Tutorial 2: Zero-Order optimization algorithms

E. Ruffio[1], D. Saury, D. Petit, M. Girault[2]

Institut P', UPR 3346, CNRS, ENSMA, Université de Poitiers
Département fluides, thermique, combustion, axe COST
ENSMA - BP. 40109, 86961 Futuroscope Chasseneuil. France
[1] emmanuel.ruffio@ensma.fr
[2] manuel.girault@ensma.fr

Abstract. This workshop aims at presenting some optimization algorithms often used in literature, that is two local search algorithms – a gradient-based and the Simplex method –, an evolutionary algorithm – "Evolution Strategies" (ES) – and a swarm intelligence-based algorithm – "Particle Swarm Optimization" (PSO). Each algorithm is applied on the Rastrigin function and compared to the Random Search Algorithm (RSA) using a statistical approach.

An implementation of ES and PSO is proposed in appendix which uses the Matlab programming language. Moreover, they share a common interface so that it is quite straightforward to switch from one algorithm to another.

1. Introduction

Nowadays, optimization has become a widely used term. The need of optimized products or processes prevails in many sectors of industry or academic research. One tries to optimize manufacturing processes, to improve airport ground traffic, to maximize engine efficiency, to minimize heat losses from buildings, to lighten mechanical structures while keeping their mechanical performances constant, to improve the frequency response of filters, or to optimize optimization algorithms (meta-optimization). In inverse problems, one tries to estimate thermal or mechanical parameters, to reconstruct boundary conditions whose direct measurement is hardly possible, to build reduced-order models for control and regulation of physical quantities, etc.

The purpose of this workshop is to introduce some optimization algorithms from the user point of view. Some key principles of each algorithm will be presented. They are helpful to select the proper algorithm that should be able to solve the optimization problem one is faced with.

This document is structured as follows. An overview of different types of optimization problem is provided. The next part is an overview of optimization algorithm. Various criterions used to classify them are introduced and basic concepts of metaheuristics are presented. Due to their stochastic nature, these metaheuristics cannot be compared easily. A paragraph is therefore devoted to two ways of comparing their efficiency. The next part presents the optimization problem (the Rastrigin function) that will be used to test the algorithms presented at the end of this document, that is:

- Random search algorithm (RSA)
- Local search algorithms: gradient-based algorithm (GRAD) and simplex method (SIMPLEX)
- Global search algorithms: Particle Swarm Optimization (PSO) and Evolution Strategies (ES).

2. Optimization problem

2.1. *Single and multi-objectives problem*

An optimization problem must be expressed as a function minimization problem[1] first. Then, solving the problem consists in finding the solution vector β that minimizes a function $J(\beta)$, called objective function or functional or optimality criterion for example. This document uses the term « criterion » to denote $J(\beta)$. This criterion is usually a real-valued function that takes a real-valued vector[2] as its input-argument. The criterion is a measure of how close we are from an objective. Since the criterion returns a single real value, this kind of problem is designated by « single-objective optimization problem ». The search space is defined in this workshop as the space that contains all real-valued vectors that meet the constraints of the problem. Let β_1 and β_2 be two candidates, i.e. two possible solutions. In practical terms, β_1 and β_2 are just two vectors of the search space. For single-objective optimization problems, the comparison operator is obvious: the candidate β_1 is better than the candidate β_2 if $J(\beta_1) < J(\beta_2)$.

If the criterion is now a vector-valued function, one is faced with a « multi-objective optimization » problem. Each component of $J(\beta)$ corresponds to an objective (a criterion) that must be optimize simultaneously with the others. In such problem, the uniqueness of solution is not guaranteed. Instead, one defines the « pareto front » as the set of solutions that cannot be improved further without degrading at least one component of $J(\beta)$. In other words, this set contains all « non dominated » solution. Indeed, when comparing two candidates, there are now 3 possible cases:

- $J(\beta_1)$ dominates[3] ($\approx$ is lower than) $J(\beta_2) \Longrightarrow \beta_1$ is better than β_2
- $J(\beta_2)$ dominates $J(\beta_1) \Longrightarrow \beta_1$ is worse than β_2
- $J(\beta_1)$ and $J(\beta_2)$ are not comparable $\Longrightarrow \beta_1$ and β_2 are not comparable

2.2. *Multimodal, dynamic*

Other kinds of optimization problem exist [1 p.171]. In multimodal problems, one is not only interested in the solution but in all (or a part of) local minimums. Lastly the user chooses a satisfying solution among these solutions, possibly based on a subjective appreciation or other constraints not modeled in the problem. For dynamic problems, the complexity comes from the fact that the criterion changes every time step. Therefore, algorithms have to find the solution and to "follow" it.

2.3. *Constraints*

The criterion takes a real-valued vector β as input parameter. Its components could be subject to various constraints (Tab.1). These constraints can be either "soft" or "hard". Hard constraints denote conditions that must not be violated by the algorithm. Hard constraints are the most difficult to handle and are unfortunately quite common. For example, if one parameter of β is the position x of a thermocouple in a sample whose width is denoted by e, then x must lie in [0,e].

Soft constraints mean "preferences" rather than "constraints" and prevent the algorithm to go in some areas of the search space. A fairly common and obvious approach is to use "penalization" [1 p.207]. It consists in assigning a penalty cost to constraint violations and to add it to the criterion.

---

1 Without loss of generality, an optimization problem is equivalent to a minimization problem. Indeed, a maximization problem consists in minimization the opposite of a function.

2 The input-argument is a real-valued vector only for continuous problem. Combinatorial, discrete or mixed problems (may) use integer-valued parameters.

3 Every components of $J(\beta_1)$ are lower than those of $J(\beta_2)$.

Table 1 : Different kinds of constraint

$A\beta = C$	Linear equality constraints
$f(\beta) = 0$	Non-linear equality constraints
$D\beta \leq E$	Linear inequality constraints
$g(\beta) \leq 0$	Non-linear inequality constraints

3. Optimization algorithm

3.1. *Generalities*

New optimization algorithms are continuously appearing and it has become necessary to classify them in families [2]. This task is quite complex due to the significant number of criterions that can be used: stochastic or deterministic algorithms, local or global search, zero-order, first-order or higher order.

Deterministic algorithms refer to methods whose output depends on the input parameters (intrinsic setting parameters and possibility an initialization vector β^0). Stochastic algorithms are the counterpart of deterministic algorithms. In these algorithms, there is some indeterminacy in the algorithm through the use of random number generators. As a consequence, two successive runs of the same algorithm will not give the same numerical result.

Local search algorithms, like gradient-based or Simplex methods, are generally deterministic. The main drawback lies in the high correlation between the quality of the result and the quality of the initialization. These methods are indeed incapable of exploring the search space and can stay confined in the first local minimum they reach. On the contrary, global search algorithms are mainly stochastic. They use random-based processes to explore the search space in order to find promising areas that might contain the global minimum. Random processes are not only used for "exploration" (or "diversification") but for "exploitation" (or "intensification") as well, i.e. to improve candidates of the current iteration.

An other criterion is based on the information exploited by the algorithm to direct the search procedure. Zero-order (or derivative-free) algorithms use only the criterion value at some positions. First-order methods require the gradient with respect to all parameters to be computed and second-order methods exploit the Hessian matrix, i.e. the second derivatives of the criterion. Zero-order algorithms have become an active field of research. One of the interesting features is the lack of strong assumptions on the criterion, like continuity or differentiability. However, when the use of a local search method (especially gradient-based) is possible[4], one will hardly find faster algorithms among stochastic ones. The efficiency of local search algorithms is often unmatched.

3.2. *Metaheuristics*

Several new optimization algorithms have emerged [3] in the past few decades that mimic biological evolution, or the way biological entities communicate in nature. These algorithms were designed to solve hard optimization problems. Hard means roughly that there is no known algorithm able to solve the problem considered in a reasonable time. Instead of finding the exact solution of the problem, one has content oneself with satisfying solutions that could possibly be far from the exact mathematical minimum. Moreover, without the huge developments of the computational capacity of computers during the end of the last century, the use of those algorithms would not have been possible.

[4] The criterion is continuous, differentiable and does not have (too many) local minimums.

Before going further in the description of metaheuristics, the term "heuristic" must be defined. A heuristic is a rule of thumb based on intuition or experience for problem solving. Heuristics are used to speed up the process of finding satisfying solutions, where an exhaustive search is impractical. However, heuristics should not be seen as procedures used to find the solution, but rather as procedures that increase the chance of finding a good solution.

For example, the greedy algorithm is a stepwise and iterative procedure based on the greedy heuristic. At each iteration, the next candidate is chosen, out of all neighbors of the current candidate, in a way that maximizes the immediate gain. Gradient-based methods are based on such behavior since they use the steepest descent direction to choose the next candidate. It corresponds to the search direction where the immediate gain should be the highest.

A second example intuitively used by everybody to pack odd-shaped items into a box. It tells: "start with the largest items and then fit the smaller items into the spaces left". By using this rule, one increases the probability of being able to put all objects in the box.

The last example presents a heuristic used implicitly by every metaheuristics. Let β_1 and β_2 be two candidates whose criterion values are respectively $J(\beta_1)=5$ and $J(\beta_2)=10$. Where should we start looking for the solution? Without additional a priori information, intuition suggests to search around the current best candidate β_1. In other words, "start by looking for the solution around the best candidates found until now".

As their names imply, metaheuristics [1, 2] are an abstraction of heuristics (however, there is no commonly agreed definition of metaheuristics). They are algorithms based on several simple heuristics. Some of them are used to explore the search space while others try to exploit the current promising candidates. "Exploration" and "exploitation" [2] are two of three key principles of metaheuristics. The third is the "memory" or "experience", that is the ability to use current candidates and possibly older candidates to guide the search. Furthermore, lots of metaheuristics share the following features:

- **approximate:** one is not trying anymore to find the exact solution but only satisfying solutions. However, there is no guarantee the algorithm will success.
- **zero-order:** generally, metaheuristics use only criterion values to look for the solution: criterion derivatives are not necessary.
- **global:** metaheuristics are less sensitive to local minimums than local search algorithms. However, it is still possible that they get stuck in a local minimum. Some metaheuristics are proven to be able to find the solution but the time needed is often unaffordable.
- **stochastic:** random-based processes, through the use of random number generators, is central.
- **iterative:** like most optimization algorithms, metaheuristics are iterative, i.e. they improve step by step (or iteration after iteration, or generation after generation) a candidate or a set (a population) of candidates.
- **single-objective:** the canonical version of metaheuristics is generally dedicated to single objective optimization.
- **population-based:** lots of them do not use only one candidate that would be improved iteratively. Instead, a set of candidates is improved iteratively. For example, genetic algorithms (GA) [1] use a population of 10 up to several hundred candidates (individuals). For PSO, the swarm contains generally from 20 to 30 candidates (particles).
- **parallel processing:** most of metaheuristics are very suitable for parallel computing since candidates are independent from each other and therefore, they can be computed on dedicated processors.

