Problem 2. Evolutionary Strategies.

Victor Montiel Argaiz

January 31, 2012

Contents

1 Introduction

In this problem set we explore the capabilities of *Evolutionary Strategies*, a technique within the *Evolutionary Computation* family used for function optimization. The technique is used against three test function used to measure the effectiveness of optimization functions [3].

The first function is the *Sphere model*, described in equation (1). The function, that for this problem has been ranged within the hypercube $[-10, 10]^n$, is smooth (C^∞), symmetrical with respect to any of the coordinate axis, and has a sole local minimum located at $x^* = \langle 0, 0, \ldots 0 \rangle$ and with value $f(x^*) = 0.0$. With these properties, the search process is straightforward, and even simple search algorithms could find it. A plot of the *Sphere model* can be found in figure (1).

$$f(\overline{x}) = \sum_{i=1}^{n} x_i^2 \tag{1}$$

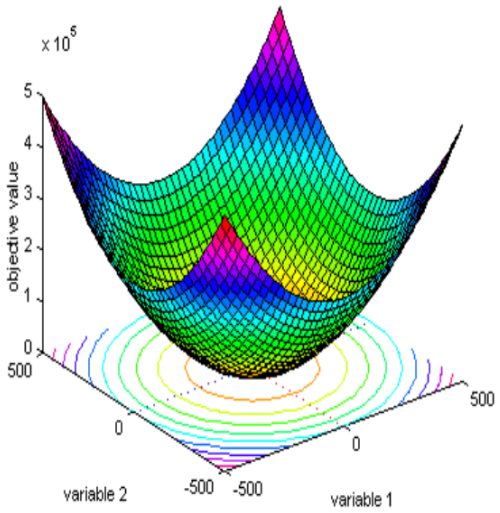

Figure 1: Sphere model. Picture from MatLab GEATbx

The second function belongs to a family of test functions widely used in optimization theory, known as *De Jong* functions [1]. In our problem, we will test our algorithm against the *De Jong's number 5* function, described in equation (2, 3). The function is defined in the subset $[-65536, 65536] \times [-65536, 65536]$ of $\mathbb{R}^2$ and it is a multi-modal, bi-dimensional function with 25 local extremes. The minimum is located at $x^* = \langle -32, -32 \rangle$, and its value is close to $f(x^*) = 1.0$. Observe that, although the function is defined in a large range, the interesting properties of it lay within a small limited region, in particular, the local extremes are found in the points specified in the matrix A. Outside this region, the function is very flat, with value of 500. A plot of the *De Jong* function can be seen in figure (2)

$$F_5(x_1, x_2) = \frac{1}{\frac{1}{K} + \sum_{j=1}^{25} \frac{1}{c_j + \sum_{i=1}^{2} (x_i - a_{ij})^6}} \tag{2}$$

with $c_j = j$, $K = 500$, $x_i \in [-65536, 65536]$, $i = 1, 2$ and

$$A = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & -16 & -16 & -16 & 0 & 0 & 0 & 0 & 0 & 16 & 16 & 16 & 16 & 16 & 32 & 32 & 32 & 32 & 32 \end{bmatrix} \tag{3}$$

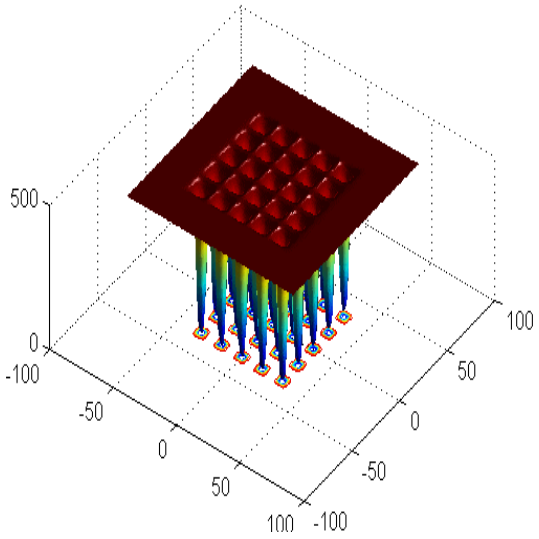

Figure 2: De Jong number 5 model. Picture from MatLab GEATbx

Finally, *Schwefel's* function, defined in equation (4), is a multidimensional, multi-modal function defined in $[-500, 500]^n$ with a large number of local extremes points. The global minimum is located at $x = \langle 420.96, 420.96, \ldots, 420.96 \rangle$ with minimum $f(x^*) = 0$.

$$f(x) = 418.9829 \cdot n + \sum_{i=1}^{n} -x_i \cdot sin(\sqrt{|x_i|}) \tag{4}$$

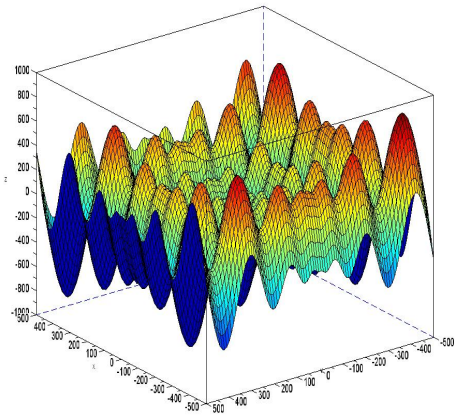

Figure 3: Schwefel's model. Picture from MatLab GEATbx

The three functions above form a test suite of increasing difficulty, passing first from a mono-modal symmetric function, the (*sphere*) model, to a multi-modal low-dimensional function, (*De Jong's*). Finally, *Schwefel's* function is both multi-modal and high-dimensional, making it the hardest target for our *Evolutionary Strategy*. All functions are defined in a bounded region of the space and are easily scalable, that is, they can be defined with a varying number of dimensions, being able to test the robustness of the strategies as the dimensionality increases. Finally, since they have been studied for many years as benchmarks of optimization methods, their solution are well-know, and we can employ the usual performance metrics to compare the different strategies.

2 Description of the Evolutionary Strategy

For real-valued functions of type $\mathbb{R}^n \mapsto \mathbb{R}$, the natural genotype representation for the problem is a *n-dimensional* vector of real values, being n de dimension of the function to be minimized. Additionally, the genotype is extended to include the control parameters of the algorithm, as we shall see, depending on the mutation scheme we use.

Mutation and recombinator operators for this representation are similar to those used in *genetic algorithms* techniques. For the mutator operator, we use random increments in each of the alleles, according to a normal distribution whose standard deviation varies with the evolutionary process. Amongst the three most common mutations, *uncorrelated one-step*, *uncorrelated n-step* and *correlated*, we have evaluated the performance of the first two methods.

In the former one, explained in equation (5), the search (mutation) is done by adding random increments located in an hyper-sphere, i.e., the distance from the original point is 'equal' (in probability terms) in all dimensions, since we are using the same σ for all axis.

$$\begin{aligned} \sigma' &= max(\sigma \cdot e^{\tau N(0,1)}, \epsilon_0) \\ x'_i &= x_i + \sigma' N_i(0,1) \end{aligned} \tag{5}$$

The second mutation, explained in formula (6), accepts different scale factors, σ_i, for each of the dimensions, creating hyper-ellipses around the original point, thus, allowing larger variations for some axes than for others.

$$\begin{aligned} \sigma'_i &= max(\sigma_i \cdot e^{\tau' N(0,1) + \tau N_i(0,1)}, \epsilon_0) \\ x'_i &= x_i + \sigma'_i N_i(0,1) \end{aligned} \tag{6}$$

Observe that, dealing with bounded functions defined in a finite subset of the $\mathbb{R}^n$ space, we have to define somehow what to do with the individuals that, after the mutation process, are located outside the region where the function is defined. The first approach was to limit the individual to the range of definition of the function, so, those individual exceeding that range for any the dimension were forced to be at the boundary. This approach did not result in good results, specially for the *De Jong's* function, as we will see, where most of the individual ended up located in the boundary of the function because of the special properties of the test model. To overcome this problem, we defined a method to 'create' valid individuals, i.e., within the range of definition of the function, after a mutation. The idea is to apply the concept of *modular arithmetic* to each dimension, and so, when for one of the dimensions i, the point x_i exceed the range of definition of the function by δ, this point is transformed in $x'_i = x_{i,0} + \delta$ as if the range were circular, being $x_{i,0}$ the lower bound in the range of definition for the dimension i.

Finally, for the recombinator operator, we distinguish between the *object part*, encoding the point in the space, and the *strategy parameters part*, encoding the parameters which control the mutation process. For the first one we use *discrete recombination*, choosing randomly the value of one of the parents, and, for the second one we use *intermediary recombination*, averaging the values of the parents' parameter to create the offspring parameter. This way, diversity within the solution space is assured, allowing very different combinations of values in the search process, and at the same time, the averaging in the parameter genes performs a more cautious adaptation.

3 Code Structure and implementation

3.1 General Structure

As for the *Genetic Algorithms* problem set, we have opted for an object oriented approach developed in C++, using Boost libraries [2] for random number generation, parsers, command line parameter and shared pointers. For result plotting we have used the gnuplot-iostream library.

Code organization is based in the following files:

- `CrossOver.hpp`: Definition of the crossover operator, defining an abstract class `CrossOver` that has to be derived for each operator. In the case of *Evolutionary Strategies*, two crossover operators are used, the *intermediary* and *discrete* recombination, for each part of the genotype.
- `Mutators.hpp`: Definition of the mutator operator, defining an abstract class `Mutator` that has to be derived for each operator. For this problem set, we have implemented *one-step* and *n-step* mutator operators.
- `Evaluators.hpp`: Definition of the evaluator operator which returns a fitness value for each individual. In the file the abstract class `Evaluator` is defined, and has to be derived for each evaluation function implemented. In the context of this problem set, the evaluator happens to be the test function to be minimized.
- `Selector.hpp`: Definition of the selector operator as an abstract class `Selector`, which defines the selection process for mating and survival. In the *Evolutionary Strategies*, selection for mating is done using *random sampling* and selection for survival is based on *ranking selection*.
- `Population.hpp`: Defines a population as a vector of individuals.
- `Genome.hpp`: It models a genome or individual, using the usual encoding for *evolutionary strategies*.
- `RandomGenerator.hpp`: Defines the random generator classes to be used through the problem set. Concretely, we use *Bernoulli*, *Gaussian* and *uniform integer* random number generators.
- `ResultsAccumulator.hpp`: Stores partial results of the population for each generation, keeping track of the best and worst individuals, the average fitness value of the population, and the consumed user-cpu time. At the same time, the class `ExecutionAccumulator` has been defined, which aggregates the results for each execution of the algorithm.
- `ES.hpp`: is the orchestrator of the *Evolutionary Strategy*, containing references to `Mutator`, `CrossOver`, `Evaluator`, `Selector` and `Population` objects. It simplifies the interface to run the simulation by defining the methods: `Initialize`, `MovetoNextIteration`, `TerminationCondition`, `Evaluate`, `UpdateStatistics`.
- `optionparser.hpp`: contains functions to parse the command-line arguments, using the `program options` classes within the Boost library. Arguments can be passed either as a command-line parameters, or embedded in a configuration file.
- `test.hpp`: test batteries for debugging purposes.
- `types.hpp`: Types used within the code.
- `main.cpp`: main method which reads the options from either the command-line or the configuration files and executes the simulations, saving results in text files as well as in *.png* pictures.

3.2 Code Flow

The execution flow of the program can be summarized, according to the code in the main method, as follows:

- Step 1: Parsing of the command-line arguments with the parameters defining the simulation to be run.
- Step 2: Creation of the operators to be used by the *Evolutionary Strategy*. This includes the mutation, recombination, evaluation and selection operators. Creation of the population object.
- Step 3: Creation of the ES object instance, passing as arguments the operators to be used in the strategy, and creation of the ResultAccumulator objects, to keep track of the partial results.
- Step 4: For each execution to be run, initialization of the evolutionary strategy, invoking the `Initialize` method on the ES object.
- Step 5: Repetition of the evolutionary process until the termination condition, i.e., maximum number of generation, is achieved. The instructions repeated in the loop are, by this order, selection of parents, recombination of parents, creation of a new population with off-springs and eventually original parents, depending on the scheme of the strategy. Then, mutation of the individuals of the population, evaluation of the resulting individuals and selection of survivors is performed.
- Step 6: The loop above is repeated for each generation, for each execution of the problem, until all instances have been run. Then, by constructing an `ExecutionAccumulator` object, we summarize the results of all of the executions of the strategy plotting charts and printing partial results to `.csv` text files.

