Intermediate Report

Nicolas Bougie, Nawel Medjkoune, Stephen Batifol,
Abdelhadi Temmar, and Abdoun Sihem

October 10, 2016

1 Introduction

1.1 Project Context

For this project, we had to work on evolution algorithms that are commonly applied to black-box optimization problems. From the algorithm pointview, evolution strategies are optimization methods that sample new candidates solutions stochastically. One had to implement a chosen algorithm described in [Hansen et al., 2015], we chose to implement the given algorithm with Python. The algorithm had his parameter control via self-adaptation. One had then to run a suite of test functions in order to benchmark the performance of the implemented algorithm. As central performance measure of algorithms COCO compute the "runtime" measured by the number of evaluations performed on the problem. In other words "runtime" is defined by the number of calls to the objective function until a target value is hit. This performance measure is independent from the used programming language and the material running the algorithm.

1.2 Project Goals

In order to know which algorithm to use for a particular numerical black-box optimization problem, we have to benchmark multiple algorithms. For this particular project, we have implemented $(\mu/\mu \overset{+}{,} \lambda) - ES$ [Hansen et al., 2015], a non-benchmarked algorithm and benchmarked it with COCO [Hansen et al., 2016b]. We can then compare our implementation of this algorithm with another group that implemented the same algorithm but with another programming language. Finally, we compare this algorithm with two baseline algorithms, BIPOP-CMA-ES [Hansen, 2009] and BFGS [Ros, 2009].

In this intermediate report, we state our progress during the first part of the project. In section 2.1, we explain the algorithm implemented and give the pseudo-code. In section 2.2, details about implementation are explained. In section 2.3, the algorithm integration in COCO is detailed. In section 2.4, we sum up the preliminary experiments configuration. Results are shown and discussed in section 2.5 and suggestion about further investigation are given.

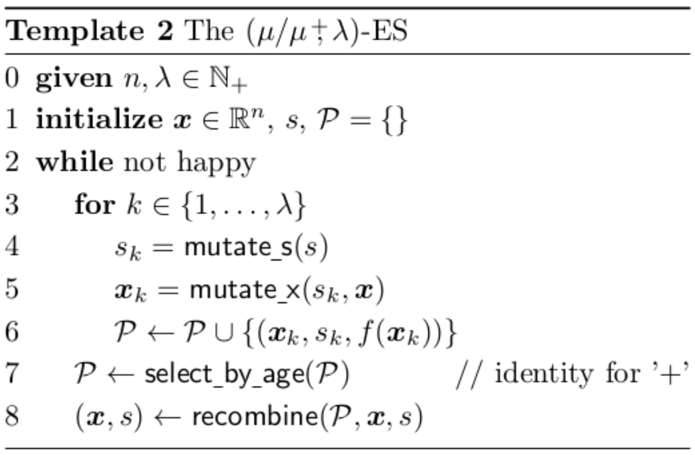
We state some of the encountered difficulties during implementation in the next section. Finally, in section 3, we sum up what remains to be done in the project.

2 Progress state

2.1 Understanding the algorithm

The algorithm studied in this project is $(\mu/\mu \overset{+}{,} \lambda) - ES$ [Hansen et al., 2015]. Prior to implement the algorithm, we first had to understand it. We proceeded by understanding what an evolutionary algorithm consist of and the mechanism behind the template 2 of the paper which corresponds to the pseudocode of the evolution strategy that is used in the article.

First of all, evolutionary algorithms are a set of algorithms inspired from the theory of biological evolution frequently used to solve optimizations problems. These algorithms are most commonly population-based where each individual of the population is considered as a candidate solution to the problem and its quality is measured by a loss function (objective function). The concept of the algorithm is close to biological evolution and Darwins theory. The initial population is randomly generated (origin of life). This population has a very low evaluation score and is weak (relative to survival). The offsprings (second generation) are generated from one or several parents. These descendants inherit genes from their parents and then mutate their genes . Only the strongest individuals survive (the ones with the best evaluation score). This leads to a stronger and stronger population, in other words : get closer to the optimal solution. The purpose is then to improve the solutions iteratively by generating new solutions through stochastic mechanisms such as: selection, recombination or mutation.