Depending on the situation, those characteristics can be interesting. Since they are zero-order algorithm, there is no use to implement subroutine dedicated to the derivatives computation. The criterion does not have to be continuous nor differentiable. Therefore, only few assumptions are done on the function shape. However, this shape could have a significant influence on the algorithm efficiency, in terms of number of criterion evaluations needed to find satisfying solutions.

3.3. *Comparison of stochastic algorithms*

The two metaheuristics presented in the following, PSO and ES, share all features mentioned above. The use of random events makes their comparison more intricate. One cannot rely on few executions of both algorithms to draw any conclusions about their efficiency. The approach used in this workshop consists in estimating the statistical distribution of outputs of each algorithm. However, contrary to local search method, the concept of convergence is not straightforward to define for metaheuristics since one never knows if a better candidate could be found elsewhere in the search space. Consequently, a common termination criterion consists in limiting the number of iteration which is nearly equivalent to limit the number of criterion evaluations[5].

Let N be the maximum number of evaluations, the probability density associated to the output z of an algorithm A is denoted by $D_{A,N}(u)$. Let y be a criterion value, the probability for the algorithm to give a value lower than y is:

$$p_N(z < y) = \int_{-\infty}^{y} D_{A,N}(u)\, du = C_{A,N}(y) \tag{1}$$

With $C_{A,N}(y)$ the cumulative probability density.

A first easy criterion could be obtain from percentiles: let N be the number of evaluations allowed, the algorithm A is considered to be more efficient than the algorithm B if the 95-percentile[6] of A is lower than the 95-percentile of B. That is if:

$$C_{A,N}^{-1}(\mathrm{p} = 0.95) < C_{B,N}^{-1}(\mathrm{p} = 0.95) \tag{2}$$

Clerc[7] suggests a different approach. It consists in evaluating the probability for the output of A to be lower than the output of B. Let x_A and x_B be the output of A and B respectively. By integrating (Eq.1) over all possible value y, it comes:

$$p_N(x_A < x_B) = \int_{-\infty}^{\infty} C_{A,N}(u)\, D_{B,N}(u)\, du \tag{3}$$

In other words, if $p_N(x_A < x_B) > 0.5$, it means that most of the time, A gives better solutions than B. If A and B denote the same algorithm, the obvious result is that $p_N(x_A < x_B) = \frac{1}{2}$.

Both criterions are not equivalent. The first one just give the threshold below which one can reasonably hope the result to be. But if A is better than B based on this criterion, it does not give any information about the probability that A is lower than B.

---

[5] The number of criterion evaluations is the number of times that the criterion subroutine is called.

[6] The 95-percentile of a sample is the value that split the samples it two parts: the first contains 95% of data while the second part contains the remaining 5%. The sample median is equal to the 50-percentile.

[7] Some notes of Clerc about the comparison of stochastic algorithms can be downloading with the following link (04/05/2011) http://clerc.maurice.free.fr/Maths/compare_algo/compare_algos.pdf

4. The Rastrigin function

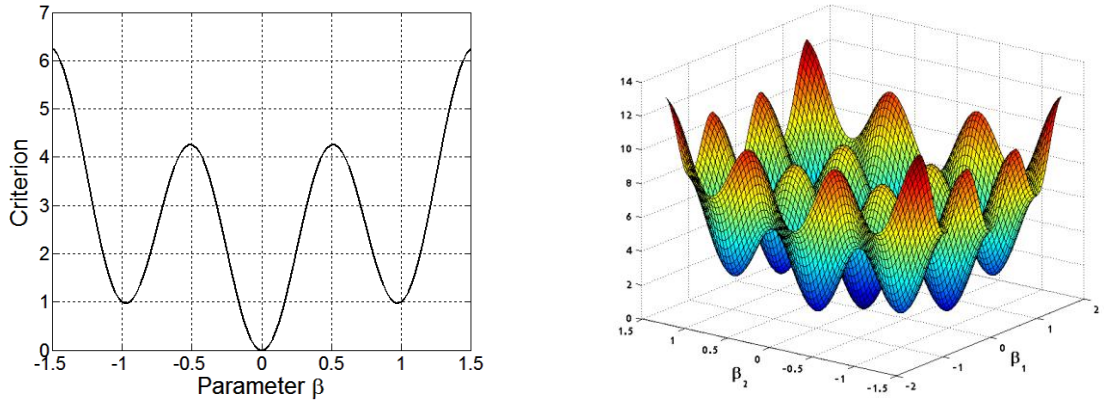
In this workshop, a continuous and single-objective problem is used to test all algorithms presented in the following. The criterion is thus a function from R^N to R. Moreover, only boundary constraints are considered. The optimization problem consists in finding β so that J(β) is minimized. The solution is then:

$$\hat{\beta} = \arg\left[\min_{\beta}\left(J(\beta)\right)\right] \qquad \text{avec } \beta_i \in [a_i; b_i] \tag{1}$$

The criterion J that must be minimized is the Rastrigin function. It is one of the famous test functions used to evaluate algorithm efficiencies. In the general case, it is defined by:

$$J(\beta) = A\,N + \sum_{i=1}^{N}\left[\beta_i^2 - A\cos(2\pi\beta_i)\right] \tag{2}$$

With A a real-valued parameter and N an integer corresponding to the dimensionality of the problem. In this workshop, with $\beta_i \in [-1.5; 1.5]$, $A=2$ and $N=2$. The minimum of $J(\beta)$ is 0. It is reached for $\beta = 0$ whatever is the dimensionality. The Figure 1 exhibits the 1d-Rastiring and 2d-Rastrigin function. All algorithms in this workshop are applied to the 2D-Rastrigin function. A contour map of the 2d-Rastrigin function is shown in Figure 5.a. The global minimum in $\beta = 0$ and the eight local minimums are visible.



a) 1d-Rastrigin b) 2d-Rastrigin

Figure 1. 1d and 2d-Rastrigin functions.

5. Random search algorithm (RSA)

The random search algorithm [37] is an approximate, global, stochastic and zero-order algorithm. It can be considered as one of the simplest and/or the most stupid algorithm. It consists in evaluating the criterion of randomly drawn candidates and in keeping in memory the best candidate. Since there is no reason to favor areas of the search space, uniform density distributions are used to generate candidates.

Where is the point of using such an algorithm? First, this kind of approach was hardly conceivable until the development of computational power of computers. The RSA ends up being often quite efficient especially when coupled with a local search algorithm. By exploring the search space, the RSA algorithm rejects unpromising areas of the search space and avoids the local search of being stuck in bad local minimums.

Furthermore, the RSA provides a reference in terms of performance and efficiency. One could expect metaheuristics to give better results than RSA. The probability for metaheuristic output being

lower than the RSA's output should be as close as possible of 1. To evaluate this probability with (Eq.3), the probability density function (PDF) D(α) and the cumulative probability density function (CDF) C(α) need to be computed. C(α) can be defined as follows: let $S = (b_1 - a_1)(b_2 - a_2)$ be the surface area of the search space, let α be a criterion value, and R(α) the "subarea of S corresponding to candidates $\beta = (\beta_1, \beta_2)$ for which $J(\beta_1, \beta_2) < \alpha$", C(α) is then:

$$\mathrm{C}(\alpha) = p(x < \alpha) = \frac{R(\alpha)}{S} \tag{10.a}$$

$$\mathrm{D}(x) = \frac{d}{dx}\mathrm{C}(x) \tag{10.b}$$

$$\text{with} \quad \begin{cases} R(\alpha) = \int_{u=a_1}^{b_1} \int_{v=a_2}^{b_2} I_\alpha(u,v)\, dv\, du \\ I_\alpha(u,v) = \begin{cases} 1 \text{ si } J(u,v) < \alpha \\ 0 \quad \text{sinon} \end{cases} \end{cases}$$

With $a_1 = a_2 = -1.5$, $b_1 = b_2 = 1.5$ and x a realization of the experience which consists in evaluating the criterion with a randomly drawn candidate.

In the case of the 2D-Rastrigin function, PDF and CDF functions and the second order development of the CDF around *α=0* are shown in Figure 2. This development is computed with the second order development of J(β) around the solution *β=(0, 0)*. This development is fairly accurate as long as the criterion is locally equivalent to a paraboloid of revolution. Around *α=1*, *α=2*, *α=8.5* and *α=10.5*, discontinuities exist due to local minimums and local maximums. Around these points, the PDF is indeed not continuous. However, due to the numerical approach and to the method employed (kernel density methods) to estimate the PDF and CDF function, discontinuities are not smoothed.

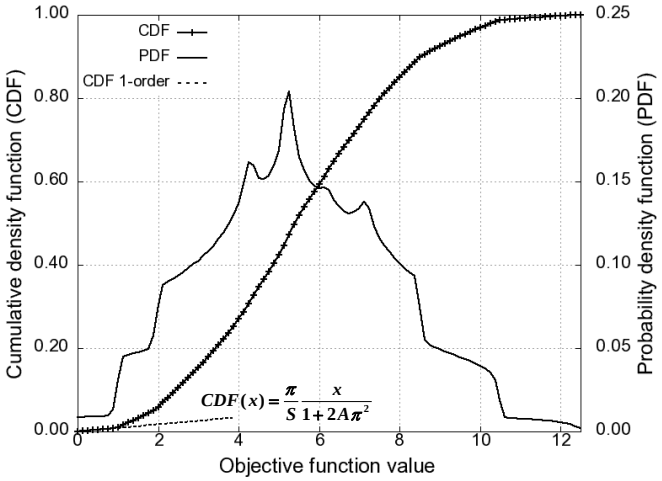


Figure 2. CDF C(x) and PDF D(x) functions with respect to criterion function value (denoted *x* in the figure) and the first order development of C(x) around x=0.