4 Results

4.1 Sphere Model

As we discussed in the introduction, the *sphere model* is the simplest function studied in this problem set. Its smooth behavior, the existence of a unique extreme point along with its symmetry makes the function an easy target even for simple optimization methods. Being *Evolutionary Strategies* robust optimization methods even for the hardest test function, we expect that the experiments in this section are easily resolved, even for a large number of dimensions.

Results for this section have been made by executing the program using the following parameters files: `question2a.cfg`, `question2b.cfg`, `question2c.cfg`, `question2d.cfg`.

4.1.1 (μ, λ) strategy with uncorrelated one-step mutation

In figure (4) we have plotted the estimation of the minimum value for the test function, using the simplest *one-step* mutation, achieving pretty good and accurate results for the optimization. All of the 30 simulations we have run draw similar results, achieving, with minimum error (of $10e^{-7}$ order), the minimum of the function, located at $x^* = \langle 0, \ldots, 0 \rangle$, and with value $f_{sphere}(x^*) = 0$. In the same figure (4) we have summarized the statistics of the best-fitted individual for the 30 executions. The range of the results, difference between worst and best cases, is really small, and the standard deviation of the population lies within the 10^{-8} order of magnitude.

In figure (5) we have plotted the evolution of the solution for one of the executions using logarithmic axis to really appreciate the speed of convergence. As we can see, convergence to the minimum is really fast, and within a few generations we reach function values close to the minimum up to the order of the minimum error, $10e^{-7}$ in this case. As we will see later, this convergence error is closely related to the

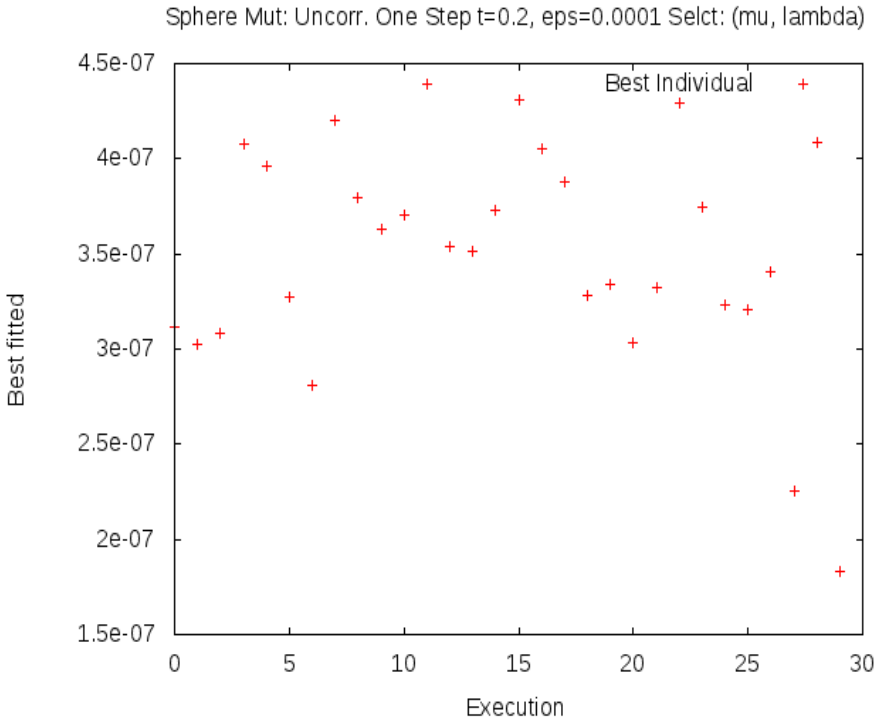

Figure 4: Sphere function with uncorrelated one-step mutation. (μ, λ) selection. Statistics: $minf(x^*) = 1.83e^{-7}, \overline{f(x^*)} = 3.50e^{-7}, maxf(x^*) = 4.39e^{-7}, \sigma_{f(x^*)} = 5.89e^{-8}$

parameter ϵ_0 used in the simulation.

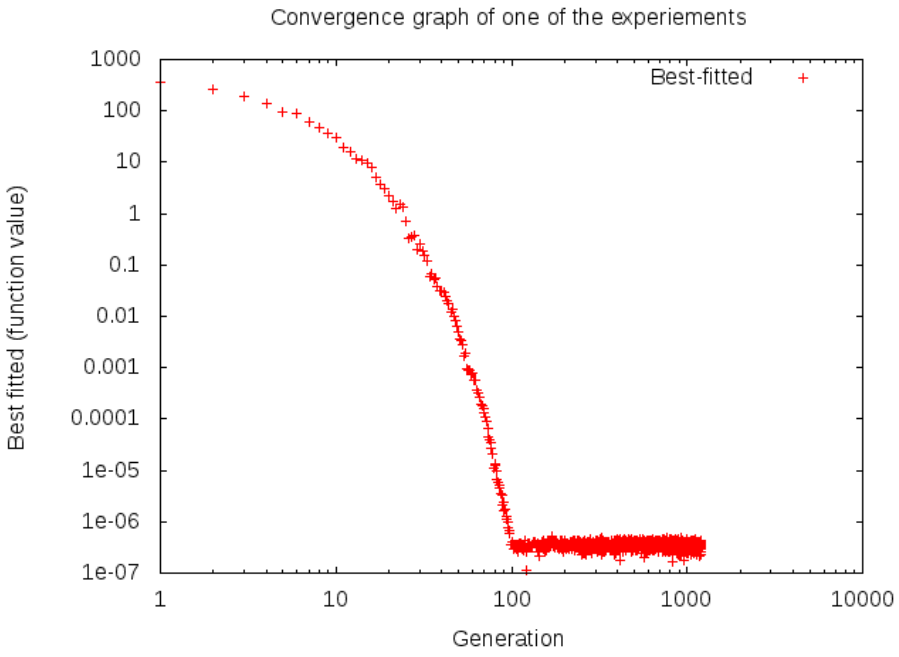

Figure 5: Evolution of the best-fitted individual for one of the executions

4.1.2 (μ, λ) strategy with uncorrelated n-step mutation

Results with *uncorrelated n-step* mutation for the *sphere* problem are pretty much the same as for the *one-step* strategy. Figure (6) plots the outcome for each of the 30 executions, along with the descriptive statistics. Although the statistics are slightly worse than for the *one-step* mutation, there is not significant differences between them, given that the final results remain in the same orders of magnitude.

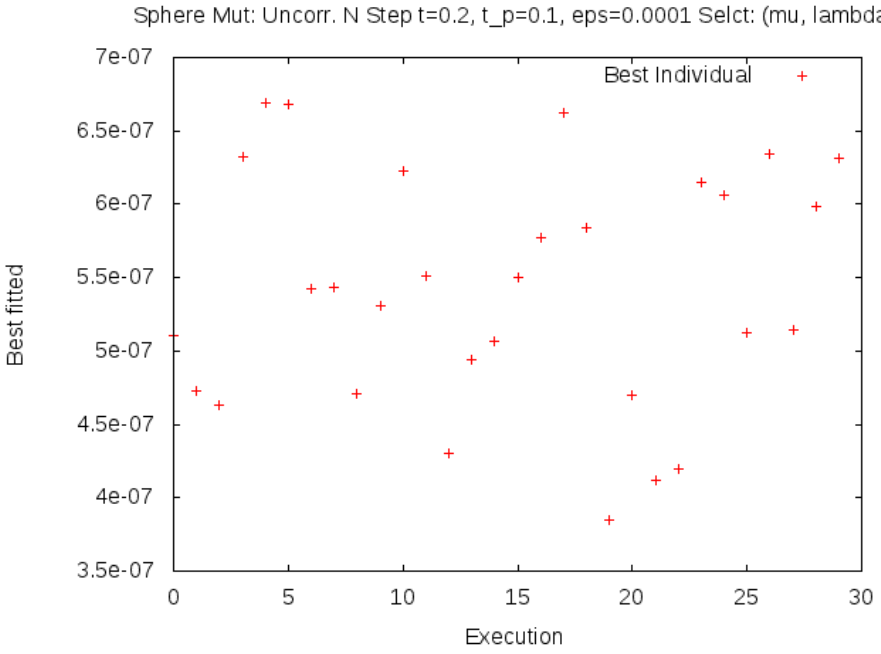

Figure 6: Sphere function with uncorrelated n-step mutation. (μ, λ) selection. Statistics: $minf(x^*) = 3.85e^{-7}, \overline{f(x^*)} = 5.42630e^{-7}, maxf(x^*) = 6.6868e^{-7}, \sigma_{f(x^*)} = 8.0894e^{-8}$

4.1.3 $(\mu + \lambda)$ strategy with uncorrelated one-step mutation

For the $(\mu + \lambda)$, *one-step* configuration, we obtain similar results, plotted in figure (7).

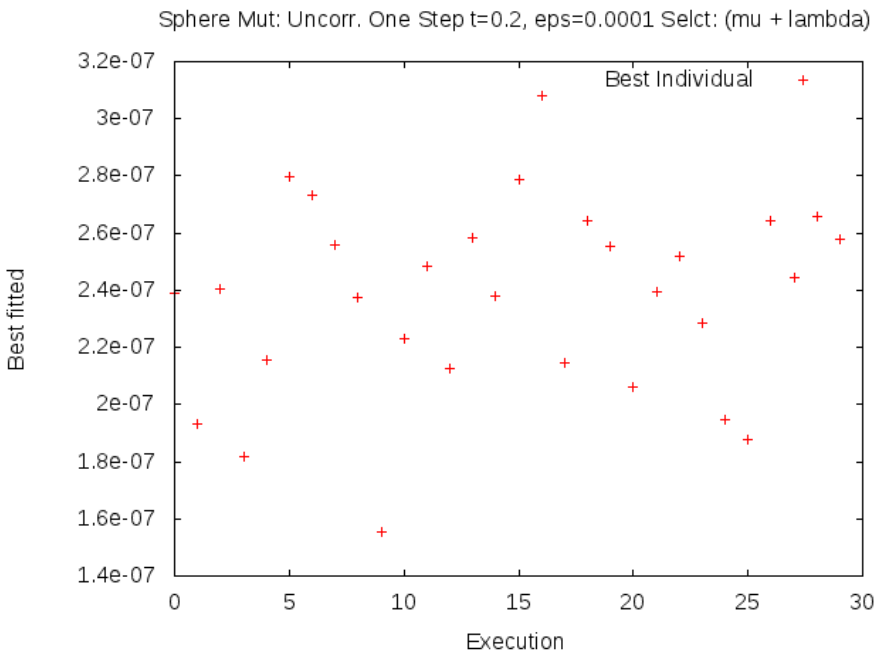

Figure 7: Sphere function with uncorrelated n-step mutation. $(\mu + \lambda)$ selection. Statistics: $minf(x^*) = 1.55e^{-7}, \overline{f(x^*)} = 3.08e^{-7}, maxf(x^*) = 2.37e^{-7}, \sigma_{f(x^*)} = 3.37e^{-8}$

4.1.4 $(\mu + \lambda)$ strategy with uncorrelated n-step mutation

Finally, figure (8) shows results for $(\mu + \lambda)$, *n-step* scheme, obtaining, as expected, very good results.

