Practical optimisation, Coursework 2

MLMI candidate H801D

Preamble

We implement two optimisation algorithms - Evolution Strategy (ES) and Tabu Search (TS) - and test the impact of the algorithms' parameters on their performance in minimizing Rana's function.

Unless otherwise specified, general reference is [Par20] - see also [SK06].

1 Introduction

We consider *Rana's function*

$$f_n(x) := \sum_{i=1}^{n-1} x_i \cos\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} - x_i + 1|}\right) + (1 + x_{i+1}) \cos\left(\sqrt{|x_{i+1} - x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} + x_i + 1|}\right)$$

in dimension n, and investigate the following problem for $n = 5$:

Problem 1.1. minimize $f_5(x)$ s.t. $-500 \leq x_i \leq 500$ for $i = 1, \ldots, 5$.

Our task was to apply two stochastic optimisation algorithms to Problem 1.1, and compare their performances for a fixed number of evaluations of f_5. We chose to implement an ES algorithm (which seemed more naturally adapted to the continuous nature of the problem than a Genetic Algorithm) and a Tabu Search algorithm, hoping that it would be well-suited to the high multimodality of Rana's function. We coded both algorithms from scratch using Matlab (see [Mat20] for documentation); all code can be found in the Appendix.

When allocating time and effort, we chose to try to implement a large variety of options and variants, including personal ideas, rather than to focus on tailor fitting the parameters to our special case to achieve the best possible performances - we understood the exercise as being more about comparing various algorithms and the effects of their parameters (and reaching conclusions that could be generalized to other similar problems) than purely solving Problem 1.1. Of course, we nonetheless tried to find reasonably good parameters.

In the same spirit, we tried to make our code as modular and flexible as possible - hence slightly slower to run than if it had been optimized for Problem 1.1.

In Section 1.1, we make a few preliminary observations regarding Rana's function, and how its properties might affect our results. In Section 2 (respectively, Section 3), we detail our implementations of the ES (respectively, TS). Our experimental protocol is described in Section 4, and the results in Section 5. We conclude in Section 6. It is followed by the Appendix, which contains all listings of our code, except those for the different CMA-ES algorithm (see below), which we attach as an archive.

[1] See [JY13] for a rather comprehensive list of such functions.

1.1 Observations regarding Problem 1.1

Rana's function is a classical benchmark function for optimisation algorithms[1], due to it being *non-separable* (it cannot be written as a sum of functions of one coordinates) and highly *multimodal* (*i.e.* it has many local optima). On the other hand, it is smooth, uniformly scaled (the norms of the derivatives w.r.t. each coordinate are of the same magnitude), and its behaviour in the neighborhood of each local minimum is quite simple: hence we can expect our algorithms to easily converge to a local minimum once it has found solutions close to it, but to struggle to find the global minimum among the local ones. In other words, exploration, rather than exploitation, should be key here.

In [ADGV14], the minimum of the five dimensional Rana's function over the domain $\tilde{D} := [-512, 512]^5$ is found with high precision; it is approximately equal to -2046.8, and the corresponding solution is on the boundary of $\tilde{D}$. In problem 1.1, we work with a slightly different domain $D := [-500, 500]^5$, but based on the scale at which f_5 varies, as well as the two dimensional case and the experimental results obtained, we can again expect the global minimum point x^* to be found close to the boundary of D, and $p^* := f_5(x^*)$ to belong to $[-2046.8, -1995]$.

In what follows, we try to distinguish what stems from properties of the algorithms themselves, and can be expected to generalize to other benchmark functions, from what is specific to Problem 1.1.

2 Evolution Strategy

We kept to the general description from the Lecture notes [Par20] (we retain the same conventions), with the following variable parameters whose impact we later investigate:

- Number N_0 of randomly (uniform distribution) selected points in the domain $D = [-500, 500]^5$ in the 0-th generation.
- Number of selected parents μ and offsprings λ, as well as number k_{vet} of *veterans*: the veterans are the k_{vet} fittest parents from the previous generation that survive to the next generation. Setting $k_{vet} = 0$ yields a (μ, λ) strategy, setting $k_{vet} = \mu$ yields a $(\mu + \lambda)$ strategy.
- Variance σ_0^2, where $\sigma_0^2 \cdot I_5$ is the covariance matrix of all first generation solutions.
- Recombination operators: one can choose between pairwise recombination (two parents per offspring) and global recombination (offsprings descend from all parents at once). Moreover, one can choose between discrete and intermediate recombination (with weight coefficient $\omega = 0.5$) for the control variables. Intermediate recombination is always applied to the strategy parameters (reasons are exposed below).
- Constraints management: as the constraints are very simple (the domain D is a full-dimensional cube), the penalty function approach seemed unnecessary. Due to the shape of D, non-mutated offsprings (both with discrete and intermediate recombination) are always feasible, but mutated solutions can escape D. We have implemented two different strategies: either Reject unfeasible mutated children, or Redraw random mutations until one of them leaves the mutated child feasible. The choice of strategy impacts the density of solutions near the boundary and the strategy parameters; this is discussed later on.
- Convergence criterion (either relative or absolute difference within the population) and tolerance ϵ. This has little impact, as our algorithms almost never converge before the maximum number of evaluations is reached.

We have also implemented the following features (that are not described in [Par20]), which we describe in more details:

- A simple version of the idea of *islands* (see for example [Whi01]). We consider several

populations, or islands, within which the usual selection, reproduction and mutation processes happen for each generation. Every T_{mig} generation, between the selection and the reproduction phase, we randomly exchange $\lfloor \mu \cdot \#\text{islands} \cdot c_{mig} \rfloor$ solutions between the various islands - they *migrate* - where $0 < c_{mig} < 1$ is the *migration rate* and $T_{mig} \in \mathbb{N}$ is the *migration frequency*.

Maintaining several distinct populations can keep the algorithm from prematurely converging to a local minimum, as dissimilar good solutions can coexist in different islands, and hence improve global search abilities.

More sophisticated variants of that idea exist: for example, migrations can be allowed only between certain islands. Different topologies for the graph of connected islands can have different effects (as mentioned in [GCZ+15]).

- A modification of the pairwise recombination operator by adding a mechanism which we call *endogamy*. To create an offspring, we would usually select with uniform probabilities two parents S_1 and S_2 among the μ best solutions. Instead, we (uniformly) select only one parent. We then select the other one among the remaining $\mu - 1$ best solutions with modified probabilities:

$$P(\text{second parent} = S_2 \mid \text{first parent } = S_1) = C \min\left(0.5, \max\left(0.1, \frac{1}{1 + \frac{||x_1 - x_2||}{c_{endogamy}}}\right)\right),$$

where x_i is the vector of control variables associated to S_i for $i = 1, 2$, the constant $c_{endogamy} > 0$ is the *endogamy scale* and C is a renormalization constant such that the probabilities sum to 1.

