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

Margot Lacour, Adrien Pavao, Eléonore Bartenlian,
Thomas Foltête, Paul-Hadrien Bourquin

October 18, 2017

Abstract

The goal of this project was to implement and benchmark a black-box optimization algorithm using the COCO platform, specifically designed for such tasks. The (μ/μ)-σSA-ES algorithm we worked on is detailed in the article from Nikolaus Hansen, Dirk V. Arnold and Anne Auger: "Evolution Strategies". In Janusz Kacprzyk and Witold Pedrycz (Eds.): Handbook of Computational Intelligence, Springer, Chapter 44, pp.871-898, 2015.

1 Introduction

In this paper, we first briefly adress the paradigms of the Evolutions Strategies, and talk about the reason why are they used in general ; then, we dive into our specific algorithm and take a look at our implementation. With the help of COCO, we will finally look at different benchmarks given by multiple algorithms, and try to extrapolate information about when to use which algorithms from that.

1.1 Optimization problems

When faced with an Optimization problem, the objective is to find the best solution (i.e the solution that minimimises an objective function) in a continuous or discrete space. The search space is often far too large to use brute-force algorithms : thus, we need to use a different approach.

We will only take into account single objective optimization problems in this paper : this means that we have only one objective function, and not several objective functions that are competing which each other (multi-objective optimization problems). It is a simpler problem since we only care about maximizing (or minimizing) one function, without having to take into account other values that might decrease when our objective function is improved.

Thanks to COCO, we have an interface to implement our algorithm in a Black Box Optimisation Benchmark : COCO provides the input for different optimisation problems, and processes the output.
The fact that this is a Black Box optimization problem allows us to use any function to test the efficiency of our algorithm.

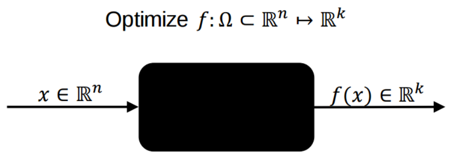
Optimize $f: \Omega \subset \mathbb{R}^n \mapsto \mathbb{R}^k$
$x \in \mathbb{R}^n$
$f(x) \in \mathbb{R}^k$

Figure 1: Black Box Optimization Problem

1.2 Evolution Strategies

Inspired from the principles of biological evolution, **evolution strategies** are the processes that are based on the **mutation**, **selection** and **recombination** of certain parts of populations. This also can be seen as optimization problems in which we sample candidates solutions. The paper from Nikolaus Hansen, Dirk V. Arnold and Anne Auger describes several template algorithms for evolution strategies.

Evolutions strategies are defined by the following features :

A population of P individuals is set, each individual being a feasible solution to the optimization problem. Each individual can be given a score, with the help of the fitness function, which gives a fitness value to each individual.

We first initialize the vector $\vec{x}$ that we want to optimize.

Then, we loop until until we reach either the budget or a good enough solution (the objective).

Random mutations are applied to $\vec{x}$ in order to obtain a population of size λ (offspring). Those changes are defined by a Gaussian distribution. They can be either slight or important.

Like in biological evolution, the less adapted individuals don't survive : We keep the μ best (a given number, $\frac{\lambda}{4}$ in our case) members of the population.

The "best" parents are defined by their fitness value.

We then recombine those μ parents by simply averaging them component by component to get one final vector $\vec{x}$ for the next iteration.

2 Description of the algorithm

An evolutionary algorithm is composed of three essential elements : a population composed of several individuals representing potential solutions (configurations) of the given problem, a mechanism for assessing the adaptation of each individual of the population to its external environment, and an evolution mechanism to eliminate certain individuals and to produce new individuals from the selected ones.

From an operational point of view, a typical evolutionary algorithm begins with an initial population often generated randomly and then a 3 steps cycle is repeated : we measure the fitness of each individual of the population by the evaluation mechanism, we select a part of the individuals, and we produce new individuals by recombining selected individuals. This process ends when the shutdown condition is checked, for example, when a maximum number of cycles (generations) or a maximum number of evaluations is reached. Just like in natural evolution, the quality of the individuals of the population should tend to improve as the process is progressing.[1]

The selection consists of choosing the individuals that will survive and/or reproduce to transmit their characteristics to the next generation. Selection is generally based on the principle of conservation of individuals better adapted and elimination of the less adapted. The recombination seeks to combine the characteristics of parent individuals to create individuals with new potentialities in the future generation. The mutation makes slight modifications to certain individuals.