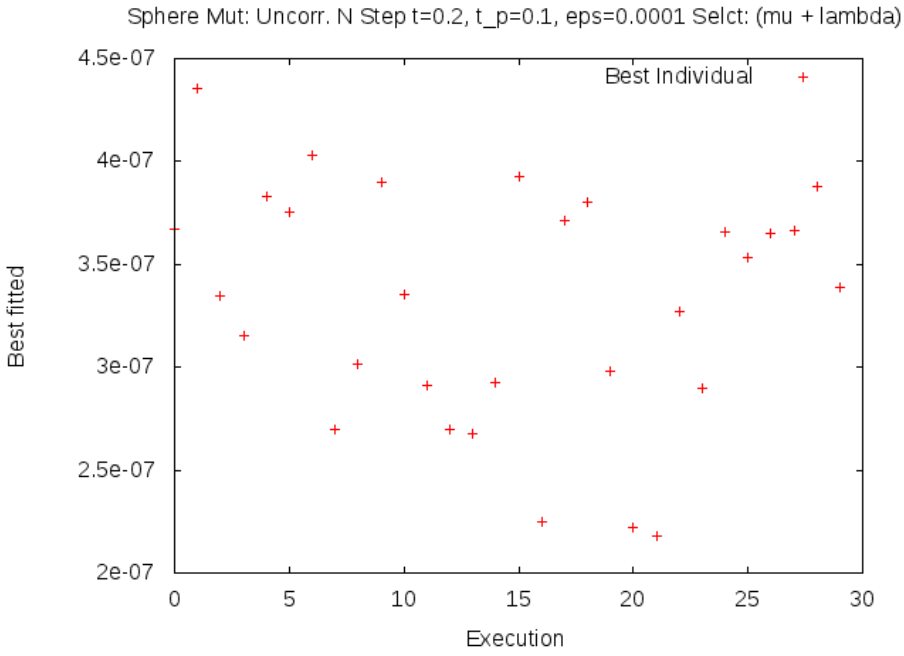

Figure 8: Sphere function with uncorrelated n-step mutation. $(\mu + \lambda)$ selection. Statistics: $minf(x^*) = 2.18e^{-7}, \overline{f(x^*)} = 3.31e^{-7}, maxf(x^*) = 4.35e^{-7}, \sigma_{f(x^*)} = 5.73e^{-8}$

4.1.5 Conclusions for the sphere model

As we can see in the previous results, for the *Sphere* test function, the four configurations of mutation and selection above have similar outcomes, stated in table (1). We can spot an slightly better performance using *one-step* over *n-step*, and using $(\mu+\lambda)$ over (μ, λ). Observe that this behaviour is somehow what we expected. The function itself have spheric symmetry and thus, using *one-step* where the iso-probability regions are hyper-spheres, is enough to achieve an accurate and fast convergence. Moreover, the greater the number of parameters, the more complex the evolutionary process is, and the convergence is eventually slower on simple and symmetric problems. On the other hand, the $(\mu + \lambda)$ scheme, which selects individuals on a larger population enabling well adapted parents to pass through the next generation, achieves better overall performance. Anyway, as we discussed, these differences in performance are not very significant, considering we are talking always on errors within the same order of magnitude.

Mutation	min	mean	max	std
Uncorrelated, one-step, (μ, λ)	$1.83e^{-7}$	$3.50e^{-7}$	$4.39e^{-7}$	$5.89e^{-8}$
Uncorrelated, n-step, (μ, λ)	$3.85e^{-7}$	$5.42e^{-7}$	$6.68e^{-7}$	$8.08e^{-8}$
Uncorrelated, one-step, $(\mu + \lambda)$	$1.55e^{-7}$	$3.08e^{-7}$	$2.37e^{-7}$	$3.37e^{-8}$
Uncorrelated, n-step, $(\mu + \lambda)$	$2.18e^{-7}$	$3.31e^{-7}$	$4.35e^{-7}$	$5.73e^{-8}$

Table 1: Statistic for the 30 executions on the *sphere* test function

Comparing the speed of convergence, and not only the final results, gives also a good insight of the efficiency of each method. In figure (9) we have run a comparison of the speed of convergence, plotting the best-fitted individual function evaluation against number of generation needed to achieve that level

of error. As we can see, results in table (1) are confirmed in the figure. *One-step* methods outperforms *N-step* method in this simple problem, probably because the complexity of adaptation of n parameter is not necessary in this symmetrical problem and lags the evolutionary process. At the same time, $(\mu + \lambda)$ schema outperforms (μ, λ) methods, since the selection pressure is bigger when the population for selection includes both parents and off-springs.

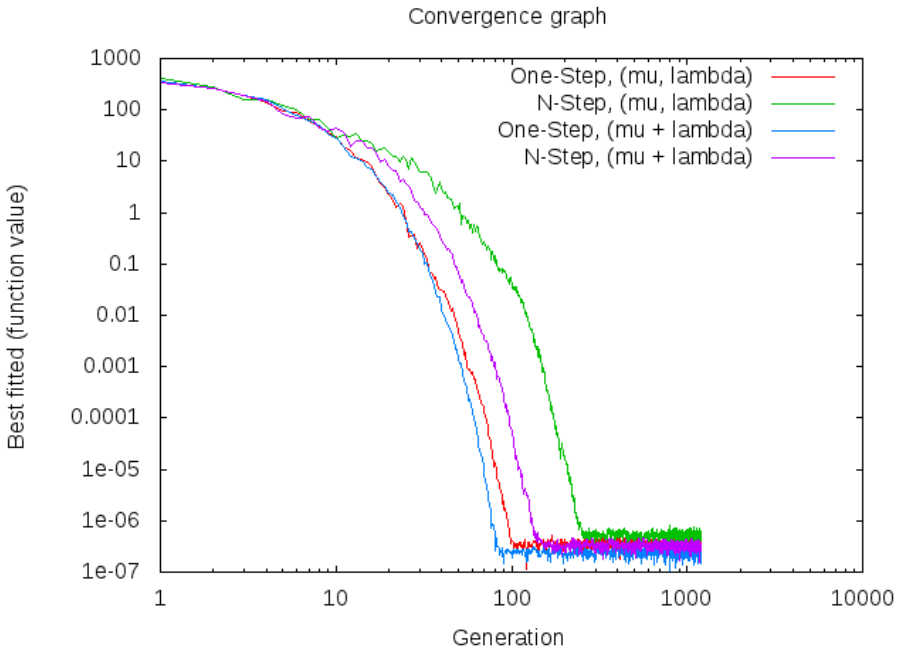

Figure 9: Speed of convergence for each scheme

CPU consumption is another interesting property to evaluate since it allows us to decide whether it is worth to implement a more complex mutation method for a given problem, or the increase in complexity does not pays off the increase in consumed CPU-time and convergence-speed. Figure (10) shows the user-time cpu consumed for each mutation and selection method. As we can see, (μ, λ) schema are less cpu-demanding methods than its counterpart $(\mu + \lambda)$, basically because the selection method (sorting, since selection is based on rank) have to act on more individuals, $(\mu + \lambda)$, instead of λ. At the same time, as expected, *N-step* schema are more time-consuming than *one-step* methods. A more clever plot to study is not the absolute cpu user-time consumed but the user-time consumed versus the achieved convergence to the final solution. Figure (11) shows these results, highlighting again the efficiency of *one-step* method for this problem. The important point in this figure is the fact that, for this particular simple problem, the cost of running the $(\mu + \lambda)$ selection method does not pays-off in time of efficiency versus the faster (μ, λ) scheme.

To properly compare the four schema, we can use formal performance measures such as *Success Rate, SR* and *Mean Best Fitness, MBF*. Since we are dealing with an optimization problem with known solution, both metrics are well defined in this case. For the *SR* metric, we have to define under which criteria we accept that we have converged to the solution. To do that, we can define a *tolerance*, and we consider the results as successful if the evaluation function is not distant from the optimum point by a distance larger than the *tolerance*. Defining a *tolerance* $10e^{-6}$, we can see in table (1) that the *SR* is 1.0 for the four methods, since in all cases, the worst individual, represented by the *max* fitness value, is smaller than the *tolerance*. Using the *MBF* metric, the *MBF* value is reflected in the *mean* column of table (1). Thus, under this performance measure, the four schema can be sorted as $one - step(\mu + \lambda)$, $n - step(\mu + \lambda)$, $one - step(\mu, \lambda)$ and finally $n - step(\mu, \lambda)$.

Figure 10: User-time CPU consumption for each scheme.

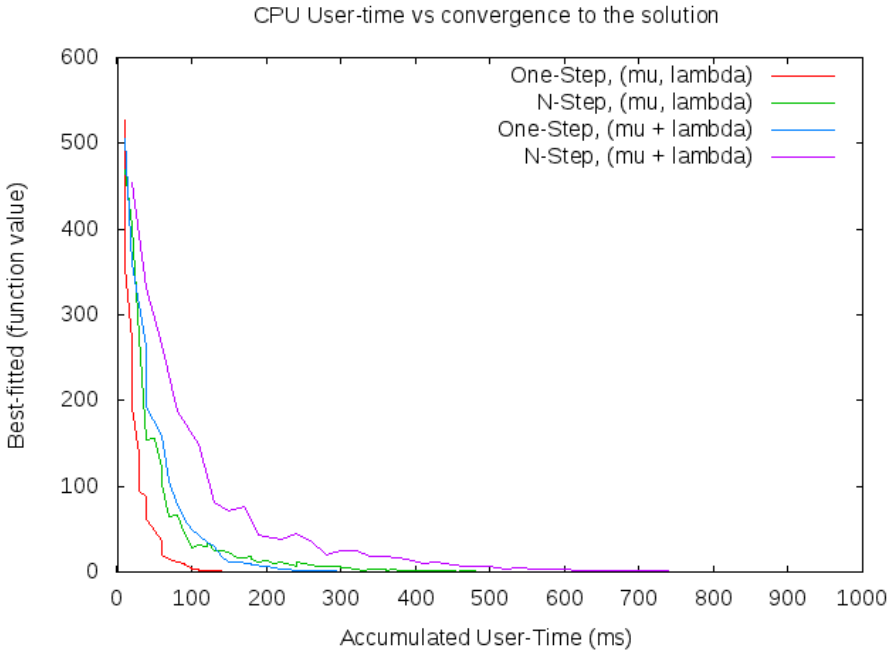

Figure 11: User-time CPU consumption vs Convergence for each scheme.

Finally, the strategies implemented in these two problem sets have two parameters that are user-defined. One is the *learning rate*, or τ and the other one is the ϵ_0, or *minimum* σ that prevent the individuals from getting stuck in the same point after the mutation in cases where the strategy parameter σ gets too small.

The *learning-rate* τ is recommended to be $\tau \propto \frac{1}{\sqrt{n}}$, where n is the dimension of the problem. For the simulation we have chosen $\tau = 0.2$ and $\tau' = 0.1$. For large values of the *learning-rate*, we have found that the algorithm convergence is seriously affected, σ values explode, the mutation operation induces

big changes in the individuals and the population after a single mutation is so different that evolutionary process have difficulties in retain the best individuals. For smaller values of the *learning-rate*, the convergence is still guaranteed, although the speed of convergence is reduced, being necessary more than 1200 generations in order to achieve reasonable results.

As for the ϵ_0 parameter, we have used 0.0001 for this problem. For unimodal function, as the one in this section, it is preferred to be small, since convergence to the unique extreme point is easier, we do not need a big variety in the population to find the path to this point. By assuring a small ϵ_0 we are guarantying a higher accuracy in the final solution, since the paths leading to the minimum point are rather easy to find. By increase the ϵ_0 value, the algorithm still advance to the surrounding points of the optimum value, but, however, once it reaches the surrounding points, due to σ values cannot fall below ϵ_0, the solution is always jumping amongst points around the minimum, instead of converging with more accuracy to the minimum value. Thus, we have observed that the accuracy of the solution in this problem is directly proportional to the value of the ϵ_0. For ϵ_0 values smaller than two decimal places, $10e^{-2}$, we achieve solutions accurate up to three decimal places, and for orders of magnitude of four decimal places, we have achieved solutions accurate up to seven decimal places.

4.2 De Jong's test function number 5

The characteristics of *De Jong's* function represent an arduous task for the designed Evolutionary Strategies. The fact of having up to 25 extreme points requires more diversity in the strategy to avoid being stuck in local minima. But the difficult problem to overcome is the shape of the function, defined in $[-65536, 65536] \times [-65536, 65536]$, where in most of the space the function has a flat value of 500 and only on a tiny region of the function domain, specifically, the subset defined in $[-32, 32] \times [-32, 32]$, the interesting extreme points are concentrated. Observe that, the region of the space where the 'real action' takes places represent a small portion of the domain. Consider a bi-dimensional uniform probability distribution, the probability of a point to lay on the interesting region where extreme points are concentrated is $\frac{64 \cdot 64}{131072 \cdot 131072} \approx 2.38 \cdot 10^{-7}$. Observe that with a configuration of $\mu = 30, \lambda = 200, generations = 1200$ we evaluate, at each execution, $7.2 \cdot 10^6$ individuals. Thus, the proportion between the number of individuals evaluated and the probability for an individual being in the 'interesting region' is quite scarce for the algorithm to perform in good conditions, specially when the region outside the $[-32, 32] \times [-32, 32]$ subset is flat. That is the reason why using $(\mu + \lambda)$ schema, where the selection process is performed on a larger base of individuals, will cast better results. Increasing the number of off-springs at each generation will have the same effect, and will improve considerably the performance of the strategy.