This decreases the likelihood of two solutions producing offsprings if they are far away from each other. The philosophy behind that feature is the same as for the islands model: we hoped that it would allow several good solutions to develop semi-independently, in order to improve global search capacities.

- Alternative mutation operator, where if x is the vector of control variables of an offspring, the mutated vector is $x' = x + \eta$, where $\eta = C \cdot t$ and C is the same square root of a covariance matrix as in the standard case, but t is an n-dimensional random variable whose i.i.d. coordinates follow Student's t-distribution with 1 degree of freedom instead of the normal distribution.

  To improve the global search capacities of the algorithm, we might simply use a normal distribution with increased variance; indeed, the ES's selection and recombination process is such that we only need a few points close to the optimum for the entire population to quickly adapt. However, too high a variance might penalize the exploitation, as solutions might mutate too much on average to precisely locate local minima - they would keep oscillating around it[2].

  As Student's t-distribution is roughly similar to the normal distribution, but with heavier tails, most mutations should be close to what would normally be the case (hence exploitation should not be too affected), but a few mutations should be more extreme - those outliers might find new local minima and thus improve the exploration.

- Making use of Rana's function's smoothness, we devised an alternative mutation operator where if x is the vector of control variables of an offspring, the mutated vector is $x' = x + \eta + \xi c_{grad} \nabla f_5(x)$ for some parameter $c_{grad} > 0$, where η is the same random mutation vector as in the standard case (see [Par20]), $\nabla f_5(x)$ is the gradient of Rana's function in x and ξ is an exponential random variable of rate parameter 1. The idea is to keep the global search

[2]Though the severity of that problem depends on the recombination operator - global intermediate recombination should not be as affected, due to the averaging process.

[3]In a sense, it is to ES what Intelligent Design is to Natural Selection (this is just a catchy name; we do not believe in Intelligent Design, nor do we pretend that this is some groundbreaking metaheuristic).

capacities of ES while increasing the speed of convergence to local optima by adding a small drift towards them[3].

In classical gradient descent, the optimal stepsize in the direction of the gradient is approximated using techniques such as the backward search or second order information. As those would be too costly in terms of evaluations, we have added an element of randomness using ξ: thus some mutations are likely to be almost optimal given the graph of the objective function in the neighborhood of x, while others will be quite aberrant (the rationale being that ES rewards having a small number of good solutions among very bad ones, rather than all of them being mediocre). We understand that this modified algorithm does not fully respect the spirit of the exercise, as it does not treat Rana's function as a black box anymore - see it as a bonus of sorts.

To keep things "fair", we consider that each evaluation of ∇f_n is equivalent to n evaluations of f_n w.r.t the maximum number of evaluations allowed, as the cost of computing each partial derivative $\partial_{x_i} f_n$ is of the same magnitude as that of computing f_n.

- We finally implemented an entirely different type of ES from the one described in the Lecture notes [Par20], called *Covariance matrix adaptation ES* (CMA-ES), which has been reported to perform very well on multimodal optimization problems (see [HK04]).

  The main ideas are the following: at each generation g, the μ best solutions are selected. Let m_g be the (weighted) average of their control variables vectors. We then let the λ offsprings $\{x_i^{g+1}\}_{i=1}^{\lambda}$ be

  $$x_i^{g+1} = m_g + z_i,$$

  where the perturbations z_i are *i.i.d.* random vectors drawn from a Gaussian distribution $\mathcal{N}(0, \sigma_g^2 C_g)$ and the covariance matrix C_g and the scale factor σ_g are shared by the entire population at generation g.

  Two other global variables, the evolution path $p_{c,g}$ (which records information regarding the evolution of m_g through successive generations) and the conjugate evolution path $p_{\sigma,g}$ (which records information regarding the successive $C_g^{\frac{-1}{2}}(m_{g+1} - m_g)$, where $C_g^{\frac{-1}{2}}$ is the unique square root of C_g^{-1} with non-negative eigenvalues), are then updated.

  Lastly, the covariance matrix C_g is updated as a weighted combination of C_g, the empirical covariance matrix of the vectors z_i, and $p_{c,g+1} \cdot p_{c,g+1}^T$, and σ_g is increased if the norm of $p_{\sigma,g}$ is larger than a certain threshold and decreased otherwise. The goal is for C_g and σ_g to adapt to the general direction and scale of past successful moves (with more weight given to recent moves), in order to capture the local geometry of the function to be optimized without having to directly compute derivatives.

  A detailed description of CMA-ES can be found in [HO01], as well as a more complete exposition of its philosophy.

Remark 2.1. *While we independently came up with three of the features above (use of the gradient or of Student's t-distribution in mutations, endogamy) and could not find anything equivalent in the literature, we expect those ideas to have been explored before (especially the use of the gradient) - we are simply not familiar enough with the domain to find the proper references.*

2.1 Practical implementation details

In the Lecture notes [Par20], the covariance matrix $C = (c_{i,j})$ associated to a given solution as a strategy parameter is described in terms of variances $\sigma_i^2 = c_{i,i}$ and angles

$$\alpha_{i,j} = \frac{1}{2} \tan^{-1}\left(\frac{2c_{i,j}}{c_{i,i} - c_{j,j}}\right).$$

However, we think that this description might be incomplete[4], as $C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\tilde{C} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$ would for example have the same variances and angles, yet correspond to different covariances.

[4]Specifically, we suspect that the σ_i^2 should be the eigenvalues of C instead.

We chose to follow [Bä96, Chapter 2] instead: using orthogonal diagonalization for symmetric matrices, we factorize $C = RD^2R^T$, where D is a diagonal matrix whose entries are non-negative, and R belongs to the special orthogonal group $SO(n)$ (in problem 1.1, we have $n = 5$). We associate to each solution the matrix RD (which is the transpose of the Cholesky factor of C) as a strategy parameter, rather than the covariance matrix C itself. For $1 \leq i < j \leq n$, denote by $R_{i,j}(\alpha_{i,j})$ the matrix associated to the (counterclockwise) rotation of angle $\alpha_{i,j}$ in the plane spanned by e_i and e_j. Let $\tau = \frac{1}{\sqrt{2\sqrt{n}}}$, $\tau' = \frac{1}{\sqrt{2n}}$ and $\beta = 0.0873$ be as in the Lecture notes and [Bä96, Chapter 2].