Figure 2 gives the distribution of the criterion values (PDF) if one draws only one candidate. If however, one is allowed to draw p samples and to store the best candidate found, what are the associated statistic laws? The stochastic process associated to the RSA can be expressed as follows: "among p random samples, what is the probability that at least one sample is lower than a specified value, denoted by α". Let $R_P = \{\omega_i, 1 < i < p\}$ be a realization of this stochastic process where ω_i denotes one draw of the criterion. The PDF $D_p(\alpha)$ and CDF $C_p(\alpha)$ of the random variable R_p can be expressed using $D(\alpha)$ and $C(\alpha)$:

$$
\begin{aligned}
C_p(\alpha) &= P\left\{(F(\omega_1) < \alpha) \text{ ou } \cdots \text{ ou } (F(\omega_P) < \alpha)\right\} \\
&= 1 - P\left\{(F(\omega_1) \geq \alpha) \text{ et } \cdots \text{ et } (F(\omega_P) \geq \alpha)\right\} \\
&= 1 - \prod_{i=1}^{P} P\{F(\omega_i) \geq \alpha\} \\
&= 1 - [1 - C(\alpha)]^p \qquad (11.a)
\end{aligned}
$$

$$
D_p(\alpha) = p\, D(\alpha)\, [1 - C(\alpha)]^{p-1} \qquad (11.b)
$$

Based on these two expressions, $C_p(\alpha)$ and $D_p(\alpha)$ can be computed for any value of p (Figure 3). As p increases, $C_p(\alpha)$ is moved to the left. In other words: as p increases, the probability for a randomly drawn value being lower than a specified threshold increases too. Functions $D_p(\alpha)$ are moved to the left as p increases, meaning that the outputs of the stochastic process are more and more confined to low values of the criterion. For example, for $p=200$, one is nearly certain that the best candidate would be lower than 1. Since the criterions of local minimums are greater than 1, the exact solution $\beta=(0, 0)$ can be found with a high probability if the local search method is applied on the best candidate found by the RSA with $p=200$ random evaluations.

The continuous lines in Figure 3 refers to the second order development of $C_p(\alpha)$. Numerical estimations and analytical approximations are in good agreement which suggests numerical errors are not significant. Indeed, one should be aware that the precision of $C_p(\alpha)$ declines as p increases.

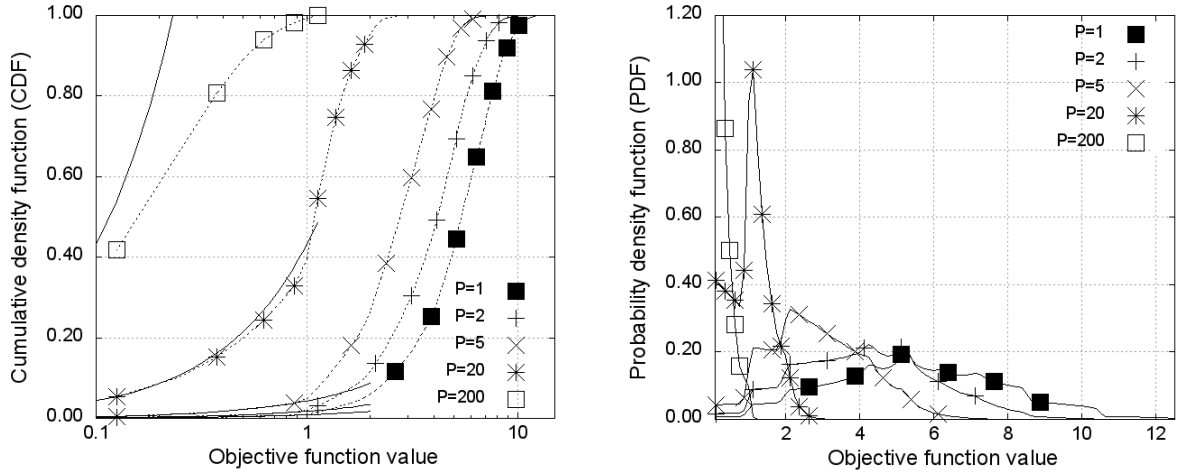


Figure 3. CDF and PDF functions with respect to the number of random samples in $\mathrm{R_p}$.

6. Local search algorithms

6.1. *Gradient-based*

Gradient-based algorithms [12,13] are well-known and common algorithms. During the Metti 5 school, a lecture devoted to these algorithms is given by Favennec [14]. They are local search, iterative and deterministic algorithms. One of their main features lies in the fact they use the criterion derivatives to find the minimum. It is in the same time the source of their efficiency and their main weakness.

The gradient (first derivatives) points out the steepest descent direction, i.e. the direction in which the decrease rate is maximized. Algorithms that use only the gradient are called "first-order algorithms". The recurrence relation between a candidate at iteration k and k+1 is:

$$\beta^{k+1} = \beta^k - \mu^k \nabla J(\beta^k) \quad (6)$$

μ^k is the step size at iteration k. Several approaches exist to choose this value. A common one consists in finding μ that minimizes the 1d-function $f(\mu) = J(\beta^k - \mu \nabla J(\beta^k))$.

An algorithm that uses the second derivatives – or Hessian – is refers as "second-order algorithm". The Hessian greatly speeds up the convergence as the algorithm comes close to the solution. The gradient reveals the search direction but gives no information about the distance. In contrast, the use of the Hessian allows the algorithm to localize precisely the minimum if the quadratic approximation of the criterion is correct[8].

For example, assuming the criterion is a positive quadratic function, the knowledge of the hessian and the gradient at one point β is enough to find the minimum with only one iteration. Let $J(\beta)$ denotes the criterion that is nullified for $\beta = x_0$:

$$J(\beta) = (\beta - x_0)^T A\,(\beta - x_0) \quad (7.a)$$

$$\frac{\partial J}{\partial \beta}(\beta) = 2A\,(\beta - x_0) \quad (7.b)$$

$$\frac{\partial^2 J}{\partial \beta^2}(\beta) = 2A \quad (7.c)$$

By definition, x_0 is unknown but it can be expressed using only the Hessian, the gradient and β:

$$\left[\frac{\partial^2 J}{\partial \beta^2}\right]^{-1} \frac{\partial J}{\partial \beta} = \beta - x_0 \quad \Longrightarrow \quad x_0 = \beta - \left[\frac{\partial^2 J}{\partial \beta^2}\right]^{-1} \frac{\partial J}{\partial \beta} \quad (8)$$

In the general case, the criterion is not quadratic, but this expression becomes valid near the solution. For this kind of algorithms, the recurrence relation between the iteration k and k+1 is:

$$\beta^{k+1} = \beta^k - \mu^k A^k \nabla J(\beta^k) \quad (9)$$

A_k denotes the Hessian matrix or an approximation.

The Hessian computation is often time consuming so that second-order method are hardly ever used. One has to find a way to simplify the Hessian computation. It leads to "pseudo second-order" methods[9]. Instead of evaluating the Hessian matrix, one has to settle for an approximation of it. At least two ways exist to approximate the Hessian. The first one is used by the Quasi-Newton algorithm while the Levenberg-Marquardt algorithm uses the second one. With the first method, the Hessian matrix is computed using gradient evaluation of previous iterations. More or less complex relations can be used for this (SR1, Broyden, FDP, BFGS[10]). Concerning the Levenberg-Marquardt algorithm, it is dedicated to least-squares based criterion only since the Hessian matrix is approximated based on the sensitivities of the corresponding inverse problem. The sensitivities are the derivatives of the model outputs with respect to the unknown parameters (that has to be estimated).

---

[8] The nearest the algorithm is of the solution, the best the quadratic approximation is and therefore the fastest the algorithm reaches the solution.

[9] Matlab has several gradient-based methods design to solve minimization problems: "fmincon" for non-linear and constraint (linear, non linear) optimization problem, "fminunc" for unconstraint non linear problems.

[10] The Broyden-Fletcher-Goldfarb-Shanno formula (BFGS) is today widely used due its efficiency.

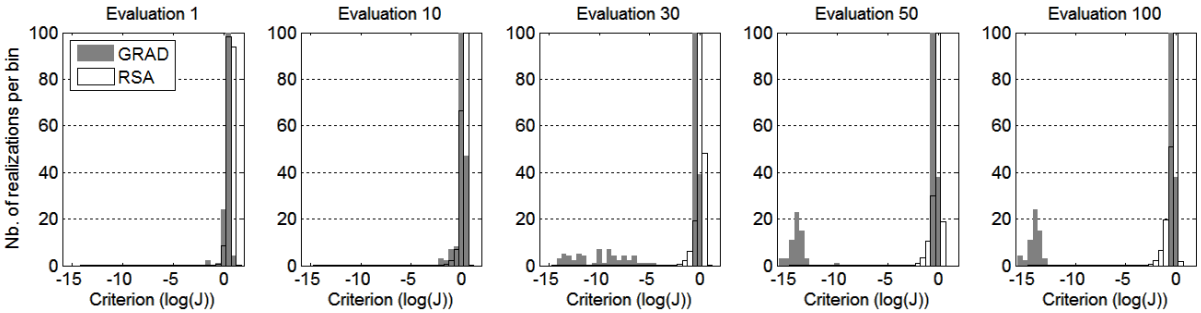


Figure 4. Output distribution of GRAD algorithm with respect to the number of evaluations compared to RSA.

Like all algorithms presented in the following, the efficiency of a gradient-based method is compared to RSA. The implementation used is the "fmincon" function of Matlab. This algorithm is denoted GRAD. To perform a statistical analysis, *n=200* independent executions are performed with randomly drawn initialization in $[-1.5, 1.5] \times [-1.5, 1.5]$. The criterion is the 2d-Rastrigin function. For all executions of the algorithm and for all calls of the criterion function, the best value ever found and the corresponding number of evaluations are stored in memory. The number of evaluations takes into account of those used to estimate the gradient by forward finite differences.