Algorithm 1: The $(\mu/\mu, \lambda)$-σSA-ES

```
1  given $n \in \mathbb{N}_+, \lambda \geq 5n, \mu \approx \lambda/4 \in \mathbb{N}, \tau \approx 1/\sqrt{n}, \tau_i \approx 1/n^{1/4}$;
2  initialize $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{\sigma} \in \mathbb{R}^n_+$;
3  while not happy do
4      for $k \in \{1, ..., \lambda\}$ do
5          //random numbers i.i.d. for all k
6          $\xi_k = \tau \mathcal{N}(0, 1)$
7          $\boldsymbol{\xi}_k = \tau_i \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
8          $\boldsymbol{z}_k = \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
9          //mutation
10         $\boldsymbol{\sigma}_k = \boldsymbol{\sigma} \circ exp(\boldsymbol{\xi}_k) \times exp(\xi_k)$
11         $\boldsymbol{x}_k = \boldsymbol{x} + \boldsymbol{\sigma}_k \circ \boldsymbol{z}_k$
12     end
13     $\mathcal{P}$ =sel_$\mu$_best$(\{(\boldsymbol{x}_k, \boldsymbol{\sigma}_k, f(\boldsymbol{x}_k)) | 1 \leq k \leq \lambda\})$
14     //recombination
15     $\boldsymbol{\sigma} = \frac{1}{\mu} \sum_{\boldsymbol{\sigma}_k \in \mathcal{P}} \boldsymbol{\sigma}_k$
16     $\boldsymbol{x} = \frac{1}{\mu} \sum_{\boldsymbol{x}_k \in \mathcal{P}} \boldsymbol{x}_k$
17 end
```

Given a population of at least μ individuals, $x_k \in \mathcal{R}^n$ represents one of these individuals and also a solution vector (one which minimizes $f(x)$). A step size σ_k is associated to each of these individuals in order to control their mutations. At a step, λ children are generated.

This algorithm uses the idea of self-adaptation (SA) : new control parameters settings (the step size for example) and new x-vectors are generated by recombination and mutation.

[1] `http://www.info.univ-angers.fr/pub/hao/papers/RIA.pdf`

Lines 6 to 8 of the algorithm implement the fact that random events are generated in order to perform the mutations. These are performed line 10 for $\boldsymbol{\sigma}$, component by component ($\boldsymbol{\xi}_k$) and also one for all components ($exp(\xi_k$s). Theses mutations are unbiased : indeed, ξ_k and $\boldsymbol{\xi}_k$ have a normal distribution. Line 11 shows the mutation of $\boldsymbol{x}$ which uses the mutated $\boldsymbol{\sigma}_k$

A single parental centroid is used in our algorithm. Then, mutation takes this parental centroid as input and recombination is done at the end of the loop. Using a single parental centroid has become the most popular approach : such algorithms are simpler to formalize, easier to analyze and even perform better in various circumstances as they allow for maximum genetic repair[2].

3 Description of the implementation

We chose to implement this algorithm in Python, using librairies such as numpy and python.random.

3.1 Initialisation of paramaters

First of all, we defined the size of our research space n as the dimension of the function given by COCO. We also defined the number of children generated at each iteration as $\lambda = 5 * n$

Line 13 to 15 in our code describes the initialisation of parameters :

- The number of parents chosen per generation $\mu = \lambda/4$
- The settings for mutations $\tau = 1/\sqrt{n}$ and $\tau_i = 1/n^{0.25}$

The applicaion of mutation and recombination on σ introduces a moderate bias such that σ tends to increase under neutral selection. For σ to be stable, we need a number of parents μ that must be large enough, thus $\lambda \geq 5n$.

We then initialized $\boldsymbol{x}$ and $\boldsymbol{\sigma}$ randomly.

In a COCO point of view, we defined the budget, or the maximum iterations to process, as 300*n.

3.2 Implementation of the different operations

Hadamard product $(a \circ b)$ is easy to implement thanks to numpy :

```python
 def hadamard_product(m1, m2):
    """ Product of 2 matrixes (or vectors) which produce a same shape matrice """
    """ result[i,j] is m1[i, j] * m2[i, j]"""
    return np.multiply(m1, m2)
```

Listing 1: Implementation of Hadamard product

We also implemented the selection of μ best children as follows :

```python
def sel_mu_best(x_offspring, sigma_offspring, fun, _mu):
    # Evaluation of best x_k with function fun (minimization)
    # We choose sigmas_k associated to the best x_k
    l = [(x, y) for (x, y) in zip(x_offspring, sigma_offspring)]
    # Sort
    l = sorted(l, key=lambda colonnes:fun(colonnes[0]))

    return zip(*l[:_mu])
```

Listing 2: Implementation of the μ selection

The function takes four arguments :

- The generated offspring x and their standard deviation σ
- The fitness function given by COCO
- μ : the number of parents chosen per generation

The function sorts the offspring depending on the value of the fitness function for each child. Then the function returns the μ best children and their corresponding σ.