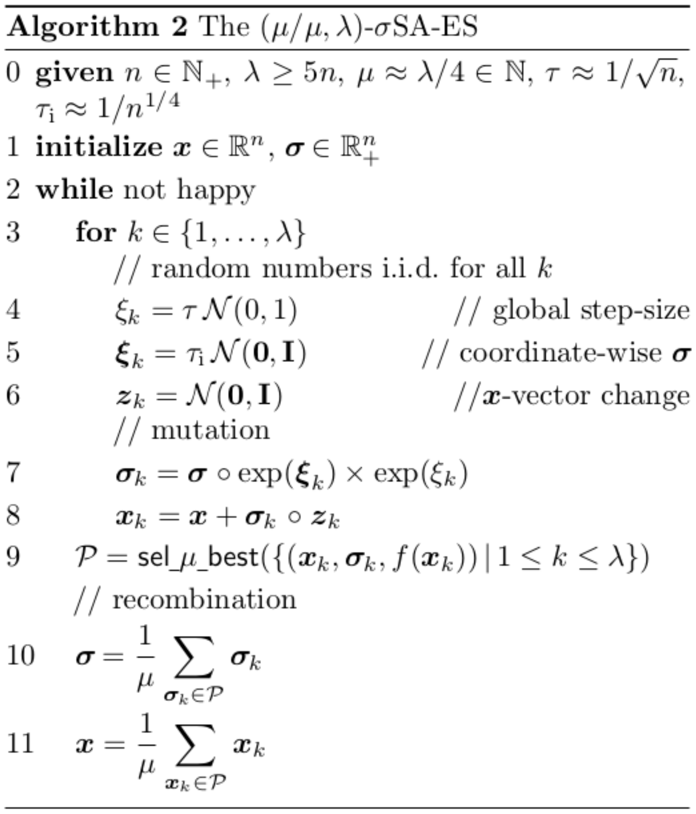


Template 2 The $(\mu/\mu \overset{+}{,} \lambda)$-ES

```
0 given $n, \lambda \in \mathbb{N}_+$
1 initialize $\boldsymbol{x} \in \mathbb{R}^n$, $s$, $\mathcal{P} = \{\}$
2 while not happy
3     for $k \in \{1, \ldots, \lambda\}$
4         $s_k$ = mutate_s($s$)
5         $\boldsymbol{x}_k$ = mutate_x($s_k, \boldsymbol{x}$)
6         $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\boldsymbol{x}_k, s_k, f(\boldsymbol{x}_k))\}$
7     $\mathcal{P} \leftarrow$ select_by_age($\mathcal{P}$)            // identity for '+'
8     $(\boldsymbol{x}, s) \leftarrow$ recombine($\mathcal{P}, \boldsymbol{x}, s$)
```

Figure 1: Template 2

Figure 1 shows the template 2 exposed in the paper. The particularity of this template is the use of a single parental centroid (x, s), which simplify the algorithm. We shall use this single parental centroid to create a new generation of offspring. S_k, more commonly called "standard deviation", is used to generate a new offspring X_k. Intuitively, The bigger S_k is, the bigger the distance between X and X_k is. This is why σ and σ_k change through iteration, we want this parameter to adapt over time, this process is called Self Adaptation. We use X and S to generate the set $(x_k, s_k, f(x_k))$ for $k = [0, \lambda]$. At line 6 and 7, the population is updated using the new samples. At line 8, X and S, the parental centroid, is updated.