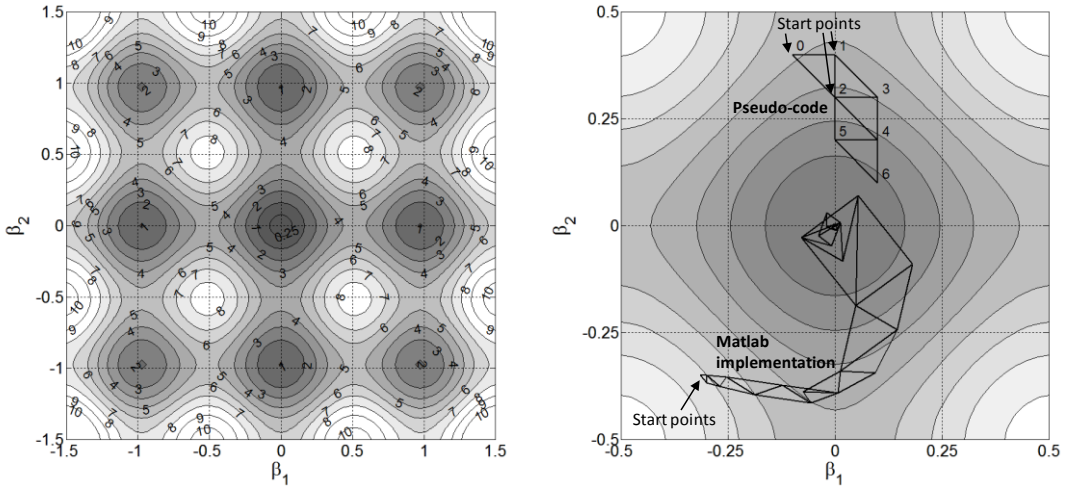
Using these 2-tuples – criterion values and number of evaluations -, one can plot histograms that show the distribution of outputs for a fixed number *n* of evaluations (Figure 4). The titles of each subfigure define the number of calls to the criterion. In the first subfigure, the distributions of GRAD and RSA should be quite similar since they have in theory the same distribution. The first evaluation – the initialization – is indeed random so that the initialization follows the same distribution shown in Figure 2. Significant differences appear for *n>30*: the algorithm is currently converging to the solution. For *n=50*, it appears the convergence ended since the additional 50 evaluations do not change the distribution.

Moreover, local minimums appear to be a serious problem: only 22% of executions found the global minimum. The others got stuck in one of the eight local minimums. However, such a poor result is still higher than the theory. Based on Figure 5.a, the contour map shows that the 2d-Rastrigin functions can be split into nine equal parts whose half edge is roughly 0.5. Therefore, the probability for the initialization being in the middle part is 1/9=11.1%. In theory, if the gradient-based algorithm is initialized outside the "central valley", it should not be able to find the global minimum. The success rate depends only of the probability to reach the "good" valley during the initialization. How to explain such a high success rate obtained with the Matlab implementation? It seems that the implementation does not only use the initialization point but try few different start points, based on boundary constraints so that the algorithm increases its chances of success.

6.2. *Simplex*

The Simplex method [10,17] has emerged in '60s. It is a local search, deterministic, iterative and zero-order algorithm. The expression "simplex" refers to a generalization of the triangle in arbitrary dimension. A triangle has 3 vertices in 2-dimensional space. A simplex has M+1 vertices in an M-dimensional space. The key ideas behind this algorithm are fairly simple but its implementation can be quite tricky especially with latest variants[11]. It does not use the gradient but it approximate the dominant trend of the criterion at a specified point by using the criterion values at each vertices.

[11] For Matlab users, the simplex algorithm is implemented in the function « fminsearch ».



a) Contour map of the 2d-Rastrigin function.

b) Contour map around the solution β=(0, 0) and simplex trajectories.

Figure 5. First iterations of the Simplex algorithm. Comparison of the pseudo-code executed up to the fourth iteration and the Matlab implementation of the Simplex algorithm (« fminsearch »).

In order to appreciate the path taking by the algorithm in the search space, a simplified pseudocode of the algorithm was implemented in Matlab:

1) Simplex initialization: The parameter vectors of each vertex are defined. (Vertices numbered 0-1-2 in Figure 5.b). The criterion is then evaluated for these three points.
2) Iterate until the difference between criterions of each vertex is below a specified threshold.
3) Replace the worse vertex by the symmetric with respect to the remaining vertices. During the first iteration, the vertex 0 is replaced by the vertex 3 (Figure 5.b). (Then, 1 will be replaced by 4 in the next iteration).
4) Go back to step 2.

It is a simplified pseudocode and cannot be used without special additional rules necessary. Those rules are necessary when the simplex is closed from the solution. The simpler approach consists in reducing the simplex size until a specified threshold is reached. Moreover, the pseudocode above refers to old and quite inefficient variants. Some recent implements, like the one proposed by Matlab, allows the simplex shape to change in order to reduce the number of evaluations needed to converge.

7. Particle swarm optimization (PSO)

7.1. *Introduction*

The Particle Swarm Optimization algorithm [9,15,16] is a method developed by Kennedy and Eberhart in 1995. It is a population-based, natured-inspired, stochastic, derivative-free and global algorithm. It simulates the behavior of a biological social system like a flock of birds or school of fishes in order to exploit the swarm intelligence, i.e., a type of intelligence based on interactions of decentralized, self-organized and unsophisticated systems. The canonical version of this algorithm is rather designed for continuous problems, i.e. for real-valued parameters.

The swarm is modeled by individuals called “particles” that have a position, a velocity and a

memory. Each particle moves around independently in the search space with a certain degree of freedom and randomness, looking for more food, i.e. area where the objective function is better. Each particle has the ability to remember its best position and to share its current performance with other particles of the swarm. Particles iteratively adjust their velocity on the information collected from their neighbors. More precisely, a position vector P_i^k corresponding to the parameter values and a velocity v_i^k are associated to each particle, where i denotes the particle index and k the iteration index. The dimension of the vectors P_i^k and v_i^k is the number of unknown parameters N_β[12]. Initially, their positions and velocities are randomly set to acceptable values. Then, particles are iteratively updated using the following expression:

$$\begin{aligned} v_i^{k+1} &= \omega . v_i^k + c_1 . r_1(\mathrm{i,k}) \otimes \left[\widehat{N_i^k} - P_i^k\right] + c_2 . r_2(\mathrm{i,k}) \otimes \left[\widehat{P_i^k} - P_i^k\right] \\ P_i^{k+1} &= P_i^k + v_i^{k+1} \end{aligned} \qquad (22)$$

With:

$\omega \approx 0.7$: the momentum of particles

$c_1, c_2 \approx 1.4$: maximum importance of each attraction point

$\widehat{N_i^k}$: best experience among the neighbors of the particle i

$\widehat{P_i^k}$: best experience of the particle i

$r_1(\mathrm{i,k}), r_2(\mathrm{i,k}) = U(0;1)$: uniform random vector (dim N_β) for each particle for each iteration

$\otimes$: component wise product operator

Remarks:

1) The values of ω, c_1, c_2 should not be chosen independently and some studies recommended the use of particular combinations. c_1, c_2 are also called respectively social and cognitive acceleration coefficients.
2) $\widehat{N_i^k}$ is the best neighbor of a particle. There are many way to define the neighborhood of a particle. In the canonical version of PSO, the neighborhood is the whole set of particles.
3) Generally, a swarm contains roughly 20 or 30 particles.

PSO is a recent algorithm and is in itself a field of research. The version presented here is the canonical version but many variant exist at every stage of the algorithm.

7.2. *Particle trajectories*

Based on the above input parameters, PSO was run and the particle positions were simultaneously recorded for all the particles and for each iteration (Figure 6). The first iteration corresponds to the initial position of the particles. One can see that particles are not perfectly distributed in the search space. On this example, the particles rush to the local minimum at the top right. But at the fifth iteration, a particle discovered a better candidate, i.e. the local minimum at the top center. But it's still the wrong one. Fortunately for us, a gentle particle wandered around the center and attracts to her every particle of the swarm.

[12] In this workshop, $\beta=(\beta_1, \beta_2)$ so $N_\beta=2$.

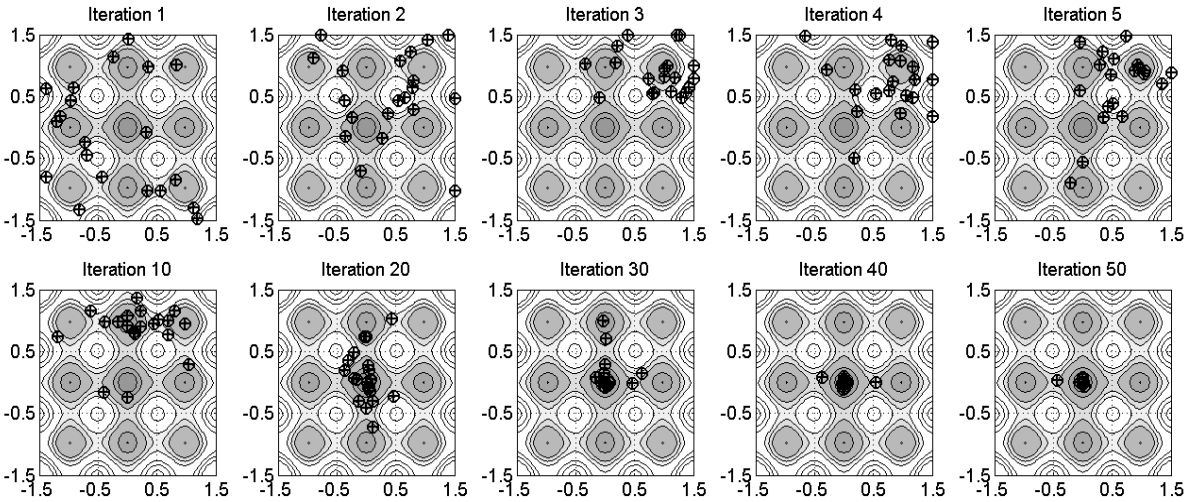


Figure 6. Example of particle positions with respect to the iteration.

The evolution of particle positions exhibited on this figure is typical of all PSO variants but significant behavior can happen depending on the neighborhood and input parameters of the algorithm. A set of parameters could be quite successful on a particular function and leads to disastrous results on another one. When in doubt, it is advisable to use the standard parameters available in the literature.

7.3. *Statistical approach*

In order to assess the efficiency of PSO on the 2D-Rastrigin function, one cannot rely on a single run of the algorithm. Instead, a statistical study must be performed. The approach is identical to the one used for GRAD. For every *n=200* realizations, the criterion is recorded for all iterations. Next, the PDF function is computed (Figure 7). Each subfigure refers to a different termination criterion.

The first one is the distribution of the PSO outputs just after the initialization. Since all particles are randomly initialized, it is not surprising that the distribution matches very well the corresponding distribution of RSA. Since PSO uses a swarm of 20 particles, the RSA distribution is obtained for *p=20* in (Eq.11.b) and by multiplying the result by the number of samples n. In fact, the histogram plot is built from samples of *log(J)* and not *J*. Therefore, an additional transformation is applied on (Eq.11.b).

