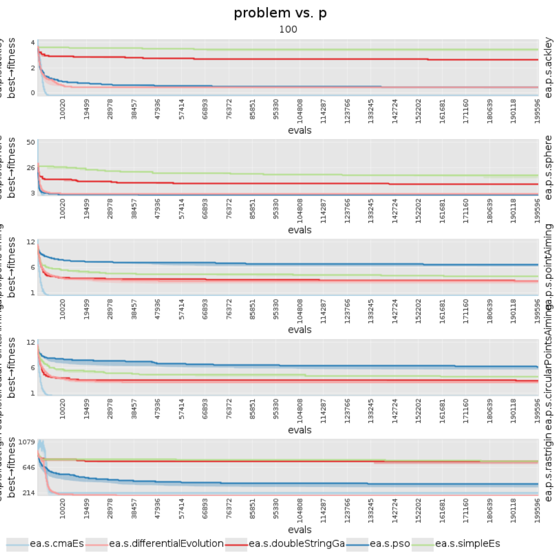
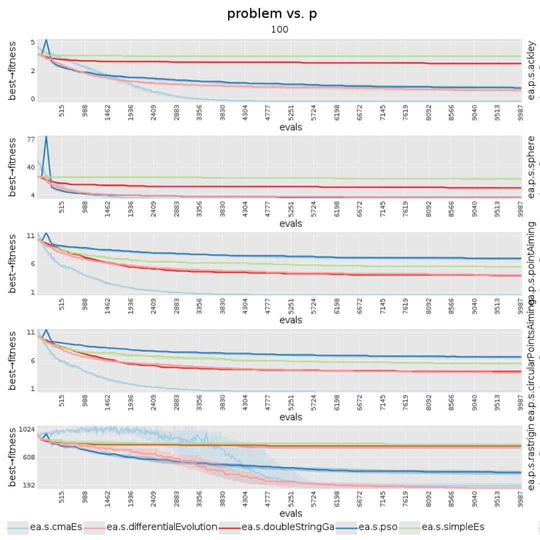

(b) PSO with case-suited parameters

Figure 5: Dimension 100

According to the plot, after a certain number of evaluations, the solvers were not able to find a much fitter solution. Then, I focused on the first evaluations.


(a) PSO with default parameters

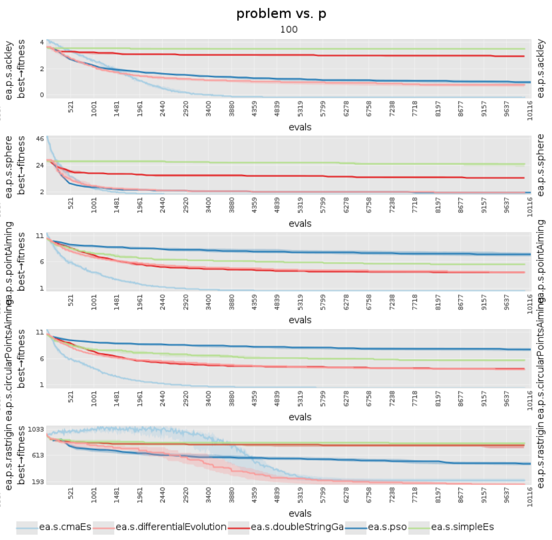

(b) PSO with case-suited parameters

Figure 6: Dimension 100, focusing on the first evaluations

6.4 Final comment

With respect to the tests done, PSO seems to work. Moreover, the standard paramater

$$
\begin{aligned}
w &= 0.8, \\
\phi_{\text{particle}} &= 1.5, \\
\phi_{\text{swarm}} &= 1.5
\end{aligned}
$$

are good in any tested situation.

References

[1] Particle swarm optimization. `https://en.wikipedia.org/wiki/Particle`$_s warm_o ptimization$.

[2] A. Manzoni. Global multiobject optimization course. `https://units.coursecatalogue.cineca.it/insegnamenti/2023/119224/2018/4/10538?coorte=2022`
2023.

[3] E. Medvet, G. Nadizar, and L. Manzoni. Jgea: a modular java framework for experimenting with evolutionary computation. In *Proceedings of the genetic and evolutionary computation conference companion*, 2022.

[4] M. E. H. Pedersen. Good parameters for particle swarm optimization. Technical Report HL1001, Hvass Laboratories, 2010.