Aware of this difficulty, we run the simulations suggested in the problem set. Since in most cases we have not achieved a reasonable level of convergence to the solution, specially on the (μ, λ), we have run the same simulation with a larger number of off-springs at each generation, $\lambda = 800$. As above-mentioned, by increasing the number of individuals at each selection, we increase the probability that some of the individuals is near the 'interesting region' where the extreme points are located, thus, improving considerably the convergence of the algorithm.

An interesting feature that we have been forced to implement to tackle with this test function is the functionality of limiting the individuals to the range of definition of the function. As described in the previous section, the mutation process can lead to individuals which are outside the boundary of definition of the function. In the special case of the *De Jong's* test function, since most of the function have a flat 500 value, many individuals were projected outside the $[-65536, 65536] \times [-65536, 65536]$ range after mutation. Using the 'modular arithmetic' idea explained above we have successfully improved the performance of the *evolutionary strategy*, by relocating the individuals projected outside the domain of definition of the function inside the same domain.

Finally, by reducing the range of definition of the function to a smaller subset, the results would have been considerably better, not having been necessary to increase the population size, λ, to achieve SR metrics close to 1.0.

Results for this section have been made by executing the program using the following parameters files: `question6a.cfg`, `question6b.cfg`, `question6c.cfg`, `question6d.cfg` and `question6a_800.cfg`, `question6b_800.cfg`, `question6c_800.cfg`, `question6d_800.cfg`

4.2.1 (μ, λ) strategy with uncorrelated one-step mutation

Figure (12) shows the results for this configuration. As discussed above, when running simulations with the problem set parameters, specially $\lambda = 200$, the strategy is not able to converge to the solution, indeed, it is not able to converge to any of the extreme point in the function, ending in all the 30 executions stuck at the flat region of the function, outside the 'interesting region' of $[-32, 32] \times [-32, 32]$. In figure (13) we have run the same simulations, this time with $\lambda = 800$. In this case, by increasing the number of off-springs at each generation, we get some of them in the 'interesting region', and the algorithm converges quickly to the solution. Indeed, in this case, the algorithm converge all of the 30 executions to some extreme points, achieving in 25 executions the global optimum with value close to 1.0.

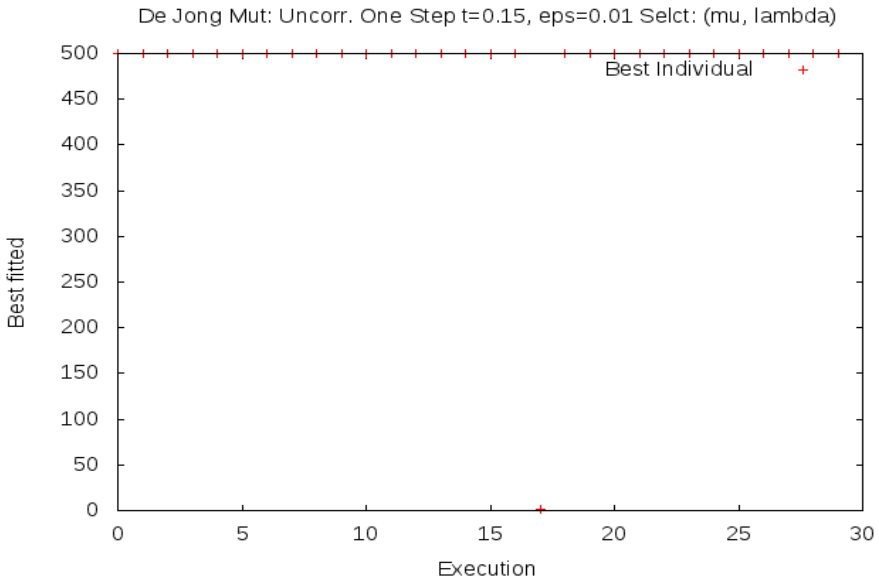

Figure 12: De Jong function with uncorrelated one-step mutation. (μ, λ) selection. Statistics: $minf(x^*) = 1.99, \overline{f(x^*)} = 483.40, maxf(x^*) = 500, \sigma_{f(x^*)} = 90.92$

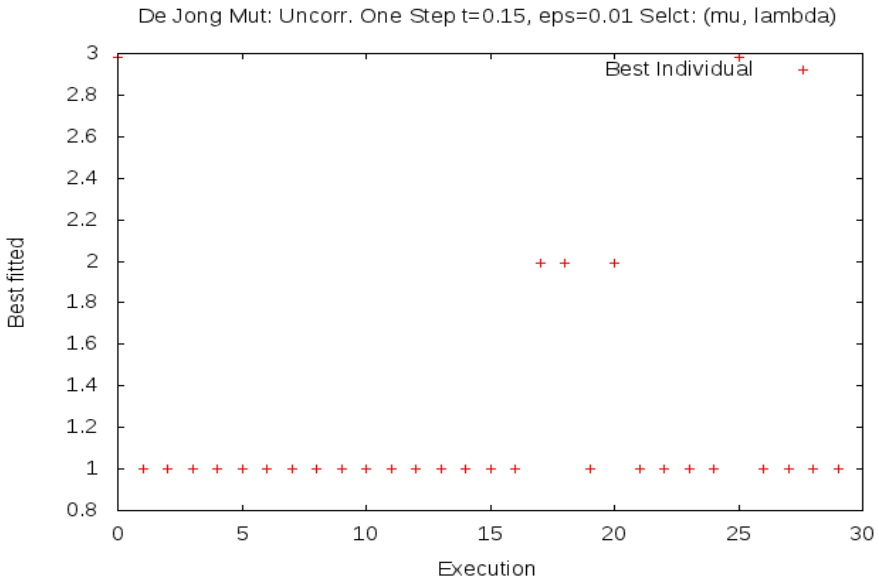

Figure 13: De Jong function with uncorrelated one-step mutation. (μ, λ) selection, $\lambda = 800$. Statistics: $minf(x^*) = 0.998, \overline{f(x^*)} = 1.23, maxf(x^*) = 2.98, \sigma_{f(x^*)} = 0.56$

4.2.2 (μ, λ) strategy with uncorrelated n-step mutation

For the *n-step* mutation, the outcome is quite similar. Figure (14) shows the results for $\lambda = 200$, as suggested on the problem set. As it happened with the previous method, the strategy is not able to converge to an extreme point. The reason is the same, the number of off-springs at each generation is not enough so that one of them falls in the 'interesting region' where the extreme points are. By increasing $\lambda = 800$, we can see in figure (15) that the strategy improves considerably, achieving always again convergence to extreme points, and, at most cases, 23 out of 30, achieving the global optimum.

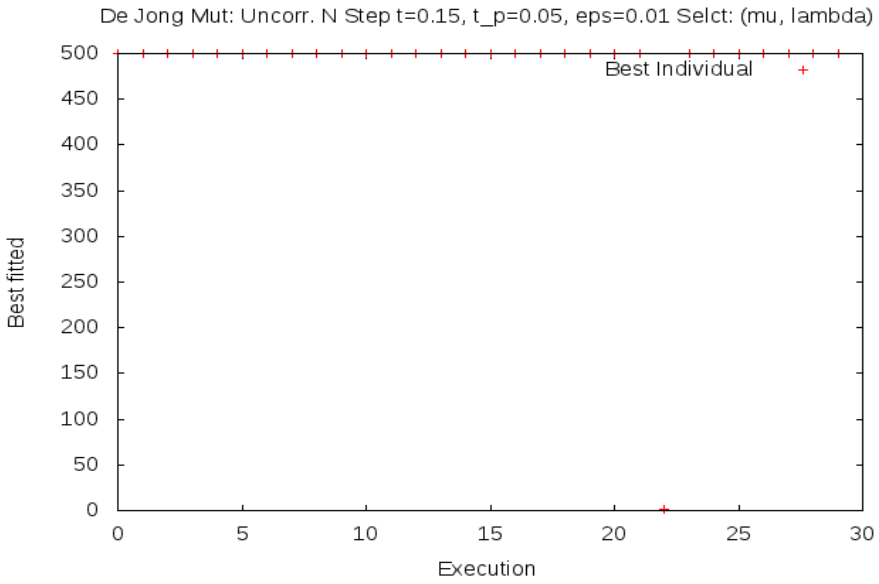

Figure 14: De Jong function with uncorrelated n-step mutation. (μ, λ) selection. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 1.12, \overline{f(x^*)} = 483.37, maxf(x^*) = 500, \sigma_{f(x^*)} = 91.08$

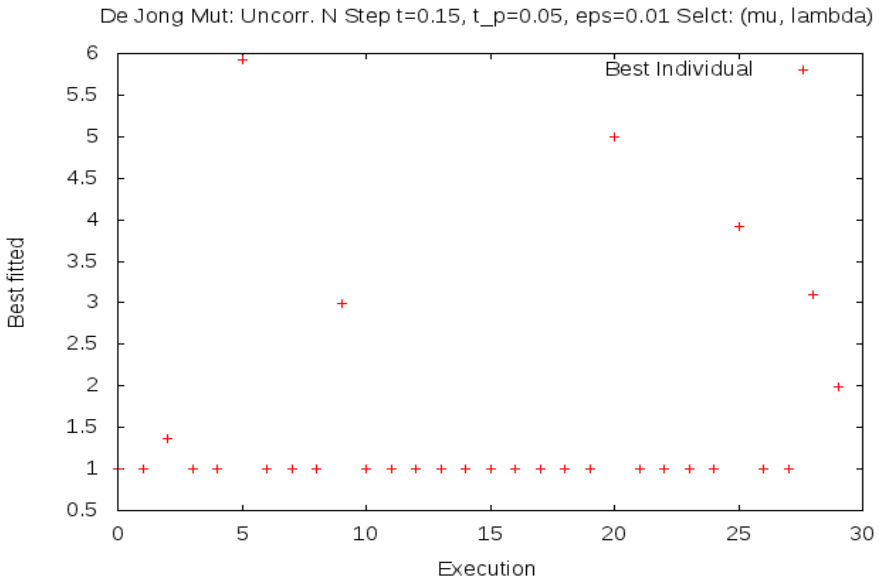

Figure 15: De Jong function with uncorrelated n-step mutation. (μ, λ) selection, $\lambda = 800$. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 0.998, \overline{f(x^*)} = 1.57, maxf(x^*) = 5.92, \sigma_{f(x^*)} = 1.29$

4.2.3 $(\mu + \lambda)$ strategy with uncorrelated one-step mutation

When we use $(\mu+\lambda)$ schema in this problem, we are, indirectly, increasing the base of individuals on which we will apply selection, thus, increasing the probability of an individual laying in an 'interesting region'. In this section we can see that even increasing slightly this base of individuals, we achieve considerably better results. Figure (16) plots the results of the strategy for $\lambda = 200$. As we can notice, the change of the selection method increase outstandingly the performance of the strategy. Now the algorithm is able to identify almost at all of the executions an extreme point, even if we do not always end up at the global optima. With this method, 14 times, out of 30, the global minimum was correctly guessed. Figure (17) shows the results of the same strategy but with a $\lambda = 800$. In this case, the algorithm correctly identifies the optimum point at each execution. The fact that the fitness function does not ends up at 1.0, but at values around 1.0 is determined by the ϵ_0 parameter, which has been defined as $\epsilon_0 = 0.01$, achieving convergence to the optimum up to a third decimal place.