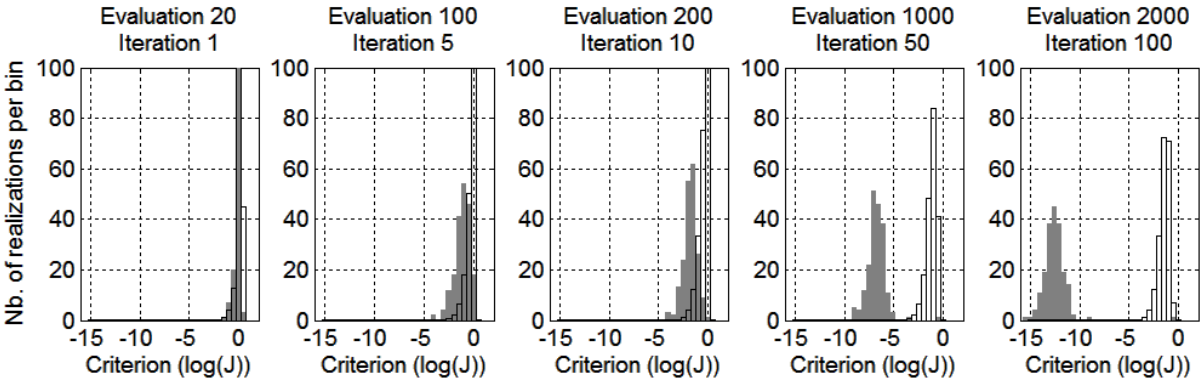


Figure 7. Distribution of PSO outputs with respect to the iteration, i.e. the termination criterion.

Iteration after iteration, PSO appears to converge faster than RSA. The last subfigure shows that even after 2000 criterion evaluation, the RSA has just barely improved its best solution whereas PSO continues to converge until the numerical precision is reached ($\approx 10^{-16}$). However, even if PSO algorithms use mechanisms to avoid getting trapped in local optima, it happens that wrong solutions (local minimums) are found instead, but that does not appear on these realizations.

8. Evolution strategies algorithm (ES)

Evolution Strategies [8,9] is an optimization method developed in the 1970's by Rechenberg and Schwefel. It is a population-based, natured-inspired, stochastic, derivative-free and global algorithm. Contrary to PSO, which simulates communications and interactions between individuals, ES is based on competition among them: only strong individuals survive, that is, individuals that are associated to a good criterion value.

8.1. *An evolutionary algorithm (EA)*

ES is considered as an evolutionary algorithm (EA) [9, 16 p.69] due to the use of characteristic operators (selection, crossover, mutation, replacement). Other evolutionary algorithms, like Genetic algorithms (GA), use the same operators but the structure of the algorithm is slightly different. Before to go into detail about operators, some specific terms of the evolutionary algorithm paradigm must be defined:

- An individual or chromosome (called previously a candidate or a particle in the particle swarm paradigm) is a real-valued vector (in the case of continuous optimization).
- A gene is simply one component of this vector. It has the same meaning as "parameter".
- A generation is a set of individuals, i.e. the population, at a specified time (or iteration).

The operators mentioned above are inspired by biological evolution and more particularly by natural evolution. They can be seen simply as functions which manipulates individuals of the population:

- The **selection operator** simulates the selection pressure that exists among species. Good individuals, i.e. individuals adapted to its environment, have a higher probability of participating to the reproduction phase. Thus, the better individuals are, the more offspring they have, so that less-adapted individuals will disappear since the population size of often kept constant. In other words, the selection operator defines the procedure that selects parents that will be used by the crossover operator.
- The **crossover operator** try to mimic the way biological crossover operates. It is used to create children from its parents whose number can be as lower as 1 or greater than 2.
- The **mutation operator** consists in applying a random perturbation to individuals' genes (parameters) of the population. It is used to maintain genetic diversity from one generation to the next. The use of this operator on an individual's parameter is controlled by the probability of mutation (for example 0.05=5%). Each generation, a random number is drawn in [0;1] for each parameter of each individual. If this number is lower than the probability of mutation, the mutation operator is applied. It works either by replacing the parameter by a randomly drawn value, by adding a random perturbation or by switching two different parameters.
- The **replacement operator** can be considered as a second **selection operator** [9 p.71]. It defines how to replace the old generation by the new one. Some operator, sometimes considered as "elitist", will make sure that the best individual out of all parents and all children is not replaced by a worse one.

As noted by [8], "selection is thus the antagonist to the variation operators mutation and recombination" (or crossover). Selection gives "the evolution a direction", whereas mutation and recombination maintain genetic diversity among the population. Without selection, EA would not be very different from RSA.

8.2. *Characteristics of ES algorithms*

In the case of ES algorithms, some notations exist to describe the variant used by the operators. The following two examples differ in the **replacement operator** denoted "+" or ",":

- $(\mu + \lambda)$-ES or more generally $(\mu/\rho + \lambda)$-ES: It refers to an ES-variant which uses a population of μ individuals. λ denotes the number of children (offspring) created at each iteration. ρ is the number of parents used to create one child. The "+"**replacement operator** starts by merging old and new individuals. Then, it selects the μ best out of all $\lambda+\mu$ individuals to create the next generation. In that way, the population size remains constant.
- $(\mu/\rho, \lambda)$-ES: The "," **replacement operator** simply takes the μ best individuals out of the λ children just created. Obviously, this operator implies that $\lambda>\mu$. If $\mu=\lambda$, there is no selection pressure and the replacement operator just replace the old generation by the new one.

The **selection operator** for ES is trivial. For each child to generate, it consists simply to select randomly ρ parents among all μ individuals of the current generation. The next step is to apply the **crossover operator** to the λ sets of ρ parents to generate λ temporary individuals, called "recombinants". Two common approaches[13] exist, called "discrete recombination" or "dominant recombination". Let β_1^+ and β_2^+ denote the parameter vector of two parents. With the first method, each component of the recombinant vector β^r is defined by taking the component of one of the ρ parents, randomly chosen. With the second one, each component of β^r is equal to the average of the corresponding components of all parents.

Concerning the **mutation operator**, a characteristic of ES-algorithm lies in the fact that an additional parameter[14] σ, called "mutation strength" is added to the parameter vector β of each individual. Instead of just estimating $\beta=(\beta_1, \beta_2)$, individuals are defined by a vector $\beta^+=(\beta_1, \beta_2, \sigma)$. As its name implies, σ defines the strength of the mutation, i.e. standard deviation of a normally distributed random variable. This random variable is used to generate values which are simply added to each parameter, except to the mutation strength. In fact, the value σ specified by the user is only the initial standard deviation. Indeed, each time the mutation operator is applied, the mutation strength is mutated using a separate method. It leads to the problem of self-adaptation of the mutation strength. One common way to adapt this parameter is simply to flip a coin and σ is either multiplied or divided by a coefficient γ depending on the result, heads or tails. In the implementation given in appendix, this coefficient depends on an additional parameter called "learning parameter" τ defines the rate and precision of self-adaptation [8]:

$$\gamma = e^{\tau N(0,1)} \tag{10}$$

With $\tau = 1/\sqrt{N}$ or $\tau = 1/\sqrt{2N}$ if the criterion is highly multimodal. $N(0,1)$ is a Gaussian random number generator with mean 0 and variance 1.

---

[13] These two crossover operators are available in the Matlab code given in appendix: "20/2+10" uses "discrete recombination", whereas "20/2I+10" uses "intermediate recombination".

[14] In this workshop, parameters β_1 and β_2 have the same scale, thus the same mutation strength can be applied to both parameters. In more complex problem, multiple external parameters σ_1, …, σ_n can be used.

8.3. *Population evolution*

In order to envision the evolution of a population generation after generation, the ES implementation given in appendix was slightly modified to return the parameters ("positions") of all individuals for all iterations (Figure 8). The settings of the (20/2+20)-ES algorithm was chosen to minimize the differences with the PSO algorithm. The initial mutation strength is σ=*0.5* and $\tau = 1/\sqrt{N}$ with *N=2* the number of unknown parameters.

In other words, the population is made of 20 individuals. To build a new generation, 20 recombinants are first generated. Each recombinant is built from 2 parents by using discrete recombination. By applying the mutation operator, recombinants become children. The new generation is built by taking the best individuals out of all parents and children.

By comparing Figure 6 and Figure 8, slight differences can be seen. Contrary to PSO, some individuals do not change (move) between two generations. It corresponds to parents who are better than children and so, they are kept by the replacement operator. Moreover, it appears that ES is able to localize multiple local minimum. It is indeed seen that for the iteration 5 and 10, the population is split in small groups which stay distant from each other. However, due to the selection, individuals outside the main minimum slowly disappear.

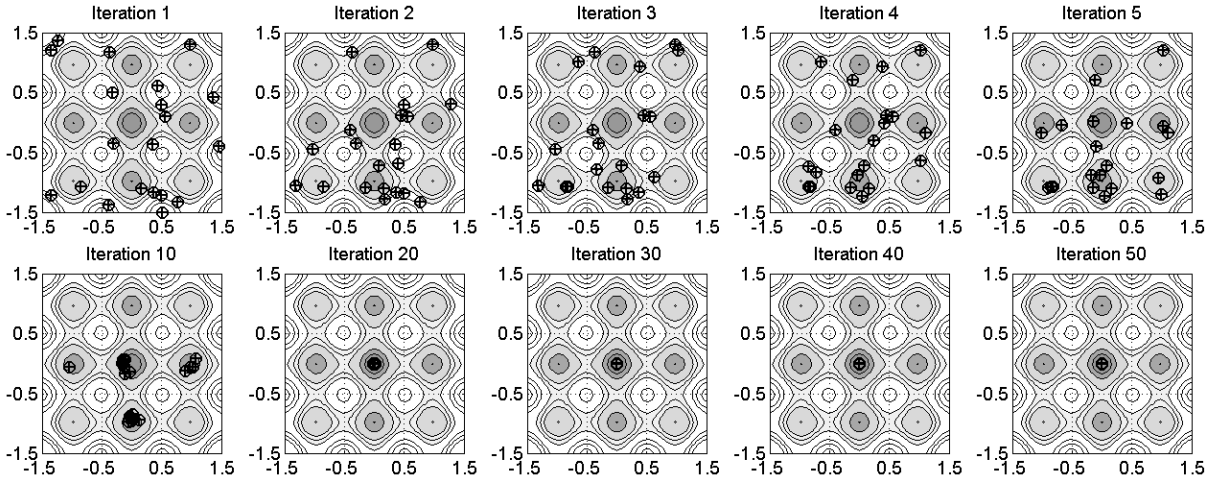


Figure 8. Positions of individuals with respect to the iteration. Variant used: (20/2+20)-ES.