The mutation operator then maps a solution (x, RD) to a mutated solution $(x+RDz, \tilde{R}RD\tilde{D})$, where $z \sim \mathcal{N}(0, I_n)$ is a normal Gaussian vector, $\tilde{D} = (\tilde{d}_{i,j})$ is a diagonal matrix whose diagonal entries are

$$\tilde{d}_{i,i} = \exp(\tau'\aleph_0 + \tau\aleph_i)$$

for *i.i.d.* random numbers $\aleph_i \sim \mathcal{N}(0, 1)$ (for $i = 0, \ldots, n$), and

$$R = \prod_{1=1}^{n} \prod_{i<j\leq n} R_{i,j}(\alpha_{i,j}),$$

with $\alpha_{i,j} = \beta\aleph_{i,j}$ for *i.i.d.* random numbers $\aleph_{i,j} \sim \mathcal{N}(0, 1)$. Note that any element of $SO(n)$ can be written as such a product of rotation matrices; note also that as rotation matrices do not always commute, in general

$$\prod_{1=1}^{n} \prod_{i<j\leq n} R_{i,j}(\alpha_{i,j}) \cdot \prod_{1=1}^{n} \prod_{i<j\leq n} R_{i,j}(\beta_{i,j}) \neq \prod_{1=1}^{n} \prod_{i<j\leq n} R_{i,j}(\alpha_{i,j} + \beta_{i,j}),$$

hence we do not exactly recover the mutation mechanism from the Lecture notes (though we do not expect this to have any major impact).

The (either pairwise or global) intermediate recombination operator for strategy parameters[5] then computes the average of the covariance matrices of the combined solutions to compute the covariance matrix of the offspring, then recovers the associated matrix RD using Cholesky decomposition.

An issue can arise when using the Redraw constraints strategy (see above) with small λ and μ (*i.e.* $\lambda \leq 50$). As the number of generations becomes fairly high, mutations can accumulate and the eigenvalues of the covariance matrices associated to solutions can become absurdly large, which causes most mutations to make the solution unfeasible. The Redraw strategy does not penalize such covariance matrices (as a mutation that makes a solution unfeasible is simply ignored, and a new mutation is randomly drawn). To keep this from happening, whenever more than $100 \cdot 2^n$ mutations have been drawn for a given offspring without any of these mutated solutions being feasible, we multiply the associated covariance matrix by 10^{-2} to rescale it.

A few remarks regarding the code: we have used 'parfor' parallel loops everywhere possible, in order to capitalize on the parallelizable nature of the algorithm. Moreover, as we have implemented as many options as we could for the sake of experimenting, our code is rather dense. Hence, in order to maintain readability and as it is a slightly different algorithm, we have coded the CMA-ES in another file than the standard one (see the Appendix).

3 Tabu Search

We followed the general description from the Lecture notes [Par20]. There are fewer parameters to be set than for the ES, namely:

- The initial stepsize δ_0 and the stepsize reduction coefficient α.
- The short term memory size N_{STM} and the

[5] We have not coded any discrete recombination operator for strategy parameters, as our implementation, with matrices RD rather than parameters $\alpha_{i,j}$ and σ_i^2, lends itself poorly to it - conversely, using parameters would have made global intermediate recombination harder to define properly. Besides, it does not seem to be a particularly reasonable mechanism and performs poorly, according to the Lecture notes [Par20].

medium-term memory size N_{MTM}.

- The Search Intensification (SI) threshold T_I, Search Diversification (SD) threshold T_D and Step Size Reduction (SSR) threshold T_R.

- The grid ratio $r \in \mathbb{N}$. We implement the long term memory (LTM) as follows: we divide the search space D in r^n sub-cubes of length $\frac{1000}{r}$, and keep track of which of these areas have been already visited. When diversifying, we pick a (uniformly chosen) random point in one of the areas that have not been visited yet, unless all areas have already been visited, in which case we reset the LTM[6].

As constraints are simple (no equality, convex domain, etc.), we just let unfeasible neighbours be tabu. We say that the algorithm has converged if the stepsize becomes smaller than a chosen threshold. Whenever the algorithm gets trapped (*i.e.* all neighbours of the current point are tabu), we simply erase the STM and resume. This happens very rarely in dimension $n = 5$.

We have also implemented the following additional features:

- *Concentric* TS, that was defined for discrete problems by Drezner in [Dre02] but easily lends itself to the case of continuous control variables.

  It works as follows: whenever we move to a new solution x either at the start of the algorithm, when intensifying, when diversifying, or when reducing the stepsize, as well as after a regular move if that new solution is the best solution visited yet, we let x be the new *center*. Then any regular move (*i.e.* not intensifying, diversifying or associated to a reduction in stepsize) must increase the Euclidean distance to x; in other words, all points that are closer to x than the current solution are tabus (instead of those in the STM).

  The idea behind that feature is that it might force the algorithm to move away from previously visited local optima more efficiently than simply using the STM, which can fail to prevent looping behaviours, as long as the loops are longer than the size of the STM, in the hope that it might improve global search capacities.

- We have also implemented a feature of our own invention[7] based on the same general intuition as the concentric TS.

  Let x be the current point. As with classical TS, the tabu neighbouring solutions are simply those in the STM. However, instead of ranking allowed moves by the value of the objective function (and picking the best), we rank them by the value of the following function (here, $n = 5$):

  $$g(\tilde{x}) := f_n(\tilde{x}) + \sigma(x) c_w \frac{\tilde{x} - x}{||\tilde{x} - x||} \cdot \left(\frac{m - x}{||m - x||} \right)^T,$$

  where $\tilde{x}$ is an allowed neighbour of x, the point m is the average of the points in the STM[8], the scale factor $\sigma(x) := \text{std}\left(\{f_n(\tilde{x_1}), \ldots, f_n(\tilde{x_k})\}\right)$ is the standard deviation of the values of f_n at the non-tabu neighbours $\{\tilde{x_1}, \ldots, \tilde{x_k}\}$ of x, and $c_w > 0$ is the "wanderlust constant".

  The idea is to encourage the algorithm to move away from the general direction of recently visited solutions to explore new areas and hence improve its global search ability, but in a "softer" way than the concentric TS does.

[6] More subtle mechanisms that took into account the number of successful move in each area appeared to yield no visible improvement in performance.

[7] Again, similar ideas might have been developed before - we simply failed to find references in the literature.

[8] A short term memory of a different size than the one used to define the tabu moves can be used.

4 Experiments

We test various parameters combinations, with an upper limit of $10'000$ evaluations of Rana's function. For each configuration, we run the algorithm 100 times, with different random seeds for each run (but the same sequence of seeds for each configuration). We record the total running time t, as well as the mean $\overline{p}$ and standard deviation ν of the set $\{p_1, \ldots, p_{100}\}$, where p_i is the best solution found in run i. We also maintain for each run an archive that tracks the best L dissimilar solutions, as in the Lecture notes [Par20].

As there are too many parameters to simultaneously modify all of them, we choose for each algorithm a "reasonable" baseline configuration that has been shown to perform relatively well in preliminary tests, then measure the effect of letting one (or a few) parameters vary from that baseline. When varying parameters, we do not necessarily follow a linear or geometric scale, but rather try to explore both small and large values for a given parameter.

We also run the algorithm a few times in dimension $n = 2$ to obtain illustrative figures.