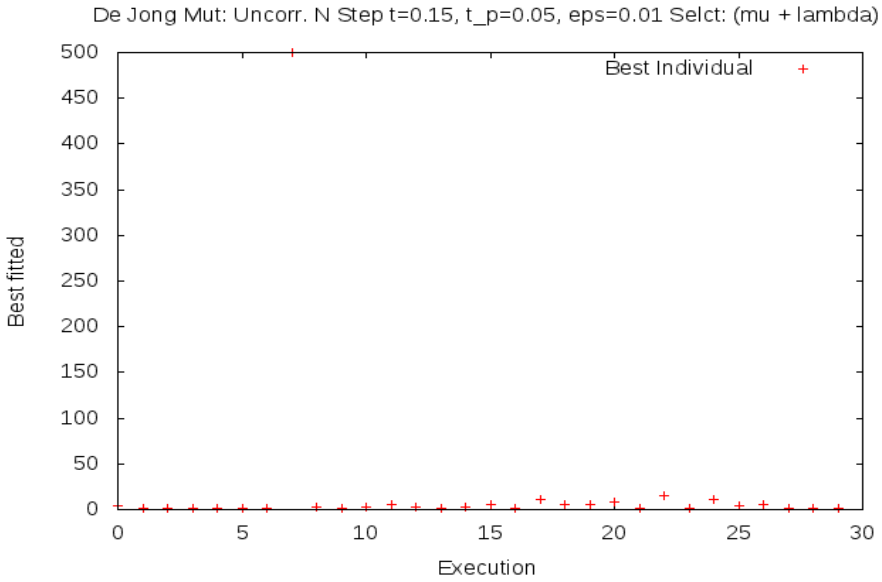

Figure 16: De Jong function with uncorrelated one-step mutation. $(\mu + \lambda)$ selection. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 0.99, \overline{f(x^*)} = 21.83, maxf(x^*) = 500, \sigma_{f(x^*)} = 90.41$

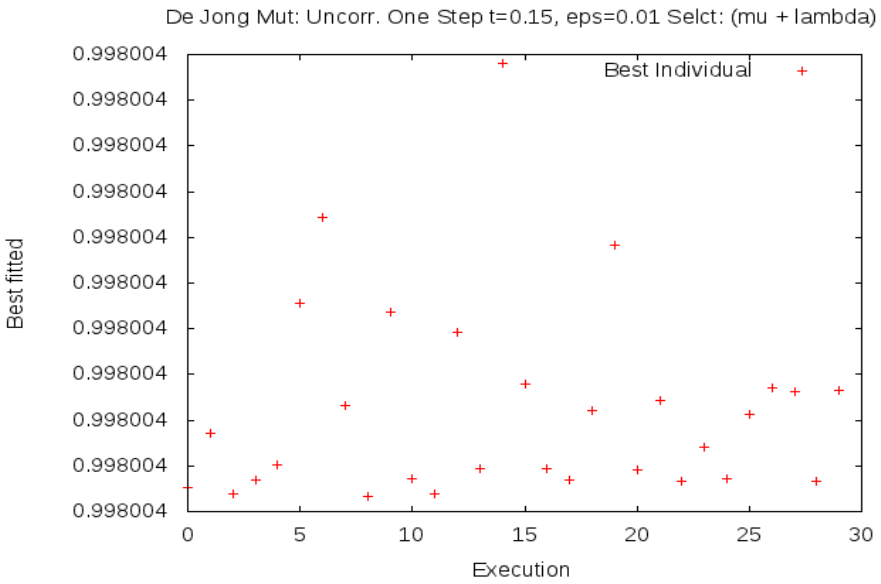

Figure 17: De Jong function with uncorrelated one-step mutation. $(\mu + \lambda)$ selection, $\lambda = 800$. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 0.99, \overline{f(x^*)} = 0.99, maxf(x^*) = 0.99, \sigma_{f(x^*)} = 0$

4.2.4 $(\mu + \lambda)$ strategy with uncorrelated n-step mutation

Results for this schema are similar to the previous section, figure (18) shows the convergence of the algorithm for the case $\lambda = 200$, and figure (19) for the $\lambda = 800$ scenario. In the first case, the algorithm has achieved convergence to one extreme point in all but one case, detecting correctly, 14 times out of 30, the global minimum. In the second one, all executions ends successfully, achieving the minimum value.

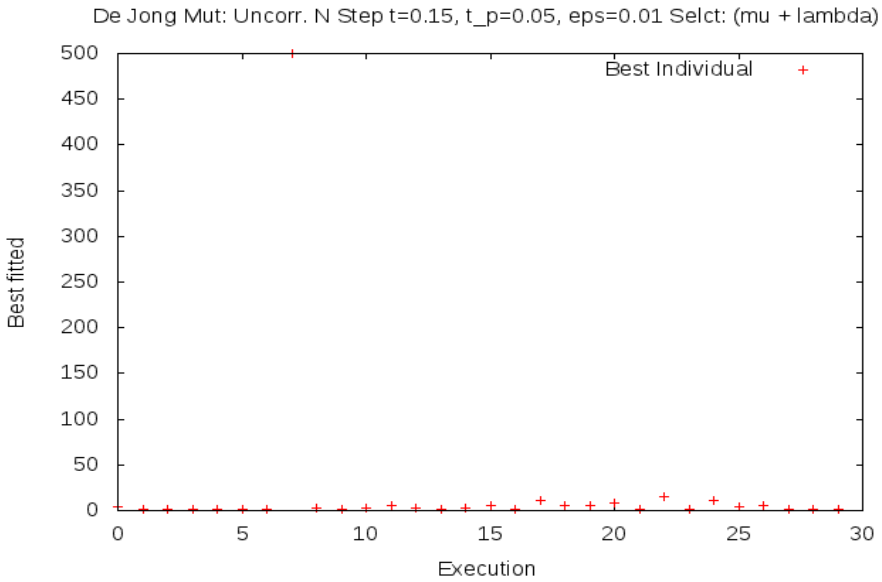

Figure 18: De Jong function with uncorrelated n-step mutation. $(\mu + \lambda)$ selection. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 1.0, \overline{f(x^*)} = 20.35, maxf(x^*) = 500, \sigma_{f(x^*)} = 90.67$

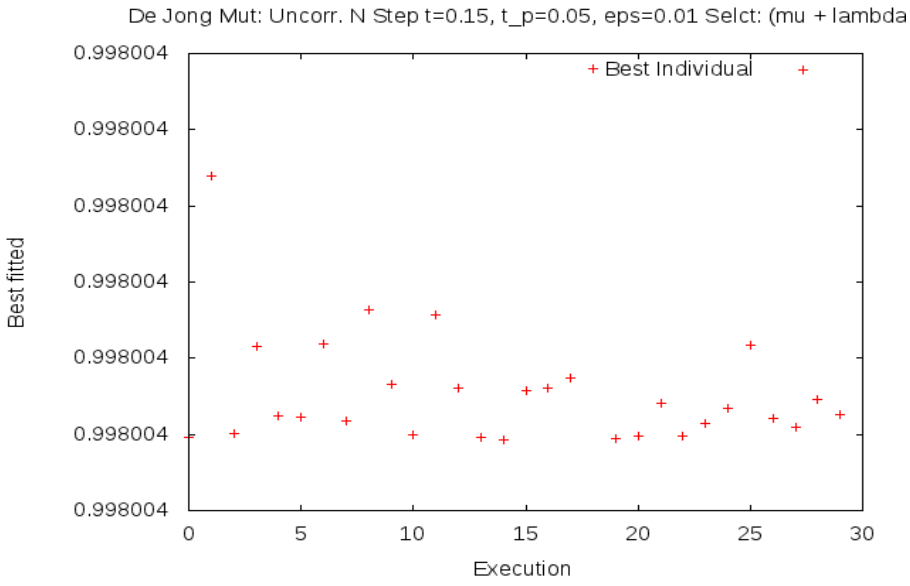

Figure 19: De Jong function with uncorrelated n-step mutation. $(\mu + \lambda)$ selection, $\lambda = 800$. Statistics: (μ, λ) selection. Statistics: $minf(x^*) = 0.99, \overline{f(x^*)} = 0.99, maxf(x^*) = 0.99, \sigma_{f(x^*)} = 0$

4.2.5 Conclusions for the De Jong model

Main conclusions for the model has been previously outlined. The difficulty of the function resides on the range of definition of the function. Most of the range $[-65536, 65536] \times [-65536, 65536]$ is a flat surface,

Mutation	$\lambda = 200$				$\lambda = 800$			
	min	mean	max	std	min	mean	max	std
Uncorrelated, one-step, (μ, λ)	1.99	483.40	500	90.92	0.998	1.23	2.98	0.56
Uncorrelated, n-step, (μ, λ)	1.12	483.37	500	91.08	0.998	1.57	5.92	1.29
Uncorrelated, one-step, $(\mu + \lambda)$	0.99	21.83	500	90.41	0.99	0.99	0.99	0.00
Uncorrelated, n-step, $(\mu + \lambda)$	0.99	20.35	500	90.67	0.99	0.99	0.99	0.00

Table 2: Statistic for the 30 executions on the *De Jong* test function

and extreme points are located in a region limited to $[-32, 32] \times [-32, 32]$. A key aspect to bypass this issue is to increase the number of individuals taking part in the selection process at each generation, either by choosing $(\mu + \lambda)$ schema or by increase the λ parameter. Both strategies have been proved successful to work around this problem, preferring increasing λ for a maximum accuracy at the expense of an important increase in CPU-time. A more detailed study would reveal the optimum λ to achieve convergence without wasting too much CPU-time.

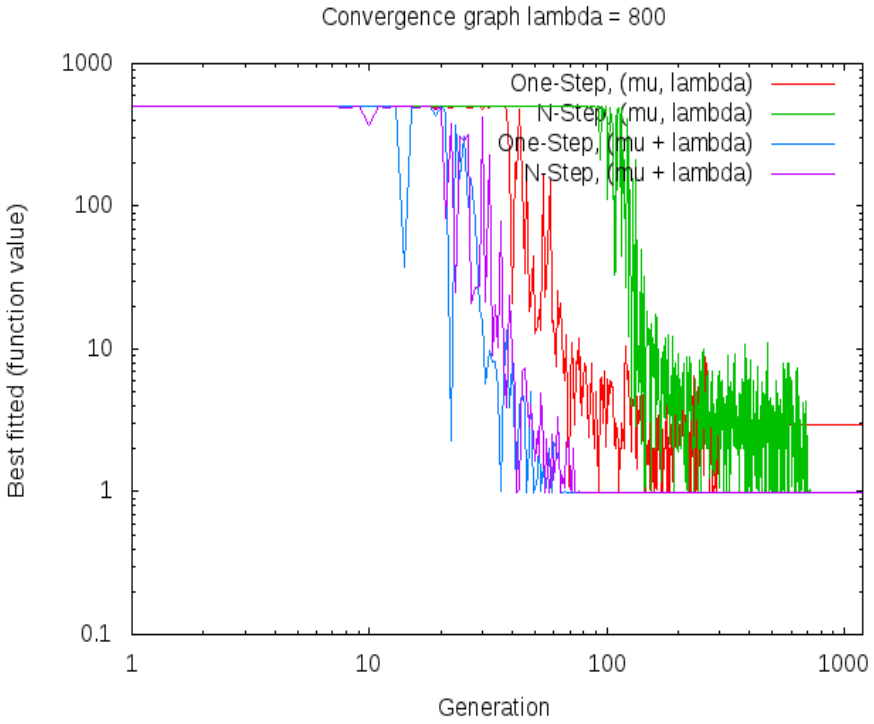

Figure 20: Evolution of the solution on number of generations

Table (2) summarizes the results for the strategies evaluated. Difference between *one-step* and *n-step* method are inappreciable. Observe again that the problem is rather symmetric in terms of the distribution of extreme points, not so for their values. $(\mu + \lambda)$ strategies outperform (μ, λ), as we have previously discussed, because of the greater number of individuals that takes places in the selection process. Finally, $\lambda = 200$ seems to be a scarce number for (μ, λ), making the strategy useless, since we are even unable to find extreme points in most of the executions. By using $(\mu + \lambda)$ we correctly identify the extreme points, reaching in most cases the global minimum. By increasing $\lambda = 800$ the algorithm correctly identifies the correct solution in all executions.