8.4. *Statistical approach*

The (20/2+20)-ES algorithm is run 200 times. σ is set to *0.5* and τ is set to $1/\sqrt{2}$. Using the individual criterions for different iterations, the PDF functions are computed and compared to those of the RSA (Figure 9). Like PSO, the first subfigure refers to the initialization stage; the distribution matches quite well the corresponding distribution of the RSA. Up to the 50th iteration, outputs are greater than $\log(J) = -15$ but at the 100th iteration, most individuals are outside the abscissa range and are not more visible whereas all realizations of PSO lie between -15 and -10. Even if the convergence rate is a little slower than PSO up to the 10th iteration, it appears that ES catches up since at the 100th iteration, ES outperforms PSO. In the 3rd and 4th figure, few realizations stay around $\log(J) = 1$. It means that sometimes the (20/2+20)-ES fails to find the right solution.

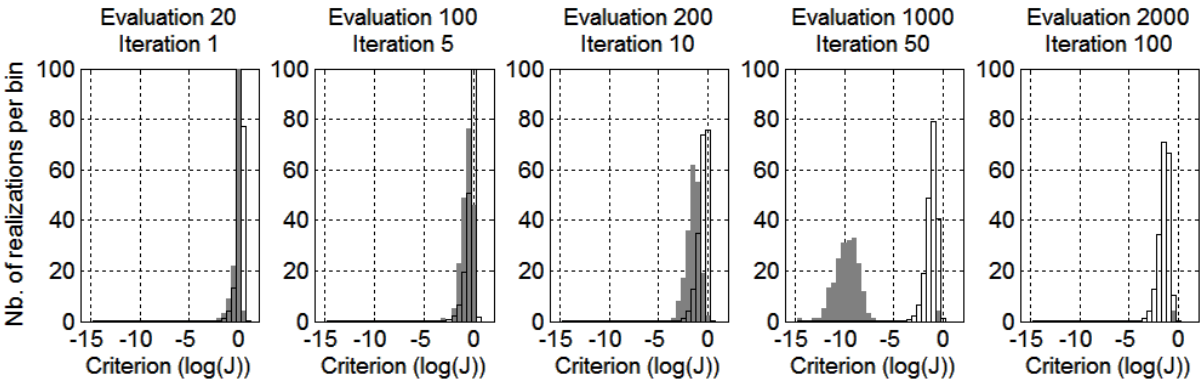


Figure 9. Distribution of ES outputs with respect to the number of iterations.

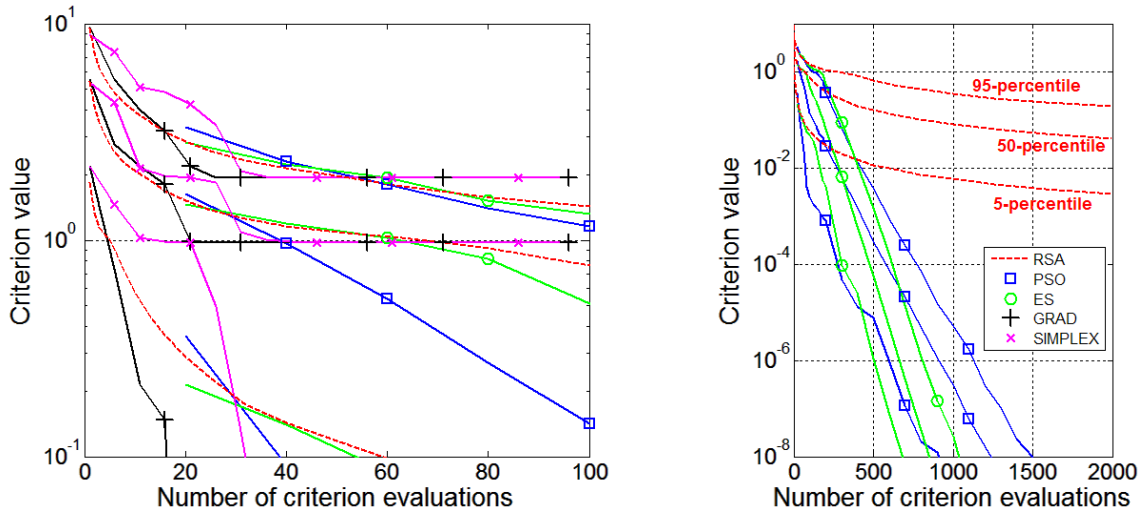
9. Conclusion

9.1. *Comparison of algorithm efficiency*

To appreciate the efficiency of each algorithm and to determine which one is the better for the 2D-Rastrigin function, a basic approach is to plot the 5, 50 ad 95-percentile curves (Figure 10) with respect to the number of evaluations *n*. Two different figures are necessary since the behaviors of the presented algorithms are significantly different. On one hand, the gradient-based and simplex algorithms converge very quickly compared to ES and PSO: no more than 50 evaluations are needed. However, for both algorithms, only the 5-percentile curve decreases to 0. It means that there is at least 5% chance of finding the solution with GRAD or SIMPLEX. The median and the 95-percentiles stabilize respectively to a value roughly equal to 1 and 2 which correspond to the local minimums. Furthermore, for *n>60*, it shows that both GRAD and SIMPLEX are less efficient than RSA if one relies on the 50 and 95-percentile curves.

On the other hand, PSO and ES need at least several thousands of evaluations to obtain correct solutions. They have the same kind of evolution. In both cases, the three percentile curves seem to converge as the number of evaluations n increases. In other words, there is at least 95% chance of finding the solution with both algorithms. Concerning the decrease of percentile curves, the lowering speed is nearly exponential compared to the relative "stagnation" of the RSA. For example, about 10^6 random evaluations would be necessary to obtain a criterion lower than 10^{-4} with a confidence of 95%. The exploitation capability of metaheuristics is therefore very useful.

Figure 11 compares the efficiency of all algorithms to RSA. It is based on expression (5.b). It shows that that PSO and ES are always better than the RSA. For *n=20* evaluations, the PSO appears to be a little below 0.5 but this is due to stochastic perturbations and numerical errors. Moreover, as seen previously in Figure 9, the convergence of ES is a little delayed compared to PSO. However, for *n=300* evaluations, both algorithms reach nearly the maximum of probability. The ES curves in the figure refer to the variant (20/2+20). With (20/2I+20), i.e. with intermediate recombination, the curve is very similar to that of the PSO.



a) Close view on GRAD and SIMPLEX quantile curves b) RSA, PSO and ES quantile curves

Figure 10. 5, 50, 95-Percentiles of RSA, GRAD, SIMPLEX, PSO and ES algorithms with respect to the number of evaluations *n*.

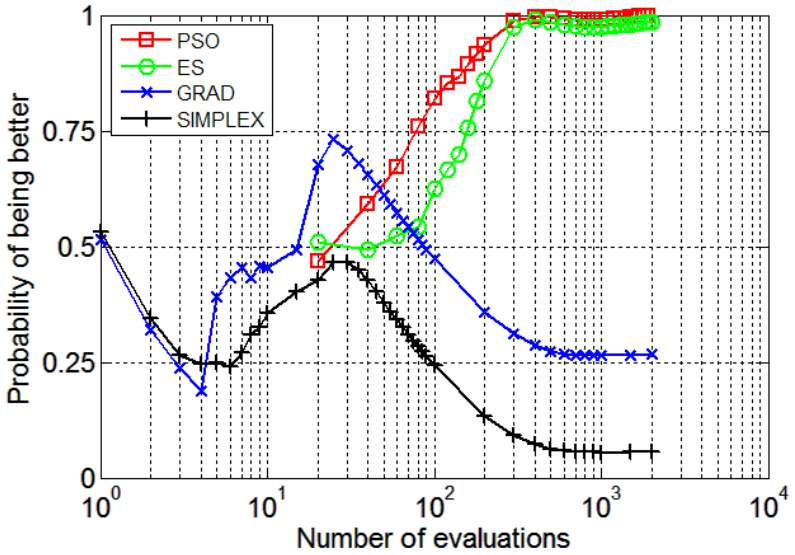


Figure 11. Probability to obtain a better solution than the RSA

About GRAD and SIMPLEX algorithms, both start at 0.5 since their initialization is random. Then the probability falls up to $n \approx 5$. Next, they catch up RSA for $5<n<30$. Thanks to its high convergence speed, GRAD is ahead of RSA around $n=30$. But since it is unable to explore the search space, it stays confined in the local minimum found. For $n>100$, RSA is again in the lead. SIMPLEX algorithm behavior is similar to the GRAD algorithm, but since it is generally less efficient than the GRAD, its curve is shift downwards.

9.2. *"No free lunch theorem"*

All comparisons made previously, all assertions that say "this algorithm is better than this one" does not hold in general case. They are only true for the example presented in this workshop. If the search space is made wider, if the parameter *A* in expression (2) is changed, if the input parameters

optimization algorithms are changed, all figures presented above could significantly changed. One can hardly say that an algorithm is better than another one without specifying the set of hypothesis and parameters that are used.

Going a step further, it's impossible to say that an algorithm A is better than B. Such a feeling is proven true thanks to the "No free lunch theorem". It says briefly that all algorithms are equivalent in terms of performance. Even for the best algorithm known until then, it should not be difficult to design a criterion that would not be successfully solved by this algorithm compared to another one.

10. References

[1] J. Dréo, A. Pétrowski, P. Siarry, E. Taillard, Métaheuristiques pour l'optimisation difficile, Eyrolles, 2003, ISBN 2-212-11368-4.

[2] C. Blum, A. Roli, Metaheuristics in combinatorial optimization : Overview and conceptual comparison, Journal ACM Computing Surveys (CSUR) Surveys Homepage archive, Volume 35, Issue 3, September 2003

[3] T. Weise, Global Optimization Algorithms – Theory and Application, e-book, 2008, www.it-weise.de/projects/book.pdf, October 2009 (available online)

[4] I. C. Trelea, The particle swarm optimization algorithm: convergence analysis and parameter selection, Information Processing Letters, Vol. 85, pp. 317-325, March 2003

[5] M. Clerc, L'optimisation par essaims particulaires, Paris, Lavoisier, 2005

[6] F. van den Bergh, An analysis of Particle Swarm Optimizers, PhD, Faculty of Natural and Agricultural Science, University of Pretoria, 2001

[7] B.W. Silverman, Density estimation for statistics and data analysis, Monographs on Statistics and Applied Probability, London, Chapman and Hall, 1986.

[8] H-G Beyer, H-P Schwefel, Evolution Strategies – A comprehensive introduction, Natural computing, Vol.1, pp.3-52, 2002

[9] J. F. Kennedy, R. C. Eberhart, Y. Shi, Swarm intelligence, Morgan Kaufman (April 9, 2001), ISBN 9781558605954.