[2] https://www.lri.fr/~hansen/es-overview-2015.pdf

3.3 Stopping criteria

Our stopping criteria is shown at the ligne 25 of our code :

```
while ((iterations < budget))
```

Listing 3: Stopping criteria

This condition means that the number of iterations we do to select and recombine individuals can not exceed the budget fixed by COCO.

```
1      #call of algorithm_2
2      _n=fun.dimension
3      _lambda=5*_n
4      solver(fun, _n, _lambda, 300*_n)
5
6     def algorithm_2(fun, _n, _lambda, budget):
7      """ Evolution strategy implementation """
8      # fun is fitness evaluation
9      # budget : max iteration
10     # _n : dimensions
11     # _lambda : offspring population, must be >= _n*5
12
13     _mu = int(_lambda/4) # number of parents chosen per generation
14     _tau = 1.0/sqrt(_n)
15     _tau_i = 1.0/(_n ** 0.25) # degree of freedom
16
17     # x : the vector want to optimize
18     # entry of evaluation function
19     x = np.random.randn(_n) # Initialize x randomly on Rn
20     # sigma : deviation vector n applied on x
21     sigma = abs(np.random.rand(_n)) # Initialize sigma randomly on Rn+
22
23     iterations = 0
24
25     while ((iterations < budget)): # stop if max iteration
26
27         x_offspring = [] # x_k list before selection and recombination
28         sigma_offspring = [] # sigma_k list
29
30         for k in range (1, _lambda): # for each individual of generation
31
32             # random numbers i.i.d for all k
33             # thoose random numbers are for mutations
34
35             # dzeta_k is a value
36             dzeta_k = _tau * random.gauss(0, 1)
37
38             # I identity matrix, isotropic distribution case
39             I = np.identity(_n) #[[1, 0], [0, 1]]
40
41             # xi_k is a vector
42             xi_k = np.random.multivariate_normal(np.zeros(_n), I, _n).T * _tau_i # loi
       normale(0, I) * _tau_i
43
44             z_k = np.random.multivariate_normal(np.zeros(_n), I, _n).T
45
46             #mutation
47             sigma_k = hadamard_product(sigma, np.exp(xi_k)) * np.exp(dzeta_k)
48
49             x_k = matrix_sum(x, hadamard_product(sigma_k, z_k))
50
51             x_offspring.append(x_k[0])
52             sigma_offspring.append(sigma_k[0])
53
54         # natural selection
55         # the mu best (minimizing fun) individual are selected
56         x_best, sigma_best = sel_mu_best(x_offspring, sigma_offspring, fun, _mu)
57
58         #recombination
59         # we combine chosen parents in a mean to have just one vector x and one vector sigma
60         x = (1.0 / _mu) * np.sum(np.array(x_best), axis=0)
61
62
63         sigma = (1.0 / _mu) * np.sum(np.array(sigma_best), axis=0)
```

```
        iterations += 1
    return x
```

Listing 4: Implementation of the algorithm

4 CPU Timing

In order to evaluate the CPU timing of the algorithm, we have run the $(\mu/\mu, \lambda)$-σSA-ES Algorithm on the bbob test suite [1] with restarts for a maximum budget equal to $300 \times D$ function evaluations according to [1]. The Python code was run on a Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz with 1 processor and 4 cores. The time per function evaluation for dimensions 2, 3, 5, 10 (instances 1-5) equals 10.8, 28.0, 45.0 and 195.0 seconds respectively.

5 Discussion of the results

5.1 Results of our algorithm

In the following part, some of our obtained results will be shown for $\lambda = n \times 60$.