As for the parameters τ and ϵ, we have observed similar results as we did for the *sphere* model. We have employed $\tau = 0.15$ and $\tau' = 0.05$, larger values for τ tend to make the evolution parameters diverge,

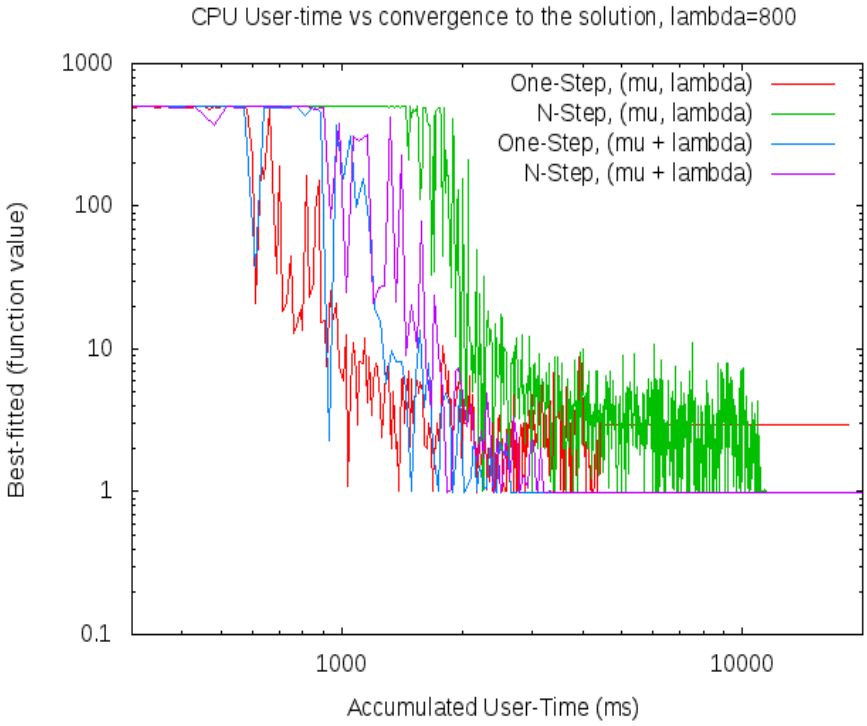

Figure 21: User-time CPU consumption vs Convergence for each scheme.

making useless the mutation. Smaller ones make evolution parameter to evolute slowly, and thus, the algorithm do not converge for low number of iterations. As for ϵ_0, we have run the simulations with $\epsilon_0 = 0.01$. Larger values for ϵ_0 identify equally the extreme points, converging to them, in a region whose size is directly proportional to this *threshold parameter*. So, choosing a small ϵ_0 we assure that once we identify the region of the space where the local extreme is located, we converge to the exact point with a decent accuracy. The evolution of σ values for this choice of ϵ_0 seems to be appropriate, σ is large when the strategy has not identified extreme points, and this allows the algorithm to explore a wider area of the space. Once extreme points are located, σ values quickly decrease to the minimum ϵ_0 to achieve maximum accuracy on the final result.

In terms of performance, $(\mu+\lambda)$ seems to be the best strategy, because of reasons previously discussed. As for the mutation type, there is not significant difference, being *one-step* methods slightly better. Even if the dimension of the problem is limited, $n = 2$, the choice of *n-step* method does not pays off, since it introduces an appreciable overhead on the convergence of the control parameters. Being again a problem with a rather symmetrical structure (the extreme points are located on a 5×5 grid), *n-step* methods does not really add any advantage to the problem. Finally, a large enough λ have been proven to be fundamental on the converge of the problem, recommending λ slightly larger than the one suggested in the problem set, still remaining in the same order of magnitude of hundreds of individuals.

With the data above, we can classify the four studied schema using the usual performance metrics. In this case we consider an execution as successful if it reaches the minimum within an accuracy of $10e^{-2}$. Table (3) gathers all the information for all execution run. These performance metrics assert again the results previously discussed, the superiority of $(\mu + \lambda)$ methods over (μ, λ) ones, the improvement of the performance by increasing λ, and the fact that *n-step* mutation does not goes along with an increase in performance.

Mutation	$\lambda = 200$		$\lambda = 800$	
	SR	MBF	SR	MBF
Uncorrelated, one-step, (μ, λ)	$\frac{1}{30} = 0.03$	483.40	$\frac{25}{30} = 0.83$	1.23
Uncorrelated, n-step, (μ, λ)	$\frac{1}{30} = 0.03$	483.37	$\frac{23}{30} = 0.76$	1.57
Uncorrelated, one-step, $(\mu + \lambda)$	$\frac{14}{30} = 0.46$	21.83	$\frac{30}{30} = 1.00$	0.99
Uncorrelated, n-step, $(\mu + \lambda)$	$\frac{14}{30} = 0.46$	20.35	$\frac{30}{30} = 1.00$	0.99

Table 3: Performance metric for DeJong function

4.3 Schwefel's function

Schwefel's test function have some properties that make it a harder target for *evolutionary strategy*. The minimum of the function is located in a point of the space where all the components have the value 420.9687. The relevant characteristics that make the function troublesome for *evolutionary strategies* are, firstly, that it is defined in a space of $d = 20$ dimensions and secondly, it is a multi-modal function, with a large number of extreme points. The first property makes more difficult the job for the ES, since the evolutionary process for a large number of parameters is cumbersome, making convergence slower. The second one can make that the ES gets stuck in some local extreme points. Concretely, for the *Schwefel's*, we have observed that the ES developed got stuck at points x where most of the components $x_i \approx 420.97$ and some of them where different from the optimum value for that coordinate. These points happen to be local extreme points where, once the algorithm converge, it is difficult to get it out of there.

Since for the suggested parameters to solve this problem set we have not achieved in most cases the correct solution, we have explored some other different configurations to fine tune our ES. In concrete, we have run executions with the suggested parameters, $\lambda = 200$, other with a reduced dimensions of $d = 10$, and other with a larger population of $\lambda = 800$.

As a general result, the problems with $d = 10$ have been correctly solved, indicating that the high dimensionality of the function is responsible for the complexity of the optimization process. Executions for $\lambda = 800$ have performed better, since they have been able to get the ES out of the local extreme points where the strategies with smaller population get stuck.

Results for this section have been made by executing the program using the following parameters files: `question7a.cfg`, `question7a-dim-10.cfg`, `question7a-lambda-800.cfg`,`question7b.cfg`, `question7b-dim-10.cfg`, `question7b-lambda-800.cfg`, `question7c.cfg`, `question7c-dim-10.cfg`, `question7c-lambda-800.cfg`, `question7d.cfg`, `question7d-dim-10.cfg`, `question7d-lambda-800.cfg`.

4.3.1 (μ, λ) strategy with uncorrelated one-step mutation

In figure (22) we can appreciate the results for the suggested configuration of the problem. As we can observe, convergence to the minimum is achieved only in 3 out of 30 executions. By increasing $\lambda = 800$, we can notice in figure (23) that results improve slightly, but still they are not good enough to use this strategy. In this case solution is achieved 6 out of 30 executions, the remaining ones ending up stuck in non-optimal extreme points. Finally, by reducing the dimensionality of the problem to $d = 10$, the success rate improves up to 18 out of 30, thus, observing the added complexity of the problem when the number of dimension is large.

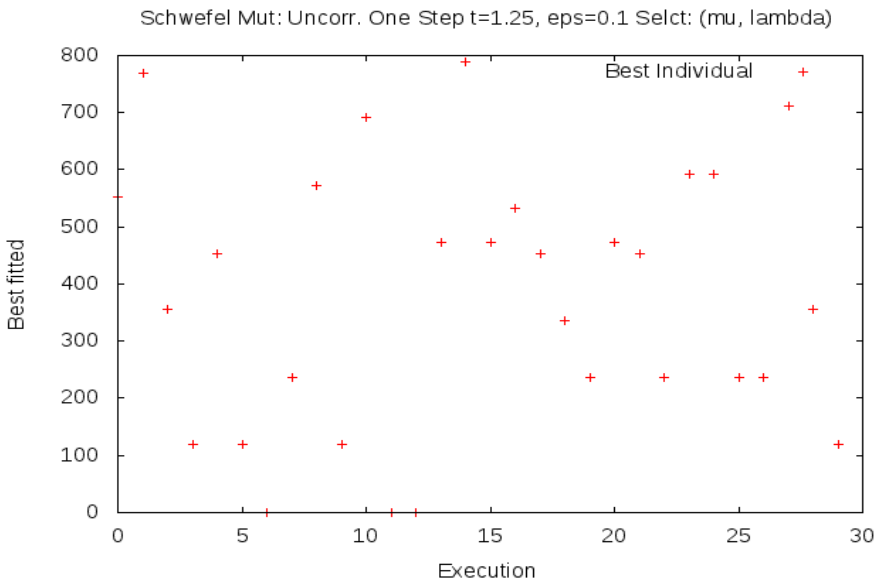

Figure 22: Schwefel function with uncorrelated one-step mutation for $d = 20, \lambda = 200$, strategy (μ, λ)

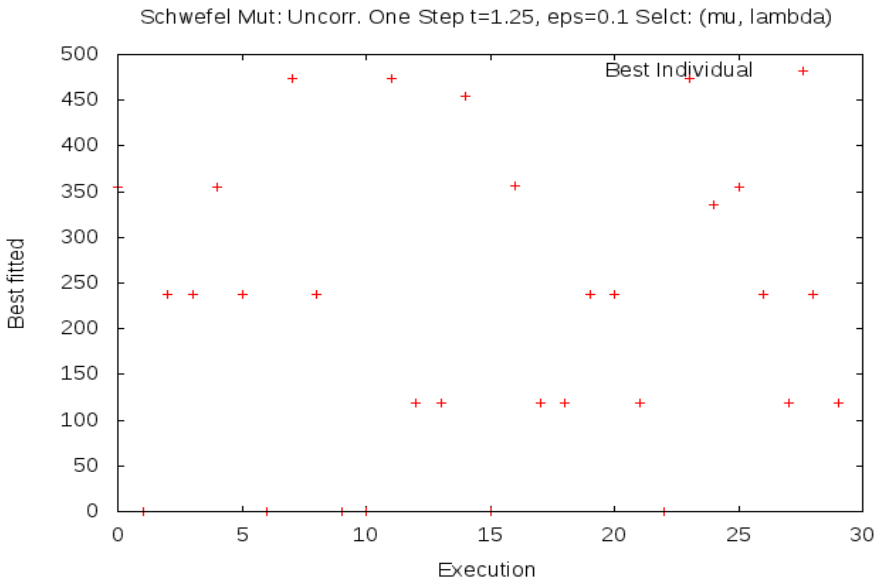

Figure 23: Schwefel function with uncorrelated one-step mutation for $d = 20, \lambda = 800$, strategy (μ, λ)

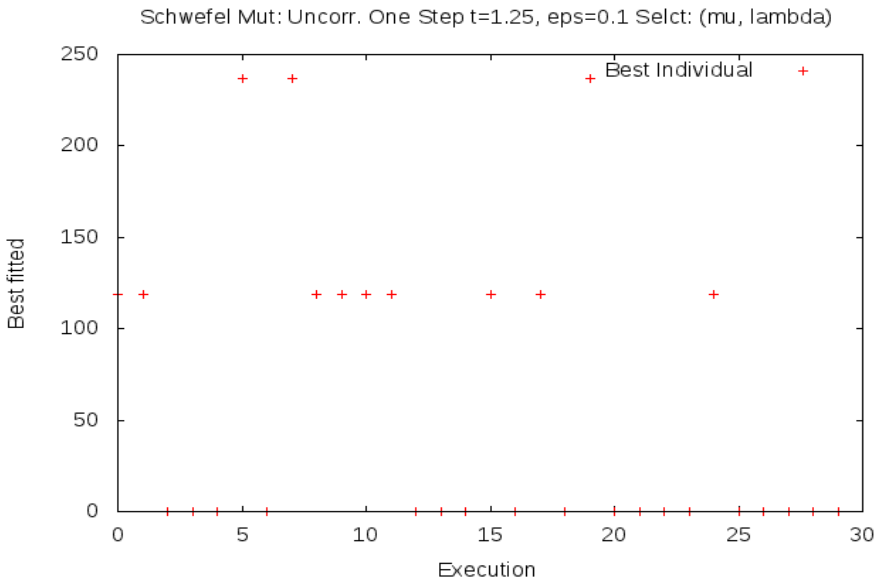

Figure 24: Schwefel function with uncorrelated one-step mutation for $d = 10, \lambda = 200$, strategy (μ, λ)

4.3.2 (μ, λ) strategy with uncorrelated n-step mutation

Results with *n-step* mutation improves the previous one. Being the *Schwefel* function a non-symmetrical one, the use of this schema improves notably the performance. In figure (25) we can see the results with the suggested configuration of the problem. Convergence to the optimum is achieved in 9 out of the 30 executions. By increasing $\lambda = 800$, in figure (26), we achieve convergence in 14 out of the 30 executions. Finally, by reducing dimensionality to $d = 10$, in figure (27), the minimum value is detected in 26 out of the 30 executions.