[10] C. Porte, Méthodes directes d'optimisation - Méthodes à une variable et Simplex, Techniques de l'Ingénieur, P228

[11] J. C. Spall, Introduction to Stochastic Search and Optimization, Wiley-Interscience, 2003, ISBN-10 0471330523

[12] J. F. Bonnans, J. C. Gilbert, C. Lemaréchal, C. A. Sagastizábal, Numerical optimization: theoretical and practical aspects, Springer, 2003, ISBN-10: 3540001913

[13] D. Petit, D. Maillet, Techniques inverses et estimation de paramètres. Partie 1, Techniques de l'Ingénieur, AF4515

[14] Y. Favennec, P. Le Masson, Y. Jarny, Lecture 7 – Optimization methods for non linear estimation or function estimation, Metti 5 – Thermal Measurements and Inverse Techniques, Advanced Spring School, Roscoff, June 13-18, 2011

[15] M. Clerc, Particle swarm optimization, Paris, Wiley-ISTE, 2006, ISBN-10: 1905209045

[16] I. C. Trelea, The particle swarm optimization algorithm: convergence analysis and parameter selection, Information Processing Letters, Vol. 85, pp. 317-325, March 2003

[17] F. H. Walters, L. R. Parker, S. L. Morgan and S. N. Deming, Sequential simplex optimization, CRC Press, Boca Raton, FL, 1991, ISBN-8493–5894–9 (available online)

11. Appendix

11.1. *PSO and ES example*

The PSO and ES algorithms were implemented with the Matlab 7 programming language. It should be compatible with newer versions. For older version, it could be necessary to implement manually some functions that could not exist, like “assert(…)”, “error(…)” or “warning(…)”.

The following examples show how to use the PSO and ES program given in sections 0 and 0. The first example is the simplest one: no option is specified by the user:

```
clear all;   % Clear all variables

% Definition of the criterion (objective function) and the termination condition.
A = 2;
rastriginFunc = @(x) A*numel(x) + sum(x.^2 - A*cos(2*pi*x));  % Rastrigin function
terminationFunc = @(n, Jvec) n >= 50;                         % Abort if the number
                                                              % of iterations exceeds 50
lBound = [-10  -10];        % Search space = [-10 10]x[-10 10]
uBound = [ 10   10];

[paramPSO criterionPSO] = PSO(rastriginFunc, terminationFunc, lBound, uBound);
[paramES criterionES]   =  ES(rastriginFunc, terminationFunc, lBound, uBound);

disp(sprintf('PSO - Solution J = %10.4E', criterionPSO));
disp(sprintf('          beta_1 = %10.4E', paramPSO(1)));
disp(sprintf('          beta_2 = %10.4E', paramPSO(2)));
disp('');
disp(sprintf('ES - Solution J = %10.4E', criterionES));
disp(sprintf('         beta_1 = %10.4E', paramES(1)));
disp(sprintf('         beta_2 = %10.4E', paramES(2)));
```

Sometimes it is necessary to provide additional input parameters to the criterion, especially in more complex cases. In that case, the function and the algorithm can be defined the following way:

```
Arg1 = ...;
Arg2 = ...;
…
ArgN = ...;
criterionFunc = @(x) externalComplexFunction(x, arg1, arg2, …, argN);

nDim = 30;                          % Search space dimension (number of parameters)
lBound = repmat(-10, nDim, 1);      % Search space = [-10 10]^nDim
uBound = repmat(-10, nDim, 1);

PSOoptions = struct('particleCount', 20, ...
                    'neighborType', 'all', ...  % All particles are neighbors.
                    'inertia', 0.8, ...
                    'acc1', 1.5, ...
                    'acc2', 1.5, ...
                    'maxSpeed', 10);
[paramPSO criterionPSO] = PSO(criterionFunc, terminationFunc, lBound, uBound, PSOoptions);

ESoptions = struct( 'Signature', '(10/2+20)', ...
                    'learning', 1/sqrt(nDim), ...
                    'sigma', 5);
[paramES criterionES] = ES(rastriginFunc, terminationFunc, lBound, uBound, ESoptions);
```

11.2. *PSO Implementation*

```
% PSO Algorithm Implementation
% Input arguments:
% - objFunc     : a function handle. "@functioname"   or   "@(beta)anonymous_function"
%                 This function takes only the parameter vector "beta" as input argument.
% - termFunc    : a function handle. "@functioname"   or   "@(n, J)anonymous_function"
%                 This function takes two arguments:
%                  => n : current iteration index (>=1).
%                  => J : vector [nx1] containing the criterion value with respect to the iteration index.
% - lowerBound  : lower bounds of each component of beta (column vector [nb x 1])
% - upperBound  : upper bounds of each component of beta (column vector [nb x 1])
% - options     : structure containing the following parameters:
%      => particleCount    : size of the swarm. By default 20.
%      => neighborCount    : number of neighbors. By default 3.
%      => neighborType     : type of neighborhood ("all", "circle", "rand"). By defaut "all"
%      => inertia          : particle momentum. By default 0.7.
%      => acc1             : social acceleration coefficient. By default 1.4.
%      => acc2             : cognitive acceleation coefficient. By default 1.4.
%      => maxSpeed         : maximum velocities. By default 1/2 * (upperBound - lowerBound).
%      => plot             : specify wether or not the criterion value must be ploted.
%      => plotPeriod       : number of iterations between each draw.
%      => trace            : if true, the output parameter "beta" will contain the parameters for all
%                                iteration. (matrix [nIter x nParam]) and J will a matrix nIter x 1
%                            if false, "beta" will be the parameters corresponding to the
%                                    best candidate found and J its criterion value.
% Output parameters:
% - beta        : [1 x nParam] or [nIter x nParam] matrix depending on <options.trace>
% - J           : [1 x 1]      or [nIter x 1] matrix depending on <options.trace>
function [beta J] = PSO(objFunc, termFunc, lowerBound, upperBound, opt)
if (nargin < 5)
    assert(nargin >= 4, 'Missing arguments');
    opt = struct();
end

% Transform vectors of unknown shape to column vectors
if size(lowerBound, 2) > 1, lowerBound = lowerBound'; end
if size(upperBound, 2) > 1, upperBound = upperBound'; end

NbParam = numel(lowerBound);

assert((size(lowerBound, 2) == 1) && (size(upperBound, 2) == 1), ...
    '<lowerBound> and <upperBound> must be column vector');
assert(numel(upperBound) == NbParam, '<lowerBound> and <upperBound> must have the same size');
assert(all(min(upperBound - lowerBound) >= 0), 'Invalid boundaries');

% Checking input parameters and initilization of missing options.
if ~isfield(opt, 'particleCount'), opt.particleCount = 20;  else assert((opt.particleCount > 0) &&
(mod(opt.particleCount, 1) == 0), 'Invalid neighbor count <particleCount>. Must be a positive integer.');
end;
if ~isfield(opt, 'neighborCount'), opt.neighborCount = 3;   else assert((opt.neighborCount >= 0) &&
(mod(opt.neighborCount, 1) == 0), 'Invalid neighbor count <neighborCount>. Must be a positive integer.');
end;
if ~isfield(opt, 'neighborType'), opt.neighborType = 'all'; else assert(any(ismember({'all' 'rand'
'circle'}, opt.neighborType)), 'Invalid neighbor type <neighborType>. Must be either all/rand/circle.');
end;
if ~isfield(opt, 'inertia'), opt.inertia = 0.7; else assert(opt.inertia >= 0.0, 'Invalid <inertia> value.
Must be a real positive value'); end;
if ~isfield(opt, 'acc1'), opt.acc1 = 1.4; else assert(opt.acc1 >= 0.0, 'Invalid <acc1> value. Must be a
real positive value'); end;
if ~isfield(opt, 'acc2'), opt.acc2 = 1.4; else assert(opt.acc2 >= 0.0, 'Invalid <acc2> value. Must be a
real positive value'); end;
if ~isfield(opt, 'maxSpeed'), opt.maxSpeed = 0.5*(upperBound - lowerBound); elseif numel(opt.maxSpeed) ==
1, assert(opt.maxSpeed > 0); opt.maxSpeed = ones(NbParam, 1) * opt.maxSpeed; else
assert((numel(opt.maxSpeed) == NbParam) && all(opt.maxSpeed > 0), 'Invalid value <maxSpeed>'); end;
if ~isfield(opt, 'plot'), opt.plot = false; else assert(isa(opt.plot, 'logical'), 'Invalid <plot> value.
Must be a logical'); end;
if ~isfield(opt, 'plotPeriod'), opt.plotPeriod = 10; else assert((opt.plotPeriod > 0) &&
(mod(opt.plotPeriod, 1) == 0), 'Invalid <plotPeriod> value. Must be a integer greater than 0'); end;
if ~isfield(opt, 'trace'), opt.trace = false; else assert(isa(opt.trace, 'logical')); end;
if strcmpi(opt.neighborType, 'circle') && mod(opt.neighborCount, 2) == 0
    warning('With a circle (index-based) neighborhood, the number of neighbors should be a even integer');
end

optFields = fieldnames(opt);
for i=1:numel(optFields)
    assert(any(ismember({'particleCount' 'neighborCount' 'neighborType' 'inertia' 'acc1' 'acc2' 'maxSpeed'
'plot' 'plotPeriod' 'trace'}, optFields{i})), 'Invalid field <%s>', optFields{i});
end
```

```
% Convenient variable definitions
NbPart = opt.particleCount;
NbNeigh = opt.neighborCount;

% Initilization of particles
Part = repmat(struct('param', [], 'speed', [], 'value', NaN, 'memParam', [], 'memValue', NaN), 1, NbPart);
for i=1:NbPart
    Part(i).param = lowerBound + rand(NbParam, 1) .* (upperBound - lowerBound);
    Part(i).speed = (2*rand(NbParam, 1)-1) .* opt.maxSpeed;
    Part(i).value = objFunc(Part(i).param);
    Part(i).memValue = Part(i).value;
    Part(i).memParam = Part(i).param;

    assert(numel(Part(i).value) == 1, 'The objective function did not return a real value');
end

iterCount = 1;   [bestValue bestPartIndex] = min([Part.memValue]);
J = bestValue;   beta = Part(bestPartIndex).memParam';     inertia = opt.inertia;

while ~termFunc(iterCount, [Part.memValue])     % Used defined termination criterion

    if opt.plot
        disp(sprintf('Itération %d', iterCount));
    end

    for i=1:NbPart
        % Neighborhood initilization
        if ~strcmpi(opt.neighborType, 'all')
            if strcmpi(opt.neighborType, 'rand')    % Randonmly drawn neighborhood
                Neighborhood = min(ceil(NbPart*rand(1, NbNeigh)+1E-10), NbPart);
            else % Circular and index-based neighborhood
                HalfNbNeigh = floor(NbNeigh/2);
                Neighborhood = min(ceil(mod(i-1 + (-HalfNbNeigh:1:HalfNbNeigh), NbPart) + 1), NbPart);
            end

            % The best neighbor is retrieved
            [bestNeighborValue bestNeighborIndex] = min([Part(Neighborhood).memValue]);
        else
            bestNeighborIndex = bestPartIndex;
        end

        % Velocity update
        Part(i).speed = inertia * Part(i).speed + ...
            opt.acc1 * rand(NbParam, 1) .* (Part(bestNeighborIndex).memParam - Part(i).param) + ...
            opt.acc2 * rand(NbParam, 1) .* (Part(i).memParam - Part(i).param);
        Part(i).speed = min(Part(i).speed, opt.maxSpeed);
        Part(i).speed = max(Part(i).speed, -opt.maxSpeed);

        % Particle's position update. Positions are modified based on boundary constraints.
        Part(i).param = min(max(Part(i).param + Part(i).speed, lowerBound), upperBound);

        % If a particle reaches a boundary, the corresponding component of the velocity is nullified.
        Part(i).speed(Part(i).param == upperBound) = 0;
        Part(i).speed(Part(i).param == lowerBound) = 0;
        Part(i).value = objFunc(Part(i).param); % Criterion evaluation with the new particle's position

        % Replacement of the best particle experience if the new position is better.
        if (Part(i).value < Part(i).memValue)
            Part(i).memValue = Part(i).value; Part(i).memParam = Part(i).param; end;

        % Asynchroneous PSO
        [bestValue bestPartIndex] = min([Part.memValue]);
    end

    iterCount = iterCount + 1;
    if opt.trace
        J = [J; bestValue];       beta = [beta; Part(bestPartIndex).memParam'];
    else
        J = bestValue;            beta = Part(bestPartIndex).memParam;
    end

    if opt.plot && (mod(iterCount, opt.plotPeriod) == 0)
        plot(1:iterCount, J);
        xlabel('Iteration');        ylabel('Objective function');
        drawnow expose;
        pause(.001);
    end
end
```