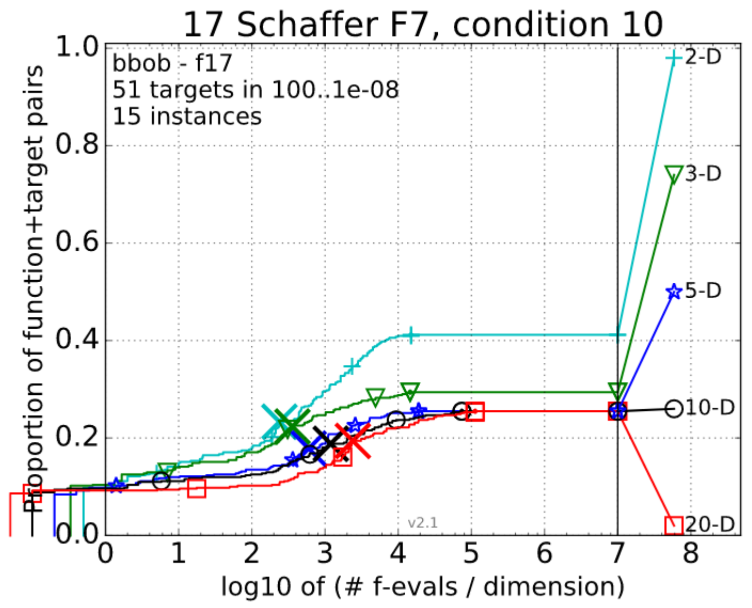

Figure 2: Results of our algorithm, function 17 Schaffer with $\lambda = 60 \times n$

As shown in Figure 2, we can see that we have decreasing performances when increasing dimensions. Indeed, we can note the different asymptote values of proportion of function and target pairs for the function 17 Shaffer :

Dimension	Asymptot value
2	0.41
3	0.32
5	0.28
10	0.28

Most of other functions have similar behaviors, such as the Gallagher function (Figure 3). However, the results may be better depending on the function we focus on.

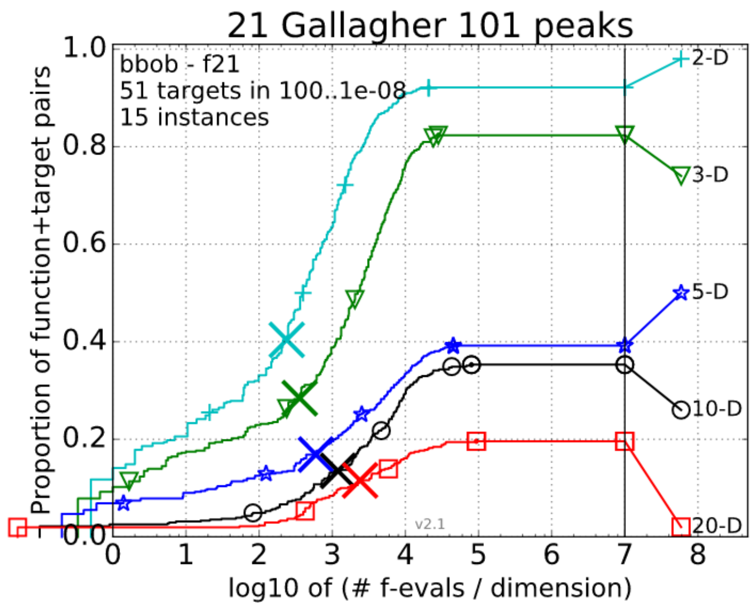

Figure 3: Results for our algorithm, function Gallagher with $lambda = 60 \times n$

Dimension	Asymptot value
2	0.92
3	0.83
5	0.39
10	0.36
20	0.19

However, for some functions such as the Bent Cigar (Figure 4), we observe really low results. This could be explained by the fact that our budget was too low ; we didn't reach the stage where the progress is expected to become asymptotic. In fact, we can observe that in general our budget was too low : we did not reach the predicted asymptot with our computation. We might need to increase the number of iterations.