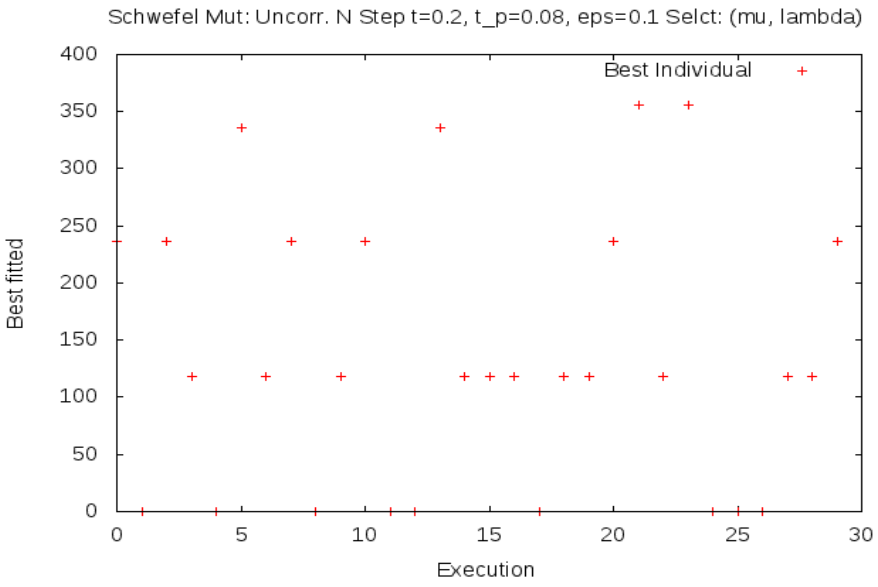

Figure 25: Schwefel function with uncorrelated n-step mutation for $d = 20, \lambda = 200$, strategy (μ, λ)

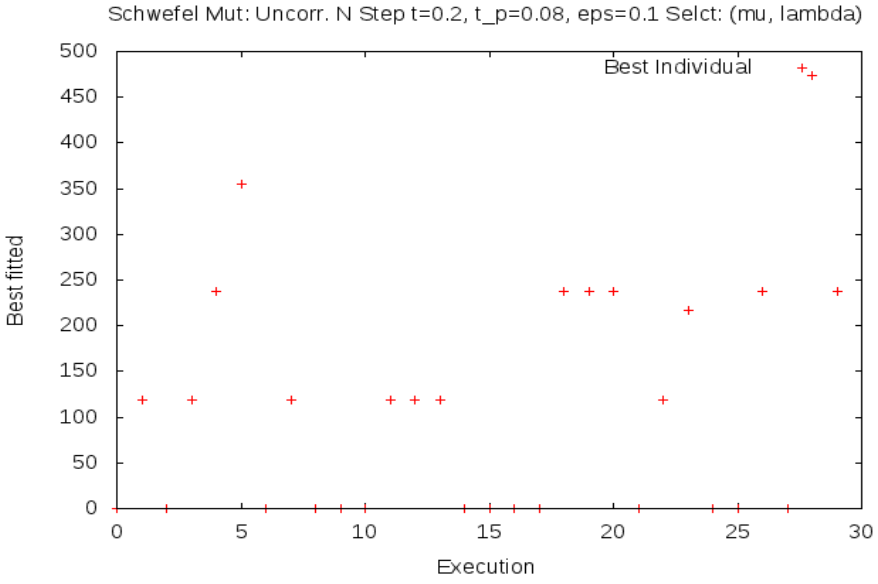

Figure 26: Schwefel function with uncorrelated n-step mutation for $d = 20, \lambda = 800$, strategy (μ, λ)

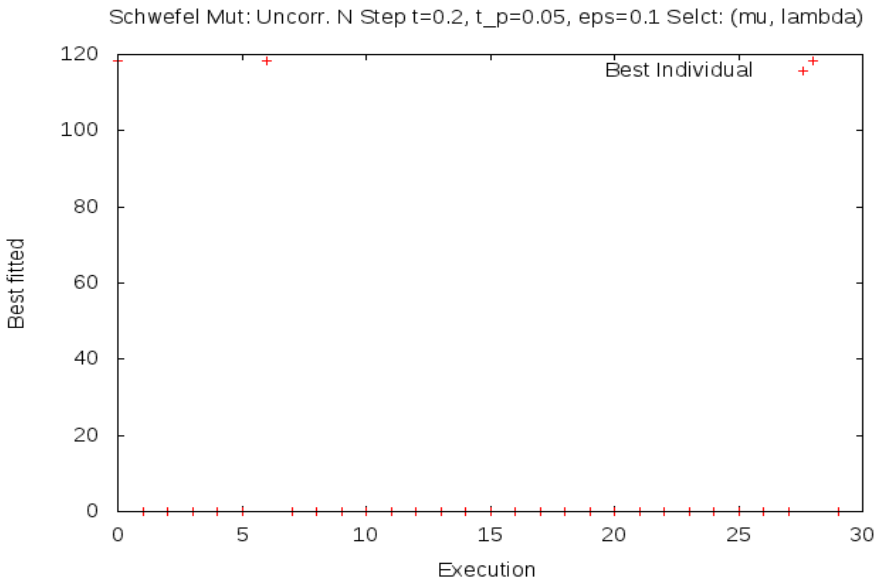

Figure 27: Schwefel function with uncorrelated n-step mutation for $d = 10, \lambda = 200$, strategy (μ, λ)

4.3.3 $(\mu + \lambda)$ strategy with uncorrelated one-step mutation

In the *one-step* schema, by using $(\mu + \lambda)$ results do not improve considerably. Figure (28) shows results for the suggested parameters, figure (29) plots results for $\lambda = 800$ and figure (30) summarizes results for a reduced dimension $d = 10$. In the three cases convergence is still very poor.

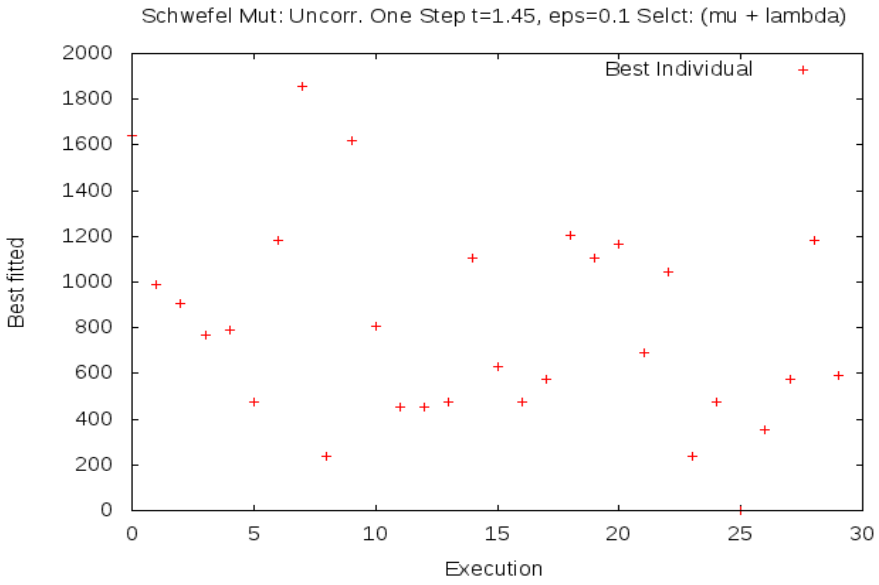

Figure 28: Schwefel function with uncorrelated one-step mutation for $d = 20, \lambda = 200$, strategy $(\mu + \lambda)$

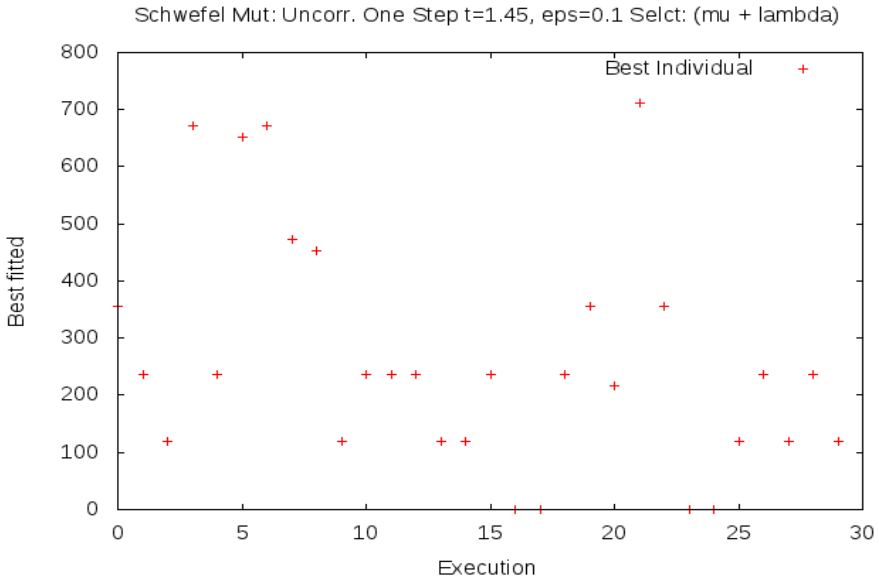

Figure 29: Schwefel function with uncorrelated one-step mutation for $d = 20, \lambda = 800$, strategy $(\mu + \lambda)$

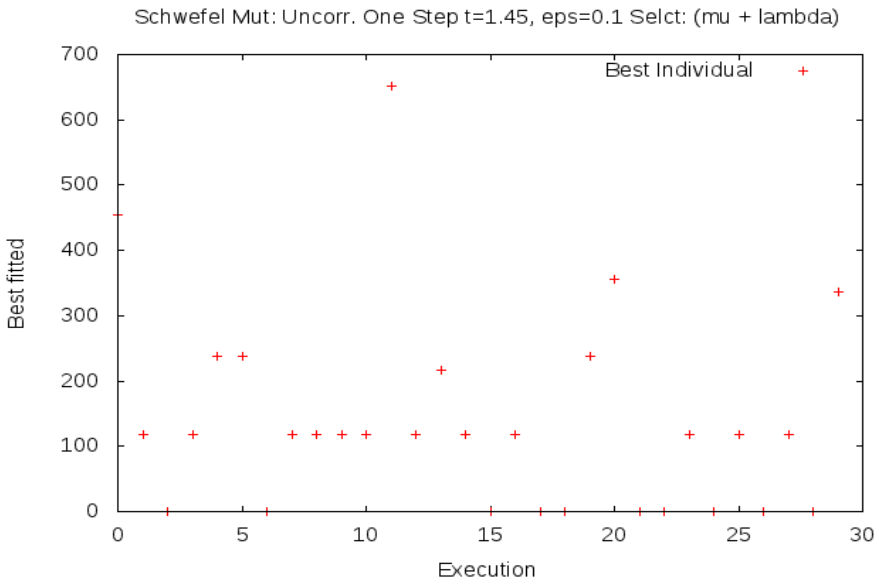

Figure 30: Schwefel function with uncorrelated one-step mutation for $d = 10, \lambda = 200$, strategy $(\mu + \lambda)$

4.3.4 $(\mu + \lambda)$ strategy with uncorrelated n-step mutation

The *n-step*, $(\mu+\lambda)$ has been proved to be the best strategy for this problem. The *n-step* mutation features outperforms the *one-step* schema in this non-symmetrical problem. At the same time, the $(\mu+\lambda)$ selection process increases the base of individuals, making the bigger selection pressure more effective. Figure (31) shows results for the suggested parameters, where 24 out of 30 executions finished successfully. Figure (32) shows the improvement when increasing the $\lambda = 800$, where 26 out of 30 execution achieved the optimum.