We additionally compare the performance of our algorithms to that of two "naive" strategies: grid-search (evaluate f_5 at the $6^5 = 7776$ vertices of a grid of length 200) and uniform random search (pick $10'000$ points in D from a uniform distribution).

4.1 ES experiments

We use the following configuration as our baseline:

N_0	λ	μ	σ_0^2	k_{vet}
1000	450	90	0.1	0
P/G	CVRO	SPRO	Constraints	Options
Pair	Discrete	Intermediate	Redraw	None

Figure 1: Baseline configuration for ES.

Here, "P/G" stands for Pairwise/Global (choice of recombination strategy), "CVRO" for Control Variables Recombination Operator, "SPRO" for Strategy Parameters Recombination Operator, "Constraints" can be either Redraw or Reject (see Section 2 for more details).

We measure the effect of varying the following basic parameters (all parameters that are not mentioned are the same as in the baseline case):

- Initial population N_0.
- Number of offsprings λ, while keeping the same ratio $\frac{\lambda}{\mu} = 5$.
- Number of parents μ, while leaving λ unchanged.
- Initial variance σ_0^2.
- Number of veterans k_{vet}.

We then test different recombination operators:

- Pairwise, CVRO=Intermediate, SPRO=Intermediate.
- Global, CVRO=Discrete, SPRO=Intermediate.
- Global, CVRO=Intermediate, SPRO=Intermediate.

Finally, we test the following advanced options:

- Reject constraints strategy.
- Student's t-distribution-based mutation vectors.
- Gradient-enriched mutations, with different values of c_{grad}.
- Endogamy, with different endogamy scales $c_{endogamy}$.
- Islands model, with different numbers $N_{islands}$ of islands and constant migration rate c_{mig} of 0.2 and migration frequency T_{mig} of 1. For each island, there are $\left\lfloor \frac{\lambda}{N_{islands}} \right\rfloor$ offsprings and $\left\lfloor \frac{\mu}{N_{islands}} \right\rfloor$ parents each generation, with initial population of size $\left\lfloor \frac{N_0}{N_{islands}} \right\rfloor$.
- Islands model with $N_{islands} = 3$ and different migration rates and frequencies.

Finally, we test the CMA-ES algorithm. Most of its parameters have a precise probabilistic signification, and are not meant to be altered by the user (see [HO01] or [HK04]). We only try different λ, with fixed ratio $\frac{\lambda}{\mu} = 2$ (as advised in [HK04]).

4.2 TS experiments

We use the following configuration as baseline:

δ_0	N_{STM}	N_{MTM}	r	α
200	7	5	3	0.9
T_I	T_D	T_R	Options	
15	25	30	None	

Figure 2: Baseline configuration for TS.

We then measure the effect of varying the following parameters (all parameters that are not mentioned are the same as in the baseline case):

- Initial stepsize δ_0.
- Stepsize reduction coefficient α.
- STM size N_{STM}.
- MTM size N_{MTM}.
- Grid ratio r.
- Different combinations of thresholds T_I, T_D and T_R.

We also test the impact of the following options:

- Concentric TS.
- "Wanderlust" TS (described above), with different wanderlust constants c_w.

5 Results and analysis

All computations were done in MATLAB, on an Intel® Core™ i7-8650U Processor under Windows. As a convention, we give all results with a precision of 0.1. However, total running times (for 100 runs), which we give in seconds, are mostly to be used as points of comparison; they should not be overly trusted, as differences of up to 20% were observed between two runs using the exact same parameters and random seeds (probably due to the computer running some background processes). Likewise, choosing a different set of random seeds can create differences of up to 5 in mean performance (and about 3 in standard deviation) over the 100 runs[9]. Hence, we do not consider differences of less than ~ 10 between the mean performances of two configurations to be meaningful.

In the tables below, all parameters that are not specified are the same as in the baseline cases described above. We keep tracks of the best $L = 20$ dissimilar solutions using the algorithm described in the Lecture notes [Par20]. In the notations of the notes, we let $D_{min} = 20$ and $D_{sim} = 1$. We also record some additional information that we would have discarded to improve memory efficiency in a "real world" situation, but which we needed to perform some tests.

The (uniformly distributed) random search yielded an average best solution of -1486.0 over 100 runs, with a standard deviation of 91.3 and total running time of 6.2 seconds. This serves as a useful comparison point to keep in mind. Both of our baseline configurations significantly outperform this naive search, which is encouraging.

The grid search did much better, with a best objective function value found of -1857.1, outperforming the TS (but not the ES). However, we feel that this is not representative of the efficiency of the method, but rather an artifact due to the particular properties of Problem 1.1. Indeed, among the points tested by the grid search are many boundaries points, and we have seen in Subsection 1.1 that many good (with particularly low values) minima of Rana's function are very close to the boundaries of D. This has been confirmed by the results of our experiments - the best solutions we have found over our many runs, such as $x := (-498.48, -497.40, -495.78, -499.00, 406.69)$ whose objective function value $f_5(x)$ is approxi-

[9] For "reasonable" configurations - extreme configurations can be more dramatically affected.

mately -1.979, were all close to the boundary.

5.1 Evolution Strategy Results

Below are the results for the baseline configuration of the ES algorithm.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Baseline	-1897.4	43.8	115.0

Figure 3: Results for the baseline ES configuration, with "S.d." being the standard deviation and the total running time (for 100 runs) t being in seconds.

In Figure 4, we see that the ES algorithm functions as intended in dimension 2. The first generation of the ES (after selection among the initial population) is rather spread out, though already more concentrated near the boundaries. At generation 5, a few areas of interest have been located and are being explores. At generation 20, all the population is concentrated around a local minimum to which it slowly converges. In Figure 5, the 20 best dissimilar solutions found during that run are shown.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$N_0 = 90$	-1871.5	72.9	136.6
$N_0 = 450$	-1888.6	51.2	134.2
$N_0 = 2000$	-1896.9	37.4	110.7
$N_0 = 4000$	-1883.1	37.3	87.7
$N_0 = 9000$	-1570.1	74.1	19.9

Figure 6: Results for various sizes of initial population N_0.

We see in Figure 6 that allowing 10 to 20 percent of computational resources to the initial population (in a sense performing a preliminary random search of sorts) helps the global search capacities by pre-locating areas of interest. Less than that and the ES tends to converge to a mediocre local optimum, while much more takes away resources from the ES itself and hurts the convergence phase. Note that a relatively wide range of values can be chosen without performance being hurt too severely. As the dimension increases (and hence the density of samples decreases for a given N_0), we can expect large N_0 to become less competitive.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$\lambda = 10,\ \mu = 2$	-1189.8	357.6	$333,6$
$\lambda = 30,\ \mu = 6$	-1340.3	405.9	502.5
$\lambda = 60,\ \mu = 12$	-1794.5	167.7	108.8
$\lambda = 120,\ \mu = 24$	-1848.2	93.9	151.9
$\lambda = 225,\ \mu = 45$	-1882.8	80.6	169.9
$\lambda = 900,\ \mu = 180$	-1825.4	44.5	152.4
$\lambda = 3000,\ \mu = 600$	-1433.6	87.6	133.4

Figure 7: Results for varying numbers of offsprings λ and fixed ratio $\frac{\lambda}{\mu} = 5$.