Algorithm 2 The $(\mu/\mu, \lambda)$-σSA-ES

```
0  given $n \in \mathbb{N}_+$, $\lambda \geq 5n$, $\mu \approx \lambda/4 \in \mathbb{N}$, $\tau \approx 1/\sqrt{n}$,
   $\tau_\mathrm{i} \approx 1/n^{1/4}$
1  initialize $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{\sigma} \in \mathbb{R}^n_+$
2  while not happy
3      for $k \in \{1, \dots, \lambda\}$
           // random numbers i.i.d. for all $k$
4          $\xi_k = \tau \mathcal{N}(0,1)$                 // global step-size
5          $\boldsymbol{\xi}_k = \tau_\mathrm{i} \mathcal{N}(\mathbf{0}, \mathbf{I})$          // coordinate-wise $\boldsymbol{\sigma}$
6          $\boldsymbol{z}_k = \mathcal{N}(\mathbf{0}, \mathbf{I})$              //$\boldsymbol{x}$-vector change
           // mutation
7          $\boldsymbol{\sigma}_k = \boldsymbol{\sigma} \circ \exp(\boldsymbol{\xi}_k) \times \exp(\xi_k)$
8          $\boldsymbol{x}_k = \boldsymbol{x} + \boldsymbol{\sigma}_k \circ \boldsymbol{z}_k$
9      $\mathcal{P} = \mathsf{sel\_\mu\_best}(\{(\boldsymbol{x}_k, \boldsymbol{\sigma}_k, f(\boldsymbol{x}_k)) \mid 1 \leq k \leq \lambda\})$
       // recombination
10     $\boldsymbol{\sigma} = \frac{1}{\mu} \sum_{\boldsymbol{\sigma}_k \in \mathcal{P}} \boldsymbol{\sigma}_k$
11     $\boldsymbol{x} = \frac{1}{\mu} \sum_{\boldsymbol{x}_k \in \mathcal{P}} \boldsymbol{x}_k$
```

Figure 2: Algorithm 2

Figure 2 shows the pseudocode of the $(\mu/\mu \overset{+}{,} \lambda) - ES$ algorithm. The goal of the algorithm is to minimize a black box function. More precisely, the algorithm tries to find a X (entry of f) which will minimize $f(X)$. For this reason, we first have to initialise randomly X and try to find near this parental centroid a better solution. To understand this algorithm, we need to understand how the mutation, selection and recombination are made. First of all, the mutation. During this step, we create a new generation (λ childs) by applying random changes following a normal distribution. We can see at line 7, that we apply 2 to $\boldsymbol{\sigma}$. One mutation who is common to all components of the vector (the scalar computed at line 4), and one who is different for each component of $\boldsymbol{\sigma}$ (the vector computed at line 6). For x, we apply a different mutation for each component (line 8). The sign of $\sigma_k \circ z_k$ will indicate for each component of X which direction to follow.

Once the mutation is done, we shall do the selection. Individuals selection is done by taking the best individuals among the lambda individuals of the new generation. In this algorithm we let individuals die after one iteration.

The recombination is simple. We compute the average of the new population to update the parental centroid (line 10, 11). However, the application of mutation and recombination introduces a moderate bias such that σ tends to increase.

In order to achieve stable behavior of σ, the number of parents μ must be large enough, which is reflected in the setting of λ.

2.2 Algorithm Implementation and General Code Structure

Prior to implementing the algorithm, several questions were raised:

1. How do we initialize X and σ?
2. When do we stop the algorithm?
3. How do we choose the value of λ?
4. What budget do we choose?

These are not trivial choices that can be decided with theory only. Experiments need to be done in order to make the best decisions.

For now, we decided to initialize X between -5 and 5 because according to the bbob dataset documentation,the optimum is always between -5 and 5. σ is initialized between 0 and 0.01 to start with a small value. This choice was made because we know that the optimum is between -5 and 5. Therefore, we do not need a big standard deviation.

The current implementation of the algorithm stops only when no budget is left, to see where the algorithm can go. In the future, the algorithm will stop itself when it finds a solution equals or smaller than the target objective.

Lambda is currently set to $5 * n$, where n is the size of the vector X. It is the minimum value advised by the authors of the paper. In the future, we will try to change that value, to see if it affects the results.

We try to test our algorithm with multiple budgets to see for which values the algorithm converges.

2.3 Algorithm integration in Coco

Coco is a platform for comparing continuous optimizers in a black-box setting. Coco provides an interface to run our own optimizer. We first changed the search space also called "dim" . We chose different dimensions to see if the algorithm could efficiently find a "good" solution in a higher dimension. We benchmarked our implementation of the algorithm on the BBOB suite, which contains mono objective and noiseless functions [REF].