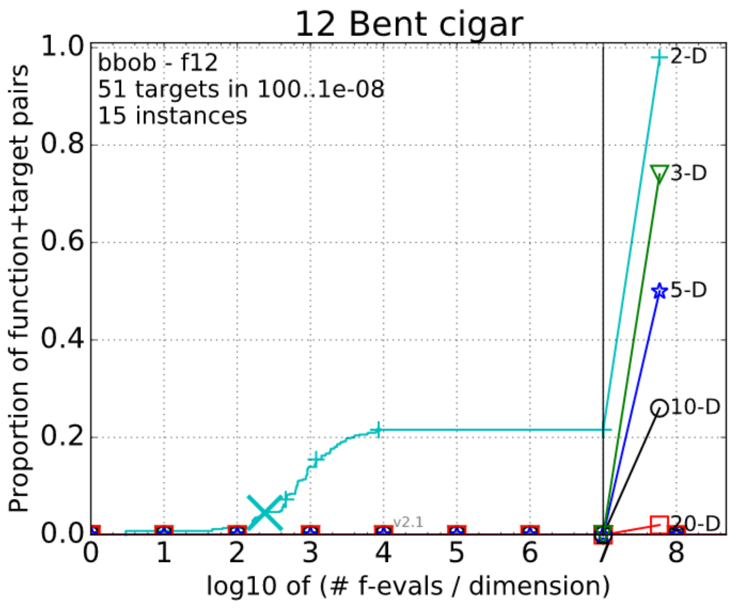

Figure 4: Results for our algorithm, function Bent cigar with lambda=60

Dimension	Asymptot value
2	0.92
3	0.83
5	0.39
10	0.36
20	0.19

We can observe that the results aren't good with a low budget and $\lambda = 60 \times n$. For the next executions we have run the algorithm with higher budget (between $300 \times d$ and $500 \times d$) and for different values of λ : $20n$, $10n$ and $5n$. The parameter which show the best results is $\lambda = 5n$. Let's see some graphs with those new parameters.

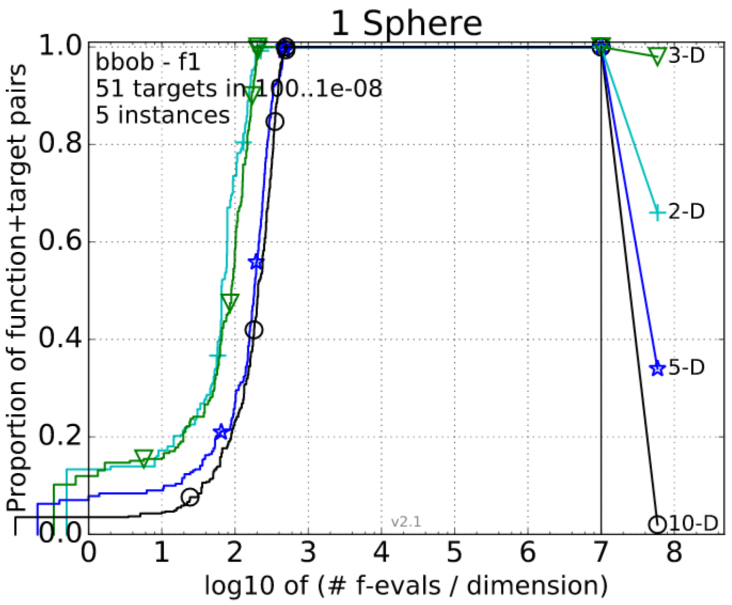

Figure 5: Results for our algorithm, function Sphere with $lambda = 5 \times n$ and $budget = 300n$

Dimension	Asymptot value
2	1
3	1
5	1
10	1

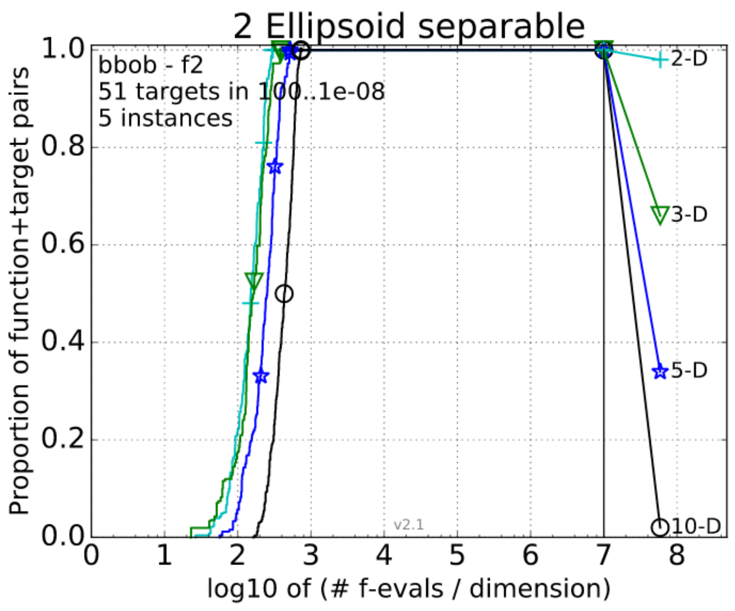

Figure 6: Results for our algorithm, function Ellipsoid with $lambda = 5 \times n$ and $budget = 300n$