As seen in Figure 7, very small λ and μ cause premature convergence, as the population lacks genetic diversity and quickly gets concentrated around some mediocre local minimum. The increased number of allowed generations also negatively impacts running time. Conversely, if the population gets overly large, the number of evaluations per generation becomes too large and the number of generations too small - convergence does not have enough time to happen. Overall, population sizes λ and μ seem to be the among the most impactful parameters.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$\lambda = 450,\ \mu = 10$	-1800.7	94.2	143.2
$\lambda = 450,\ \mu = 30$	-1885.5	65.6	155.9
$\lambda = 450,\ \mu = 60$	-1897.1	52.9	158.6
$\lambda = 450,\ \mu = 150$	-1843.3	56.1	168.2
$\lambda = 450,\ \mu = 225$	-1718.5	97.1	179.1

Figure 8: Results for varying numbers of parents μ and fixed number of offsprings λ.

Choosing a small number of parent μ corresponds to increased selection pressure, as visible in Figure 8. If μ is too small, global search is jeopardized as it will typically favor a handful of solutions in the best area currently discovered, discarding slightly worse solutions that might be in a promising neighborhood. Conversely, large μs improve global search but slow down the convergence, as bad solutions are not rooted out as quickly. Note that it seems to be less impactful that the number of offsprings λ (as it is when creating new offsprings that the algorithm truly explores new areas).

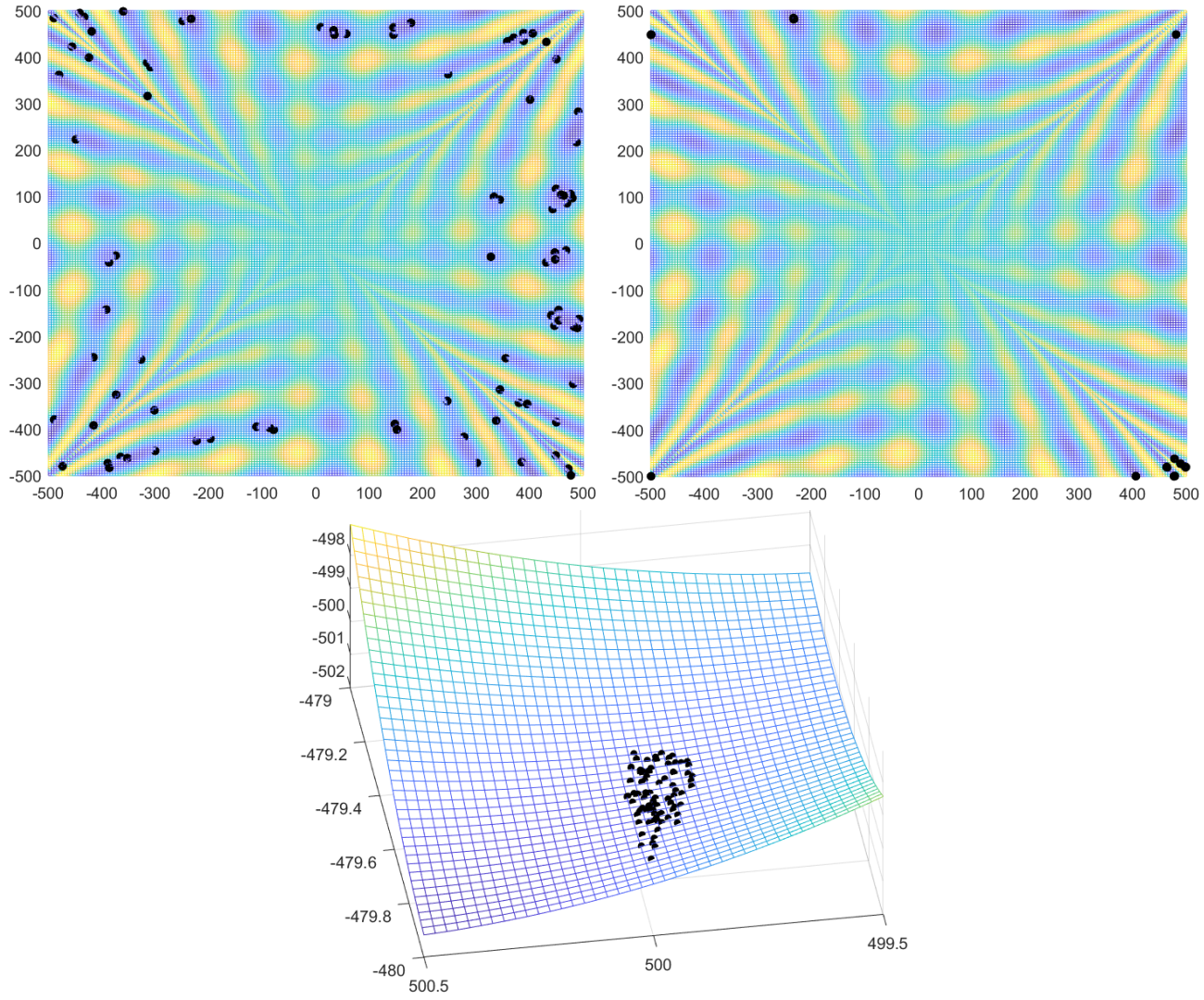

Figure 4: Generations 1 (after the first selection among the initial, randomly drawn population), 5 and 20 of the ES algorithm using the baseline configuration in dimension 2.