The optimizer is called from the run_optimizer function. This function takes as parameters the evaluation function, the dimension of the search space, the maximum number of evaluations and the optimal solution. These values are provided by COCO. Inside this function, the parameters specific to our algorithm are generated. Then, we call our optimizer with all the previous parameters.

We also set the budget (maximum number of evaluations), we chose a small value to accelerate the computations. Our function does not need to be modified. It just returns the best solution and check that the number of evaluation is not exceeded. Coco automatically measures the performances for each problem and generate a folder containing the output.

Coco also provides a LateX template to display a summary for the single-objective BBOB suite and algorithm performances.

2.4 Experiments configuration

To sum up what has been said before, we present the overall configuration for the preliminary experiments. We ran the algorithm $(\mu/\mu\overset{+}{,}\lambda) - ES$ on the BBOB noiseless functions test suite over different configuration settings: first of all, for the purpose of debugging and understanding the algorithm behaviour, we experimented relatively small values of budget and dimension. We tested three different budgets (500, 1000, 5000) with three different dimensions (2,3,5). The algorithm $(\mu/\mu\overset{+}{,}\lambda) - ES$ was implemented in Python. The initial solution is generated randomly in $[-5, 5]$ Dim and a very small value for sigma is generated randomly as well. The parameter λ is initially set to $5 * dim$ and μ is one fourth of λ according to the paper. Finally, the maximum number of runs (restarts) in set to its default value: $1e9$.

2.5 Results and comments

Results from our preliminary experiments are shown in the next figures. Figure 3 represents runtime distribution (ECDFs) over all functions, targets and instances.

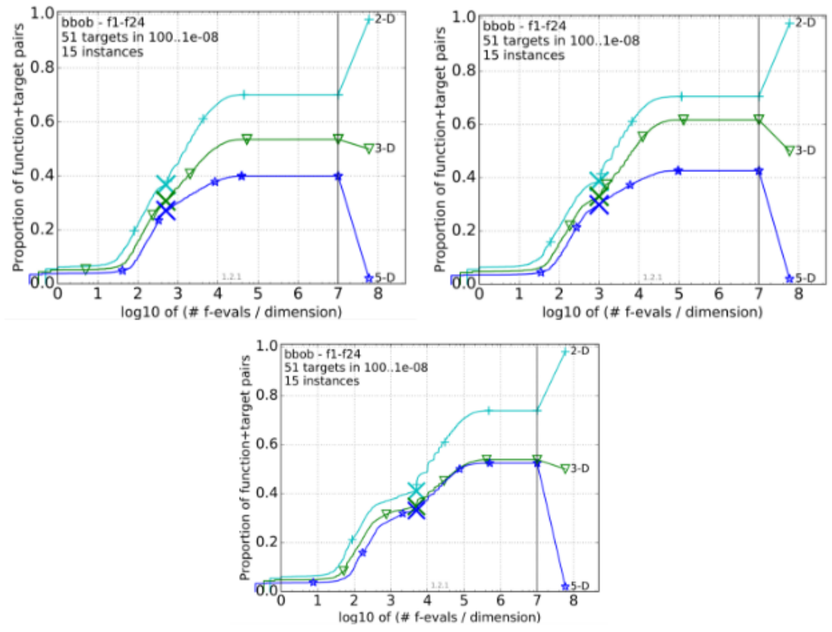


Figure 3: Runtime distributions (ECDFs) over all targets with budget=500, 1000, 5000 from left to right.

We observe that the algorithm with this configuration barely reaches 40% of success over all the problems. However, we notice that the proportion of the problems solved increases slightly when the budget is increased. This observation is a motivation to go deeper in the experiments and test even bigger budgets. For the remaining discussion, we consider the results from the last run (i.e. with the budget of 5000 and dimension of 5).