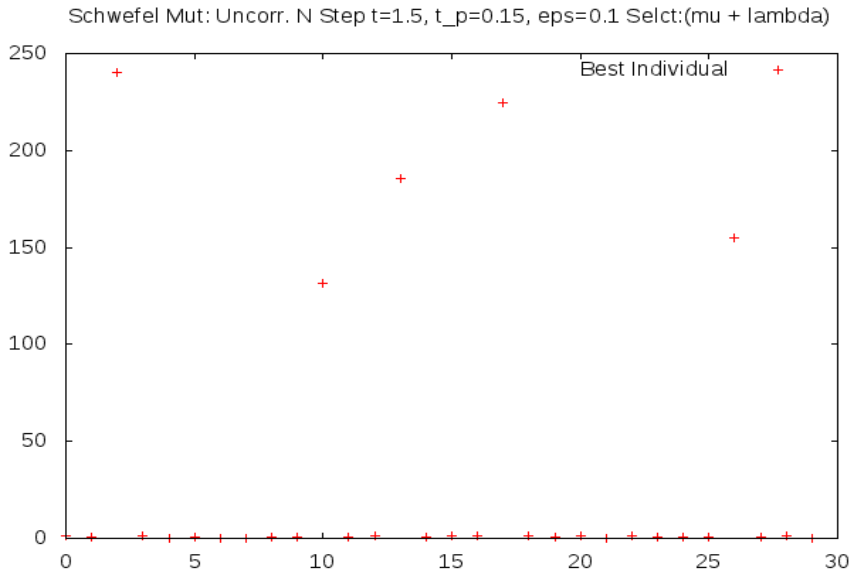

Figure 31: Schwefel function with uncorrelated n-step mutation for $d = 20, \lambda = 200$, strategy $(\mu + \lambda)$

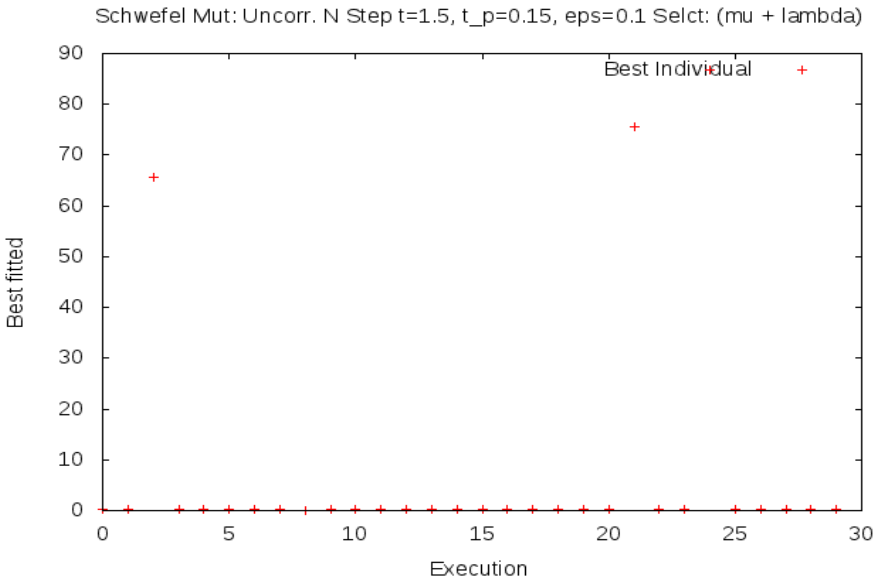

Figure 32: Schwefel function with uncorrelated n-step mutation for $d = 20, \lambda = 800$, strategy $(\mu + \lambda)$

4.3.5 Conclusions for the Schwefel's function

As we have previously guessed in the introduction, for this problem, *n-step* methods seem to be the right choice, since we are dealing with a non-symmetrical function with a large number of extreme points. The increase in the number of individuals within the selection process is equally important, and that is why $(\mu + \lambda)$ schema performs outstandingly better than (μ, λ). In most cases, the executions ended up in some local minimum, but the convergence to the optimum requires a high variation of the population to avoid getting stuck in local extreme points. Indeed, we have observed that most of the executions finished stuck in points of the space where most of the dimension were $x_i = 420.96$ and some individual dimensions were different from this value.

Results of the simulation for each schema are summarized in table (4), and performance metrics for each experiments are showed in table (5). Observe that for the case of *n-step*, $(\mu + \lambda)$ results are worse than for the *n-step* experiment. This is not usually the case, and this particular results are due to a bad choice of the *learning rate* parameter.

The choice of the τ, τ', ϵ_0 parameters have, in this problem, the same implications as in the previous ones. ϵ_0 controls the accuracy of the solution, the smaller the parameter, the closer the algorithm can approach to the solution. The choice for τ, τ' in this problem has been more difficult, and we have tried many configurations until we have reached values that make the *evolutionary strategy* converge properly to the solution. For the *n-step*, we have used parameters considerably different to the ones recommended in the text-book. The behavior of the strategies with the *learning rate* is similar to the experiment above, with large values of τ, the strategy parameters σ tend to explode, making the evolution useless, and with small values convergence is slow, ending up, most of the time, in local extreme points. A more detailed study for the choice of the *learning rate* parameters should be done for the most complex test functions.

	$\lambda = 200$				$\lambda = 800$				$d = 10$			
Mutation	min	mean	max	std	min	mean	max	std	min	mean	max	std
one-step, (μ, λ)	0.026	376.40	789.62	232.00	0.01	211.94	473.774	155.13	0.00	59.22	236.88	80.81
n-step, (μ, λ)	0.03	136.90	355.37	118.43	0.01	109.91	473.78	127.36	0.00	11.85	118.45	36.14
one-step, $(\mu + \lambda)$	0.02	802.23	1857.28	443.45	0.01	262.55	710.64	205.28	0.00	138.18	651.42	153.17
n-step, $(\mu + \lambda)$	0.11	72.771	240.09	31.85	0.10	7.76	86.83	23.32	.	.	.	.

Table 4: Statistic for the 30 executions on the *Schwefel's* test function

	$\lambda = 200$		$\lambda = 800$		$d = 10$	
Mutation	SR	MBF	SR	MBF	SR	MBF
one-step, (μ, λ)	$\frac{3}{30} = 0.10$	376.40	$\frac{6}{30} = 0.20$	211.94	$\frac{18}{30} = 0.60$	59.22
n-step, (μ, λ)	$\frac{9}{30} = 0.30$	136.90	$\frac{14}{30} = 0.46$	109.91	$\frac{26}{30} = 0.86$	11.85
one-step, $(\mu + \lambda)$	$\frac{1}{30} = 0.03$	802.23	$\frac{4}{30} = 0.13$	262.55	$\frac{10}{30} = 0.33$	138.18
n-step, $(\mu + \lambda)$	$\frac{24}{30} = 0.80$	72.77	$\frac{26}{30} = 0.86$	7.76	.	.

Table 5: Statistic for the 30 executions on the *Schwefel's* test function

5 Summary and Conclusions

We have tested three function of increased difficulty: mono-modal symmetrical function (*sphere*), multi-modal, low-dimensional, quasi-symmetrical function (*De Jongs*) and highly-multi-modal, highly-dimensional asymmetrical function (*Schwefel*). *N-step* methods have been proved to be preferred for highly-dimensional and asymmetric functions, where the cost of evolving n control parameters pays off the improvement in performance. $(\mu + \lambda)$ methods have been shown superior to (μ, λ), since selection is made on a larger base of individuals. However, depending on the values of μ, λ parameter, and the difficulty of the test function, the increase in cpu user-time consumed does not justify the increase in performance, specially for easy problems as the *sphere* model.

Parameters such as *learning rate*, τ, τ' have to be carefully chosen, specially in difficult problems. The convergence and stability of the *evolutionary strategy* depends greatly on them. Finally, ϵ_0 parameters can be used to set a desired accuracy when converging to the optimal value.

Correlated mutation methods have been not tested on this problem set. As we have seen, in most cases, the increase in the number of control parameters carries more complexity for the evolutionary process, making harder or slower, at sometimes, the convergence. In the case of correlated methods, the number of control parameters to evolve explodes with the number of dimensions, needing to evolve n σ-parameters and $\frac{n(n-1)}{2}$ α parameters, making the method awkward even for low dimensional problems.

In all of the three problems we have obtained good levels of SR metrics, with a sample of 30 executions. For the first two test function, optimal configuration of selection and mutation method have achieved outstanding $SR = 1.0$ rating. For the more difficult third problem, we have obtained $SR = 0.86$. There would be, however, large room to improve the SR in this case, just by increasing the population size.

6 Appendix

A Program Compilation

Program building process has be done using the standard make gnu tool. To compile and link the program just invoke the `make` command on the root directory. The binaries will be found either under the `bin/debug` or `bin/release` directories, depending on the `dbg` flag specified at `config.mk` file. To link the program and generate the binary, we need the Boost Libraries properly installed in our system. The `libraries.mk` file contains the list of libraries that we need for the linking process.

B Program Execution

The program `ES.bin` is invoked from the command line, passing all required parameters needed to the execution of the algorithm. For a comprehensive list of parameters, we can invoke:

```
victor@kokomero:~/ES.bin --help
Allowed options:
  --conf_file arg           Path to the configuration file. If omitted,
                            arguments are read from command line
  --epsilon arg             Minimum value for the std
  --test_function arg       Test function: sphere, dejong, schwefel
  --search_space arg        value of the search space, if x, search will be
                            limited to [-x, x] for each dimension
  --dimension arg           Dimension parameter n, for Sphere and Schwefel test
                            function
  --mutation arg            Mutator operator, one of: onestep (uncorrelated
                            one-step), multistep (uncorrelated n-step),
                            correlated (correlated)
  --tau arg                 Tau parameter for mutation
  --tau_p arg               Tau prima parameter for n-step mutation
  --selection arg           Selection operator, one of: replacement (mu,
                            lambda) or competition (mu + lambda)
  --mu arg (=30)            Number of individuals in the population
  --lambda arg (=200)       Number of offsprings created at each generation
  --generations arg (=1200) Number of generations for the termination condition
  --executions arg (=10)    Number of independent executions
  --test                    Run mutation and crossover tests for debuggin
                            purposes
  --help                    print out help message
```

All configuration parameters can be specified in a `.cfg` file, as the one below, `question7a.cfg`:

```
mutation=onestep
tau=1.25
tau_p=0.25
epsilon=0.1
dimension=20
search_space=500
test_function=schwefel
selection=replacement
mu=30
lambda=200
generations=1200
executions=30
```

An example of execution would be thus:

```
victor@kokomero:~/ES.bin --conf_file question7a.cfg
```

which read the configuration parameters from `question7a.cfg` file and will return the information about the experiment, along with the partial result of each execution:

```
victor@kokomero:~/ES.bin --conf_file question7a.cfg
Simulation Parameters:
Function to minimize: Schwefel with 20 dimensions
Search space for each dimension: [-500, 500]
Number of Executions: 30

ES Parameters:
Max number of Generations: 1200
mu: 30
lambda: 200
Mutator operator: Uncorr. N Step t=1.5, t_p=0.15, eps=0.1
Cross Over operator: Discrete for object part, intermediary for strategy parameters
Parent Selection operator: Random
Survivor Selection operator: Ranking
Selection method: (mu + lambda)
Execution number : 1
Generation: 240 best individual: Genotype: [421.743, 421.604, 421.296, 420.595, 421.132, 422.06
Generation: 480 best individual: Genotype: [420.479, 420.363, 420.764, 421.355, 422.165, 421.09
Generation: 720 best individual: Genotype: [419.896, 420.861, 420.93, 419.764, 421.345, 419.618
Generation: 960 best individual: Genotype: [421.384, 420.678, 421.283, 421.595, 421.211, 421.81
Generation: 1200 best individual: Genotype: [420.65, 421.239, 421.085, 421.042, 420.856, 419.92
...
```

References

[1] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975. AAI7609381.

[2] Robert Demming and Daniel J. Duffy. *Introduction to the Boost C++ Libraries, Volume I - Foundations.* Datasim Education BV, 2010.

[3] Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf*, (c):1–43, 2005.