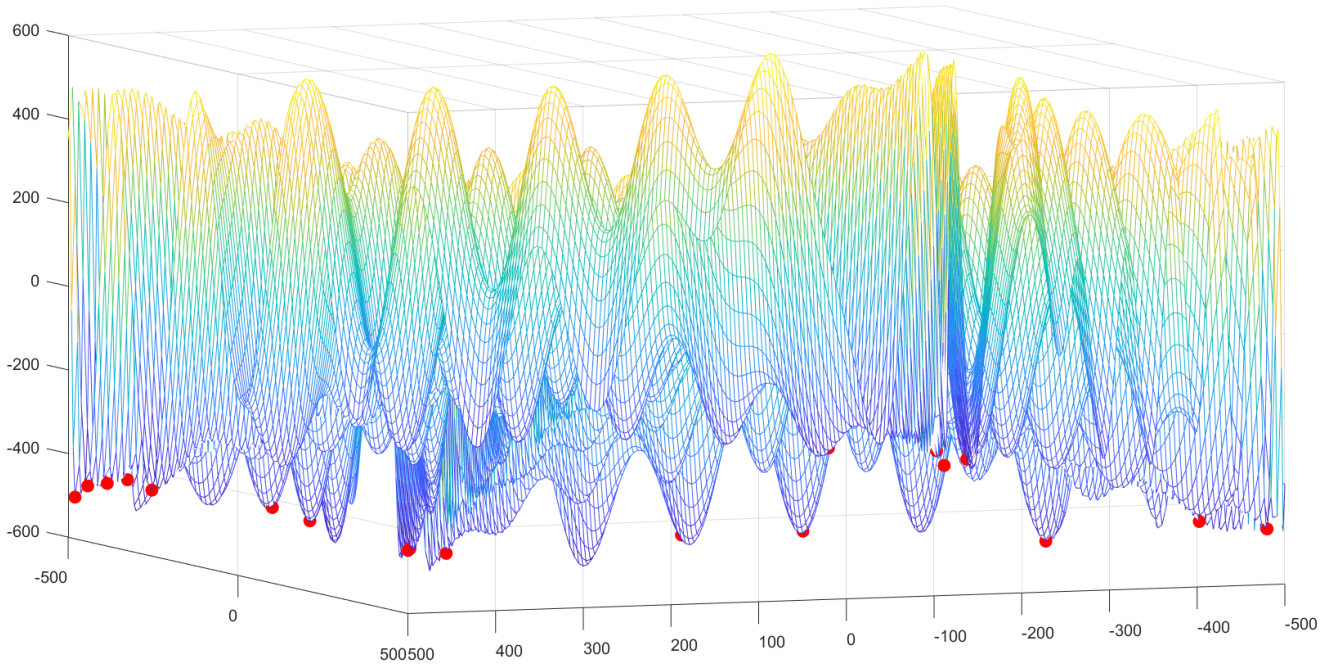

Figure 5: 20 best dissimilar solutions for a run of the ES algorithm using the baseline configuration, with $D_{min} = 20$ and $D_{sim} = 1$ (see [Par20]).

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$\sigma_0^2 = 10^{-6}$	-1864.5	52.3	160.4
$\sigma_0^2 = 0.01$	-1895.2	59.7	159.6
$\sigma_0^2 = 1$	-1864.7	53.6	163.0
$\sigma_0^2 = 10$	-1667.6	102.1	165.9
$\sigma_0^2 = 100$	-1179.5	117.9	1233.1

Figure 9: Results for varying initial variance σ_0^2.

As seen in Figure 9, surprisingly small initial variances are rewarded by our algorithm. Our suspicion is that it is due to the constraints strategy chosen, Redraw. The Reject strategy has the negative effect of decreasing the density of solutions near the boundaries, which is particularly ill-suited to our problem, as solutions close to the boundaries are more likely to have unfeasible mutated offsprings (they "mutate out of the domain"), and thus disappear from the genetic pool.

Conversely, the Redraw strategy does preserve solution density near the boundary, but has the drawback of not punishing solutions whose associated covariance matrix has overly large eigenvalues, as they are sure to enjoy the same degree of reproductive success as if their children had no tendency to become unfeasible. The first effect has more impact than the second in the case of our particular problem[10], which makes the Redraw strategy more successful, but as a consequence, we suspect that the strategy parameters become essentially meaningless after a few generations: all the selection pressure is focused on the control variables, to which almost entirely random mutations are applied. Small initial variances have the effect of delaying the generation by which strategy parameters have become completely aberrant.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$k_{vet} = 1$	-1904.9	38.2	116.6
$k_{vet} = 2$	-1904.0	50.1	116.9
$k_{vet} = 5$	-1914.4	43.3	134.3
$k_{vet} = 20$	-1909.3	47.8	120.7
$k_{vet} = 45$	-1908.2	46.7	121.0
$k_{vet} = 90$	-1908.0	45.8	121.2

Figure 10: Results for varying number of veterans k_{vet}. The case $k_{vet} = 90$ corresponds to the $(\mu + \lambda)$ strategy.

[10] We expect the situation to be very different for unconstrained problems, for example.

While allowing too many veterans to survive can slightly slow down the adaptation rate of the population, we find in Figure 10 that keeping a small number of them increases performance (somewhat contrary to what is said to usually be the case in [Par20]), as the best solutions found so far are sure to survive. This might be due to the "aberrant mutations" phenomenon described above, which increases the risk for good solutions to be lost.

We have highlighted in red the mean performance for $k_{vet} = 5$, which is the best we have achieved for any choice of algorithm or configuration, with relatively low standard deviation and running time too.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
P, CVRO=I, SPRO=I	-505.0	131.6	115.5
G, CVRO=D, SPRO=I	-1756.7	91.4	177.7
G, CVRO=I, SPRO=I	-166.8	50.7	176.1

Figure 11: Results for different choices of recombination operators: "P" stands for "pairwise", "G" for "global", "D" for "discrete", "I" for "intermediate", "CVRO" for "control variables recombination operator", and "SPRO" for "strategy parameters recombination operator".

As seen in Figure 11, global recombination slightly hurts the performance, as it keeps distinct local optima to be simultaneously explored. Much worse is intermediate recombination, be it local or global. This might be essentially due to the nature of Problem 1.1, whose optima are close to the boundaries. Indeed, intermediate recombination is an averaging process, which tends to produce solutions close to the center of the domain. Conversely, the offspring of two or more solutions that are close to the boundary (and potentially very far apart) has reasonably high chances of also being close to the boundary, which is a key property here.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Reject	-1730.2	265.1	138.5

Figure 12: Results for the Reject constraints strategy, where an offspring that has been made unfeasible by its mutation is "killed off".

We have discussed above (in relation to Figure 9) the compared advantages and disadvantages of both constraints strategies. As stated before and illustrated in Figure 12, Reject performs worse on average, with much increased variance.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Student	-1882.1	48.6	170.4

Figure 13: Results for the modified mutation operator using Student's t-distribution.

Though it does not seem to particularly hurt performance, we see in Figure 13 that the modified mutation operator that uses Student's t-distribution offers no improvement either (and increases running time). This might again be due to the erratic nature of the strategy parameters that has been discussed above. Further parameter tuning might lead us to a more definitive conclusion.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Grad., $c_{grad} = 0.01$	-1524.5	83.6	65.9
Grad., $c_{grad} = 0.1$	-1444.9	110.4	69.0
Grad., $c_{grad} = 0.5$	-1328.3	121.4	73.3
Grad., $c_{grad} = 1$	-1307.7	122.1	101.2
Grad., $c_{grad} = 5$	-1283.0	126.3	196.6

Figure 14: Results using the gradient-enriched mutation operator (remember that each evaluation of Rana's function's gradient is considered to be equivalent to $n = 5$ evaluations of the function itself) with different coefficients c_{grad}.