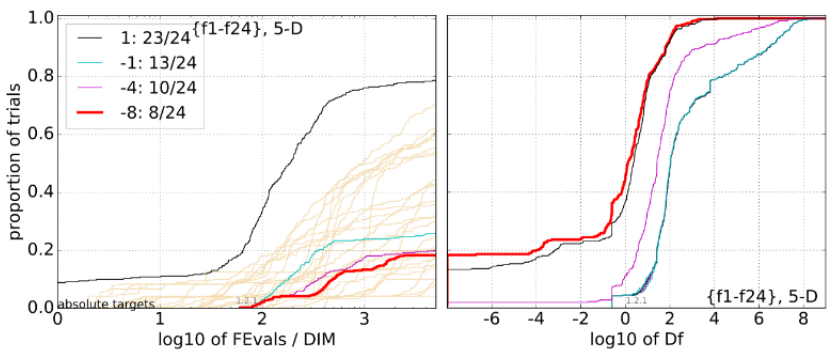


Figure 4: left: runtime distributions (ECDFs) for selected targets (precision), right: Legends indicate for each target the number of functions that were solved in at least one trial within the displayed budget.

In figure 4, we can observe the overall performance of the algorithm on all the functions depending on the target precision. For large precisions (i.e. $\Delta f = 101$), most of the problems are solved. The less the precision is, the worst the algorithm performance is. This problem needs to be investigated in the second half of the project. First suggestion is to allocated bigger values for the budget. Another suggestion is to vary the values of multiple input parameters of the algorithm and try other initialisation types. However, the algorithm performance over difficult precisions is still good when compared to the BBOB 2009 benchmarking results (light brown lines)

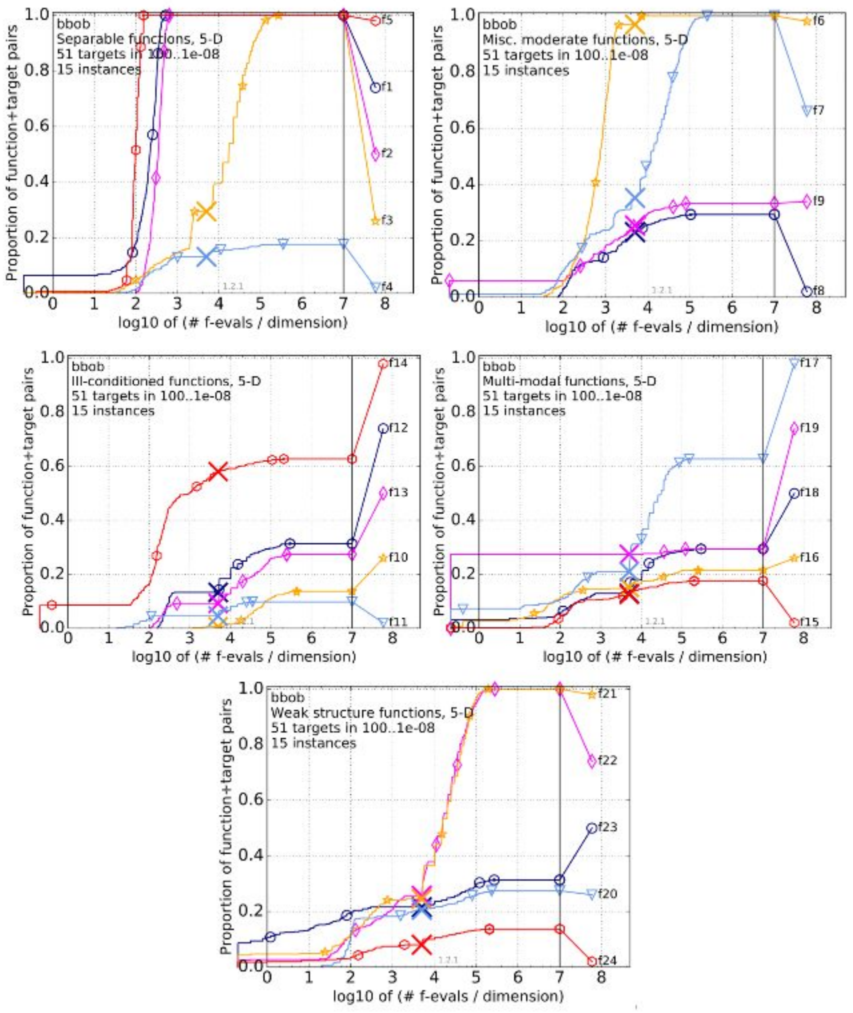


Figure 5: Runtime distributions (ECDFs) per group and per function over all targets (dimension=5)

Figure 5 shows ECDFs per groups for dimension 5. For the first group, 3 functions out of 5, as well as one function in the second group, are 100% solved in the allowed budget. That said, most of the remaining problems, especially for the multi-modal and weak structure functions, do not exceed the 30% threshold. Undoubtedly, problems with small precision are the ones that are solved unsuccessfully.