11.3. *ES implementation*

```
% ES Algorithm Implementation
% Input arguments:
% - objFunc     : a function handle. "@functioname"   or   "@(beta)anonymous_function"
%                 This function takes only the parameter vector "beta" as input argument.
% - termFunc    : a function handle. "@functioname"   or   "@(n, J)anonymous_function"
%                 This function takes two arguments:
%                       => n : current iteration index (>=1).
%                       => J : vector [nx1] containing the criterion value with respect to the iteration
index.
% - lowerBound  : lower bounds of each component of beta (column vector [nb x 1])
% - upperBound  : upper bounds of each component of beta (column vector [nb x 1])
% - options     : structure containing the following parameters:
%      => signature        : ES variant definition. Ex: "(20/2+20)" (default), "(10/2I,40)"
%      => learning         : learning rate coefficient used to update the mutation strength
%      => sigma            : initial mutation strength;
%      => plot             : specify wether or not the criterion value must be ploted.
%      => plotPeriod       : number of iterations between each draw.
%      => trace            : if true, the output parameter "beta" will contain the parameters
%                                for all iteration. (matrix [nIter x nParam]) and J will
%                                be a matrix nIter x 1
%                            if false, "beta" will be the parameters corresponding to the
%                                best candidate found and J its criterion value.
% Output parameters:
% - beta        : [1 x nParam] or [nIter x nParam] matrix depending on <options.trace>
% - J           : [1 x 1]      or [nIter x 1] matrix depending on <options.trace>
function [beta J sigma] = ES(objFunc, termFunc, lowerBound, upperBound, opt)
if (nargin < 5)
    assert(nargin >= 4, 'Missing arguments');
    opt = struct();
end

% Checking input parameters and initilization of missing options.
if size(lowerBound, 2) > 1, lowerBound = lowerBound'; end
if size(upperBound, 2) > 1, upperBound = upperBound'; end

assert((size(lowerBound, 2) == 1) && (size(upperBound, 2) == 1), '<lowerBound> and <upperBound> must be
column vector');
NbParam = length(lowerBound);

assert(length(lowerBound) == length(upperBound), '<lowerBound> and <upperBound> must have the same size');
assert(min(upperBound - lowerBound) >= 0, 'Invalid boundary constraints');

% Initialization of default parameters
if ~isfield(opt, 'signature'), opt.signature = '10/2+20';
else
    %Ex:                                          10 /   4     +       2
    assert(~isempty(regexpi(opt.signature, '^\(? \d+ /  \d+ I? (\+|,)  \d+  \)?$', 'freespacing')),
'Signature incorrecte <%s>', opt.signature);
end

if ~isfield(opt, 'learning'), opt.learning = 1/sqrt(NbParam); else assert(numel(opt.learning) == 1 &&
isa(opt.learning, 'double') && opt.learning > 0.0, 'Invalid <learning> value'); end;
if ~isfield(opt, 'sigma'), opt.sigma = 0.25 * mean(upperBound - lowerBound); else assert(((numel(opt.sigma)
== 1) || (numel(opt.sigma) == NbParam)) && isa(opt.sigma, 'double') && min(opt.sigma) > 0, 'Valeur
incorrecte de sigma (initial value of mutation standard deviation)'); end;
if ~isfield(opt, 'plot'), opt.plot = false; else assert(numel(opt.plot) == 1 && isa(opt.plot, 'logical'));
end;
if ~isfield(opt, 'plotPeriod'), opt.plotPeriod = 10; else assert((opt.plotPeriod > 0) &&
(mod(opt.plotPeriod, 1) == 0), 'Invalid <plotPeriod> value. Must be a integer greater than 0'); end;
if ~isfield(opt, 'trace'), opt.trace = false; else assert(isa(opt.trace, 'logical')); end;

optFields = fieldnames(opt);
for i=1:numel(optFields)
    assert(any(ismember({'signature' 'learning' 'sigma' 'plot' 'plotPeriod' 'trace'}, optFields{i})),
'Invalid field <%s>', optFields{i});
end

options = regexpi(opt.signature, '^\(? (?<popSize>\d+) /  (?<parentCount>\d+) (?<recomb>I)?
(?<operator>\+|,)  (?<childrenCount>\d+) \)?$', 'freespacing', 'names');
popSize = str2double(options.popSize);
parentCount = str2double(options.parentCount);
childCount = str2double(options.childrenCount);
operator = options.operator;

assert(~isnan(parentCount) && (parentCount > 0), 'Invalid number of parents. Must be positive.');
assert(~isnan(childCount) && (childCount > 0), 'Invalid number of children. Must be positive.');
assert(~isnan(popSize) && (popSize > 0), 'Invalid population size. Must be positive.');
```

```
if strcmp(options.operator, ','), assert(childCount >= popSize, 'With '','' operator, the number of
children must be greater than the population size'); end;

% Initialization of individuals
Population = repmat(struct('param', [], 'sigma', opt.sigma, 'value', NaN), 1, popSize);
for i=1:popSize
    Population(i).param = lowerBound + rand(NbParam, 1) .* (upperBound - lowerBound);
    Population(i).sigma = opt.sigma;
    Population(i).value = objFunc(Population(i).param);
    assert(numel(Population(i).value) == 1, 'The objective function did not return a real value');
end

iterCount = 1;
[sortedPop sortedIndex] = sort([Population.value]);
Population = Population(sortedIndex);

J = sortedPop(1);
beta = Population(1).param';
sigma = Population(1).sigma;

while ~termFunc(iterCount, J(end,1))
    assert(min(J(end,:)) == J(end,1));

    children = repmat(  struct(     'param', [], ...
                                    'sigma', opt.sigma, ...
                                    'value', NaN), ...
                        1, childCount);

    for i=1:childCount

        % "Recombinant" generation. % Parents and randomly selected
        parents = min(ceil(popSize * rand(parentCount, 1) + 1E-10), popSize);

        if strcmp(options.recomb, 'I')      % Intermediate recombination
            children(i).sigma = mean( [ Population(parents).sigma ] );
            children(i).param = mean( [ Population(parents).param ], 2);
        else                                % Discrete recombination
            children(i).sigma = sample( [ Population(parents).sigma ] );
            children(i).param = sample( [ Population(parents).param ], 2);
        end

        % Mutation strength update
        children(i).sigma = children(i).sigma * exp(opt.learning * randn);
        children(i).param = children(i).param + children(i).sigma * randn(NbParam,1);

        % Individual positions are clamped based on boundary constraints
        children(i).param = min(max(children(i).param, lowerBound), upperBound);
        children(i).value = objFunc(children(i).param);             % Criterion evaluation
    end

    if strcmp(operator, '+')            % Replacement of the old population
        Population = [Population children];
        [sortedPop sortedIndex] = sort([Population.value]);
        Population = Population(sortedIndex(1:popSize));
    elseif strcmp(operator, ',')
        [sortedPop sortedIndex] = sort([children.value]);
        Population = children(sortedIndex(1:popSize));
    else
        error('Operateur <%s> incorrect', operator);
    end

    iterCount = iterCount + 1;

    if ~opt.trace
        J = Population(1).value;   beta = Population(1).param';    sigma = Population(1).sigma;
    else
        J = [J; Population(1).value];
        beta = [beta; Population(1).param'];  sigma = [sigma; Population(1).sigma];
    end

    if opt.plot && (mod(iterCount, opt.plotPeriod) == 0)
        plot(1:iterCount, J);
        xlabel('Iteration');
        ylabel('Objective function');
        drawnow expose;
        pause(.001);
    end
end
end % Function end
```

```
% This function returns a randomly drawn component of a column vector
% If <A> is a matrix, sample(A) treats the columns of A as vectors, returning a row vector.
% If <dim> is provided, it specifies how vectors must be extracted from the matrix: columns or rows
(default = 1 -> returns a row vector)
function M = sample(A, dim)
    assert(ndims(A) <= 2, 'A must be a matrix');

    if nargin == 1
        i = ceil(size(A,1) * rand);
        j = ceil(size(A,2) * rand);
        M = A(i,j);
    elseif nargin == 2
        assert((dim >= 0) && (dim <= 2), 'Invalid dimension');

        if dim == 1
            M = zeros(1, size(A,2));
            for i=1:size(A,2)
                M(1,i) = A(ceil(size(A,1)*rand), i);
            end
        else
            M = zeros(size(A,1), 1);
            for i=1:size(A,1)
                M(i,1) = A(i, ceil(size(A,2)*rand));
            end
        end
    else
        error('Missing argument');
    end
end
```