Though the gradient-enriched mutation operator might slightly improve local search, the increased cost in terms of evaluations, and hence the decreased number of generations allowed, more than compensate for that and lead to a net decrease in performance, as illustrated in Figure 14. Global search capacities might also be slightly impacted by this somewhat greedy search method. When running the gradient-enriched ES for the same time (115 seconds) as the baseline configuration, rather than for the same number of evaluations, the comparison is not as unfavourable (mean performance of about -1750), but remains on the side of the baseline configuration.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
End., $c_{endogamy} = 10$	-1884.3	54.7	127.3
End., $c_{endogamy} = 30$	-1882.9	51.8	128.6
End., $c_{endogamy} = 100$	-1889.5	58.6	129.5
End., $c_{endogamy} = 300$	-1885.9	58.3	129.6

Figure 15: Results using the "endogamy" recombination operator, with different values for $c_{endogamy}$.

The endogamy recombination operator seems to have no significant impact (potentially slightly negative, though more tests would be required to reach definitive conclusions) on the performance, based on Figure 15. It might be because the population quickly becomes concentrated in a rather small area, at which points it has no effect anymore.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Islands, $N_{islands} = 3$	-1880.9	57.4	117.7
Islands, $N_{islands} = 6$	-1859.2	55.2	118.3
Islands, $N_{islands} = 12$	-1807.9	67.9	120.0
Islands, $N_{islands} = 24$	-1741.2	83.0	122.0
Islands, $N_{islands} = 45$	-1655.9	90.9	138.6

Figure 16: Results using the islands model with various numbers of islands and fixed migration rate $c_{mig} = 0.2$ and migration frequency $T_{mig} = 1$.

As seen in Figures 16 and 17, the islands model does not seem to improve performance, even hurting it when the number of islands becomes too large, surely because small pockets of bad solutions are allowed to survive in islands, slowing down the convergence rate. Better choices of parameters (we could not test enough combinations of $(N_{islands}, c_{mig}, T_{mig})$ and their interactions with other parameters, especially λ and μ), and particularly of connectivity graph between islands, might improve performance, in line with what is suggested in [GCZ+15] - our model might have been too naive.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Islands, $c_{mig} = 0.05$	-1862.2	58.6	115.6
Islands, $c_{mig} = 0.1$	-1878.0	51.6	116.7
Islands, $c_{mig} = 0.4$	-1880.7	51.6	148.6
Islands, $T_{mig} = 2$	-1873.8	58.1	115.4
Islands, $T_{mig} = 4$	-1868.0	55.0	115.3
Islands, $T_{mig} = 8$	-1862.7	55.7	116.0

Figure 17: Results using the islands model with $N_{islands} = 3$ islands, different migration rates c_{mig} and fixed migration frequency $T_{mig} = 1$, as well as fixed migration rate $c_{mig} = 0.2$ and various migration frequencies T_{mig}.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
CMA, $\lambda = 10$	-1483.7	204.0	114.4
CMA, $\lambda = 30$	-1540.3	148.3	49.4
CMA, $\lambda = 100$	-1584.4	131.7	39.3
CMA, $\lambda = 220$	-1621.8	131.9	37.7
CMA, $\lambda = 450$	-1570.7	127.4	37.0
CMA, $\lambda = 900$	-1496.7	121.9	37.0

Figure 18: Results for the CMA-ES algorithm, using an initial variance $\sigma^2 = 50$ and varying number of offsprings λ. The number of parents is $\mu = \frac{\lambda}{2}$, and the other parameters are as given in [HK04].

As seen in Figure 18, CMA-ES performed surprisingly poorly, while it is supposed to be among the state-of-the-art ES algorithms. This is especially surprising considering that it has been reported to perform very well on Rastrigin's function (see [HK04]), which shares many similarities with Rana's function. Moreover, the best performance was recorded for $\lambda = 220$, which is significantly larger than the value of $\lambda = 10$ recommended in dimension $n = 5$ in [HK04]. One reason for that might be that while CMA-ES proceeds locally in a much smarter fashion than standard ES, as it adapts the covariance matrix to the past successful moves and thus captures the local shape of the graph of the objective function, it might not offer any significant improvement in terms of global search capacities.

Furthermore, we suspect that the constraints might have played an important role here: as good solutions are found near the boundary, but no point is allowed outside of D, the distribution

of points is artificially modified in the neighborhood of good solutions. There will be points near the local minima being explored on one side of the boundary, but not on the other, which negatively impacts the next generation's mean and covariance matrix and misleads the algorithm regarding the local situation.

5.2 Tabu Search results

Below are the results for the baseline configuration of the TS algorithm.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Baseline	-1752.0	111.4	69.7

Figure 19: Results for the baseline TS configuration

As seen in Figures 20 and 21, the algorithm works as intended in dimension 2. After a rather erratic first phase (due to the rather high initial stepsize), the search is intensified around promising areas.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$\delta_0 = 1$	-1285.6	253.5	62.2
$\delta_0 = 10$	-1492.8	177.1	78.5
$\delta_0 = 50$	-1645.3	123.4	65.7
$\delta_0 = 100$	-1681.3	100.4	65.6
$\delta_0 = 300$	-1783.7	97.1	73.0
$\delta_0 = 500$	-1775.8	93.4	78.2

Figure 22: Results for various values of the initial stepsize δ_0.

In Figure 22, we see the importance of the initial stepsize. Essentially, a very large initial stepsize allows the algorithm to jump around for a few hundred moves, as seen in Figure 20, which improves global search, before proceeding in a more conservative manner and converging to the most promising solutions.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$\alpha = 0.2$	-1543.1	192.6	45.7
$\alpha = 0.4$	-1570.0	163.3	60.3
$\alpha = 0.6$	-1633.1	159.2	64.4
$\alpha = 0.8$	-1708.1	135.0	66.1
$\alpha = 0.95$	-1734.9	94.3	75.7

Figure 23: Results for various values of the stepsize reduction coefficient α.

Reducing the stepsize too fast hurts exploration (and even exploitation for α very small, as the algorithm virtually stops moving after a few reduction steps), while not decreasing it fast enough slows down the convergence process.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$N_{STM} = 1$	-1618.0	123.2	61.0
$N_{STM} = 3$	-1722.7	112.4	68.7
$N_{STM} = 12$	-1786.4	99.2	70.6
$N_{STM} = 20$	-1771.1	105.1	70.3
$N_{STM} = 30$	-1776.7	112.7	70.3

Figure 24: Results for different values of the STM size N_{STM}.

As seen in Figures 24 and 25, short and medium term memory sizes do not have a particularly significative impact on performance, except for very small N_{STM}, at which point the algorithm essentially behaves like a Hooke and Jeeves search (see [Par20]), with the addition of the intensification, diversification and stepsize reduction steps.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$N_{MTM} = 1$	-1743.2	114.3	72.0
$N_{MTM} = 3$	-1737.4	115.3	69.5
$N_{MTM} = 8$	-1755.3	119.9	70.4
$N_{MTM} = 12$	-1757.0	112.5	69.7

Figure 25: Results for different values of the MTM size N_{STM}.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
$r = 1$	-1757.4	103.5	65.8
$r = 2$	-1743.3	120.6	70.7
$r = 5$	-1739.2	106.0	73.2
$r = 10$	-1748.1	107.6	169.8

Figure 26: Results for various values of the grid ratio r.