Dimension	Asymptot value
2	1
3	1
5	1
10	1

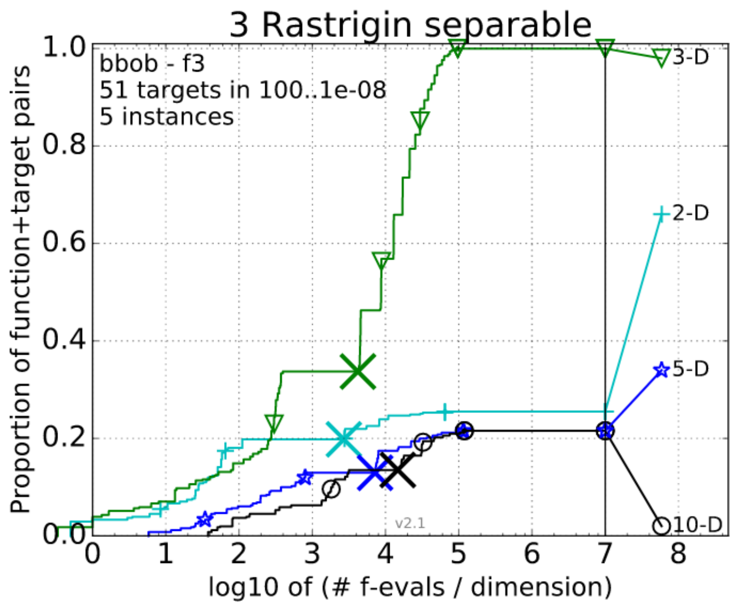

Figure 7: Results for our algorithm, function Rastrigin with $lambda = 5 \times n$ and $budget = 300n$

Dimension	Asymptot value
2	0.2
3	0.34
5	0.15
10	0.15

We see that function Sphere and function Ellipsoid are solved for dimensions 2, 3, 5 and 10 with a sufficient value for budget. Rastrigin function seems to be more tricky for our algorithm, and we can observe that it does better for 3 dimensions than for 2 dimensions on that problem.

5.2 Comparison with other algorithms

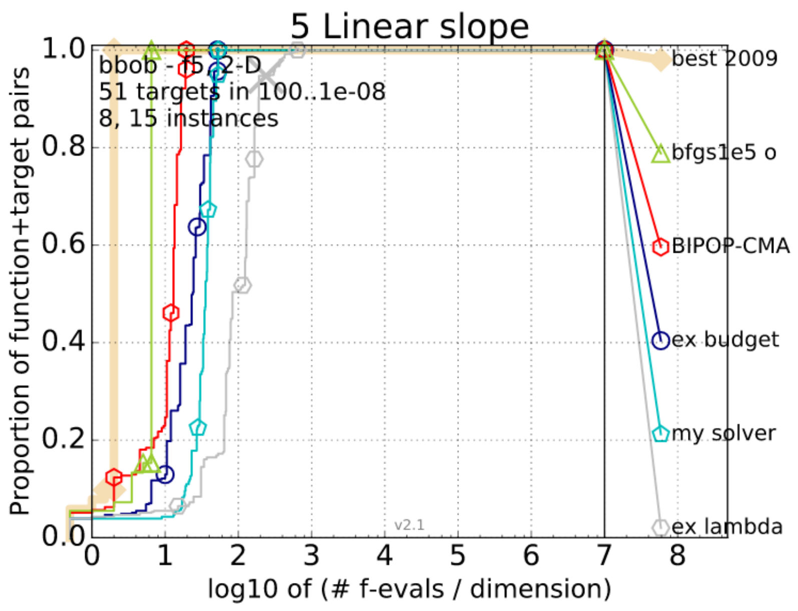

Figure 8: Function 5 dimension 2 for comparison

- ex budget : our algorithm, lambda 5*n, high budget (best version)
- ex lambda : our algorithm, lambda 60*n, low budget
- Best 2009 : Automatic by Coco (best optimizer ?)
- bfgs1e5 : BFGS one
- BIPOP-CMA : BIPOP-CMA
- my solver : other's group algorithm, budget max

We compared our data with two algorithmes : BIPOP-CMA-ES and BFGS.

The CMA-ES algorihm is a stochastic blackbox search template to minimize $f : R^n -> R$. We initialize distriuion parameters θ, set the population size $\lambda \in N$ and while not happy do :

- Sample distribion $P(x|\theta) -> x_1, ..., x_\lambda \in R^n$
- Evaluate $x_1, ..., x_\lambda$ on f
- Update parameters $\theta <- F_\theta(\theta, x_1, ..., x_\lambda, f(x_1), ..., f(x_\lambda))$

In general, sample disributions are multivariate Gaussian distributions.

The BIPOP-CMA-ES is a BI-population Covariance Matrix Adaptation Evolution Strategy.