Our main point of investigation for the future is, as said earlier, trying to increase maximum budget allowed and observe the results. Bigger dimensions have to be investigated as well.

2.6 Encountered difficulties

When we implemented the first version of $(\mu/\mu \overset{+}{,} \lambda) - ES$, we made some experiments with small values for the budget and dimensions, in the purpose of debugging our code. This method proved to be useful since we noticed very poor proportion of solved problems (which was strange for the separable functions) and some inconsistencies on the plots. Despite the few coding errors we patched, another modification was to change the initialisation pattern of solutions from completely random to a uniform distribution in $[-5, 5]^{Dim}$ as suggested in the COCO documentation. This modification improved significantly the results.

3 Remaining work

In order to complete this project, we shall carry more experiments with more iterations. Because we only tested if our algorithm was correct and our understanding of COCO, we only allocated a maximum budget of 5,000. Our objective is to test our algorithm on a budget as high as 50,000. With this budget, we could compare our results with the ones produced previously on the BBOB serie. Another task would be to compare our implementation with the implementation of an other group. We could also vary the offspring population size to assess the impact of this variable on the results

3.1 Go deeper in the experiments

As suggested in the article, the change of the population size parameters μ and λ has a significant impact on the search characteristics of an evolution strategy. We will carry multiple experiments in order to investigate the impact that these parameters will have on the algorithm performance.

3.2 Comparing results with other algorithms

As mentioned above, we chose the Python programming language to implement the chosen algorithm. Since there is another group working on the same algorithm but in C/C++, it would be interesting to compare our respective algorithms on the basis of their performance. Two baseline algorithms, namely BIPOP-CMA-ES [Hansen, 2009] and BFGS [Ros, 2009], are also used for the comparison.

4 Conclusion

As a conclusion for the first part of this project, we can say that we understood how COCO works and how it benchmarks algorithms. With the work done so far, we also understood what an evolutionary algorithm is and how it works and how to deal with numerical black-box optimization problems. As previously stated, for the remaining part of this project, we want to run our algorithm with a higher budget, we also want to change the population size parameters to investigate the impact that they will have on the algorithm performance.

As mentioned before, compare our algorithm with other evolutionary algorithms is crucial. We could compare it with different optimisation techniques such as particle-swarm, differential evolutions or variable-neighborhood search.

References

[Hansen, 2009] Hansen, N. (2009). Benchmarking a bi-population cma-es on the bbob-2009 function testbed. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2389–2396. ACM.

[Hansen et al., 2015] Hansen, N., Arnold, D. V., and Auger, A. (2015). Evolution strategies. In *Springer Handbook of Computational Intelligence*, pages 871–898.

[Hansen et al., 2016a] Hansen, N., Auger, A., Brockhoff, D., Tusar, D., and Tusar, T. (2016a). COCO: performance assessment. *CoRR*, abs/1605.03560.

[Hansen et al., 2016b] Hansen, N., Auger, A., Mersmann, O., Tusar, T., and Brockhoff, D. (2016b). COCO: A platform for comparing continuous optimizers in a black-box setting. *CoRR*, abs/1603.08785.

[Hansen et al., 2010] Hansen, N., Auger, A., Ros, R., Finck, S., and Pošík, P. (2010). Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '10, pages 1689–1696, New York, NY, USA. ACM.

[Hansen et al., 2011] Hansen, N., Finck, S., and Ros, R. (2011). COCO - COmparing Continuous Optimizers : The Documentation. Research Report RT-0409, INRIA.

[Hansen et al., 2016c] Hansen, N., Tusar, T., Mersmann, O., Auger, A., and Brockhoff, D. (2016c). COCO: the experimental procedure. *CoRR*, abs/1603.08776.

[Ros, 2009] Ros, R. (2009). Benchmarking the bfgs algorithm on the bbob-2009 function testbed. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2409–2414. ACM.