In BIPOP-CMA-ES, after the first single run (when at least one of stopping criteria is met) with default population size, the CMA-ES is restarted in one of two possible regimes by taking into account the budget of function evaluations spent in the corresponding regime. Each time the algorithm is restarted, the regime with smallest budget used so far is used. [3]

The BFGS algorithm is a Quasi-Newton Method. The key idea of Quasi Newton is to successively iterate x_t, x_{t+1} and gradients $\nabla f(x_t), \nabla f(x_{t+1})$ yield second order information : $q_t \approx \nabla^2 f(x_{t+1} p_t$

[3]https://hal.inria.fr/hal-00818596/document

where $p_t = x_{t+1} - x_t$ and $q_t = \nabla f(x_{t+1}) - \nabla f(x_t)$

Looking at the performances, we see that our algorithms is often outperformed by other algorithms. Comparing our result to the bests algorithms, best 2009 and BIPOP-CMA, which are almost always far closest to the optimal solution than us, we see that the gap of performances between them and us is correlated (most of the time) with the dimension of the problem.

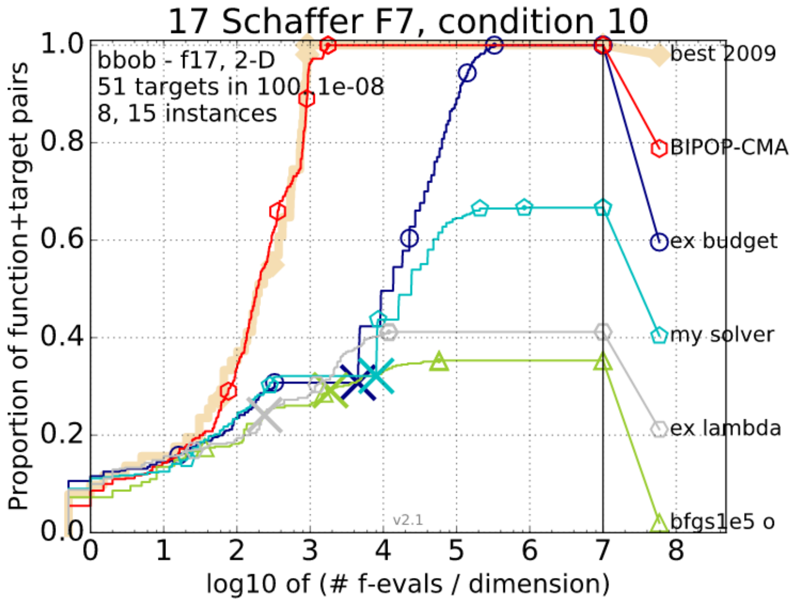

Figure 9: Example, 2 dimensions

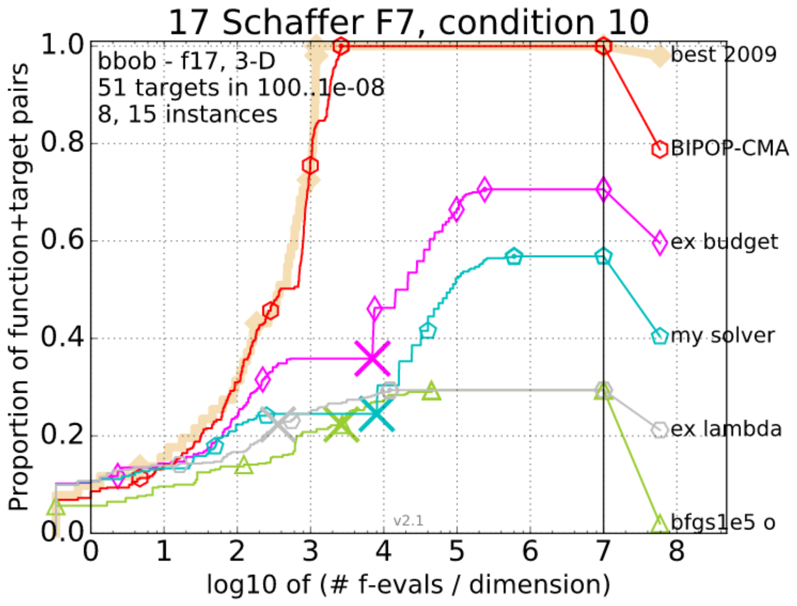

Figure 10: Example, 3 dimensions

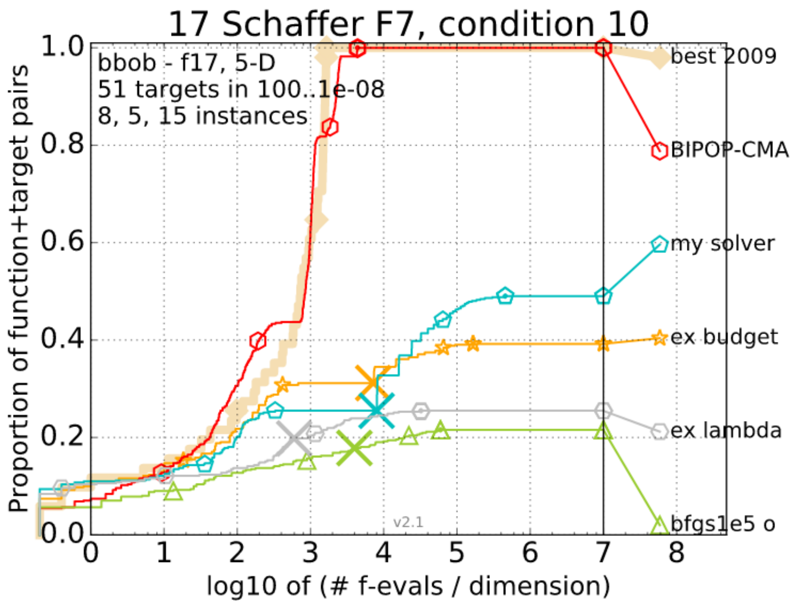

Figure 11: Example, 5 dimensions

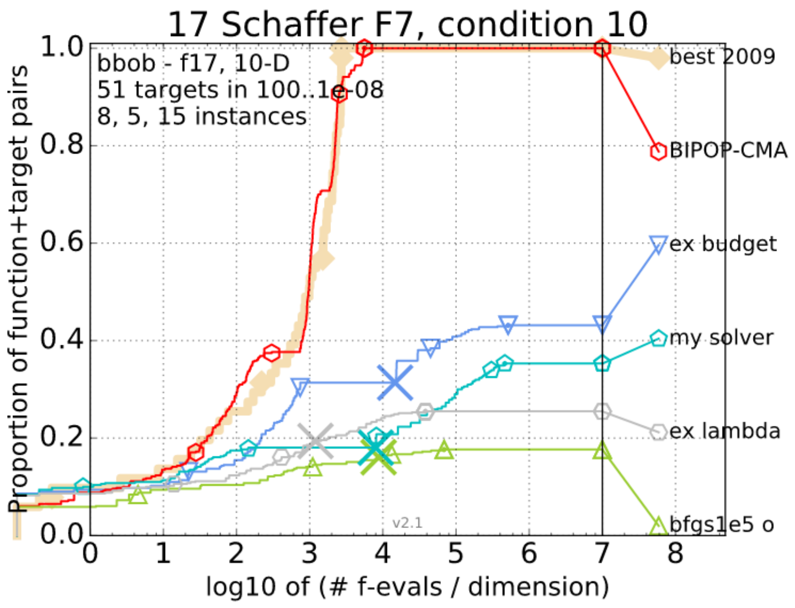

Figure 12: Example, 10 dimensions

While we often hit values between 0.4 and 1 in the lower dimensions, we rarely overstep 0.2 in the 20 dimensions optimization problems. We didn't try the 40 dimensions for time reasons, but we can extrapolate and suggest that we would be way behind the others, more performing algorithms.

6 Conclusions

We saw that, while our algorithm has the advantage of giving good results in low dimensions problems, and being relatively quick and simple to implement and to run, it is less efficient compared to the others.

To improve further the performances of the algorithm, we could adress the problem of the noise occuring in the selection process : despite good paramaters, the randomisation aspect can cause good parents to have bad offspring, or the other way around ; this happens enough frequently to be significant. We could de-randomise the algorithm to avoid this phenomenom as suggested in the article in the Algorithm 3.

To increase performances of our algorithm, we can increase the bugdet. We saw in the different results that the proportion of function and target pairs is estimated by COCO to be increasing for a higher budget. We have also seen that the size of the offspring λ population impacts our results. So maybe we could find a optimum λ or at least, observe the different results with a budget and other parameters fixed in order to increase the performances.

To conclude, our results seem to be consistent with the article : in fact when we compared our performances with other groups, our results were not always as good as their, but they had implemented improved versions of our algorithm, so it seems coherent.

References

[1] Nikolaus Hansen, Dirk V. Arnold and Anne Auger
Evolution Strategies