Grid ratio seems to have no real impact on performance, as seen in Figure 26.

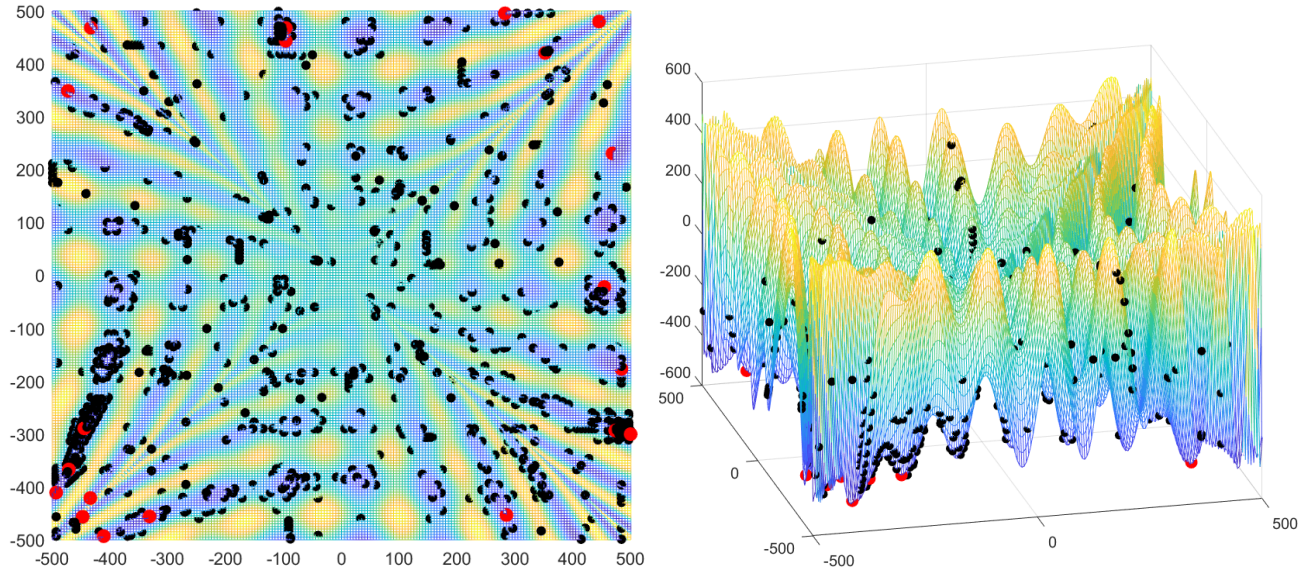

Figure 20: All visited solutions for a run of the TS algorithm using the baseline configuration, seen from below and in 3D respectively. The 20 best dissimilar solutions (with $D_{min} = 20$ and $D_{sim} = 1$) are in red.

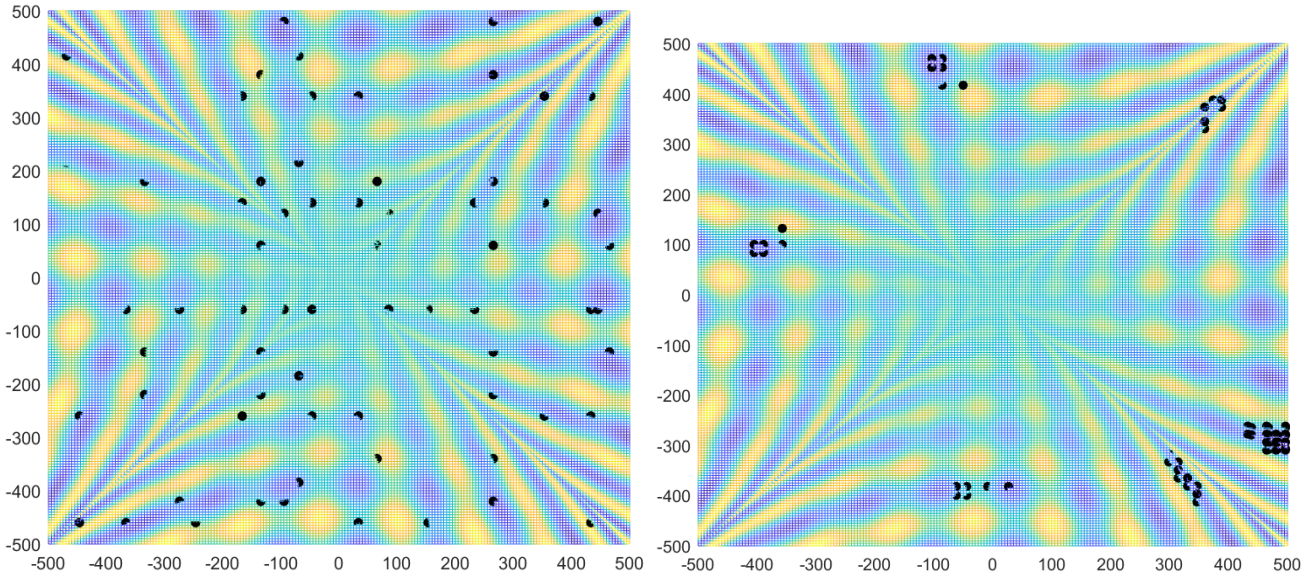

Figure 21: The first figure shows the first 100 moves of the TS algorithm (baseline configuration), while the second one shows moves 900 to 1000.

Conf. (T_I, T_D, T_R)	Mean $\tilde{p}$	S.d. ν	Time t
$(5, 10, 15)$	-1725.9	115.9	64.4
$(10, 15, 20)$	-1737.4	119.1	66.7
$(40, 60, 80)$	-1658.3	118.5	77.4
$(100, 150, 200)$	-1530.9	133.9	81.9
$(200, 300, 400)$	-1404.8	138.3	83.0

Figure 27: Results for different values of the intensification threshold T_I, the diversification threshold T_D and the stepsize reduction threshold T_R.

As seen in Figure 27, letting the thresholds get too high slows down TS convergence, as the algorithm spends too many iterations exploring unpromising areas and having too high a stepsize to efficiently converge.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Concentric	-1808.2	101.0	58.6

Figure 28: Results using the concentric TS (all other parameters being the same as in the baseline case).

The concentric TS does improve upon the standard TS, as it forces the algorithm to move away from previously visited local minima, instead of potentially circling around them.

6 Conclusion

We have compared the effect of various configurations for both algorithms. Overall, ES performs better, perhaps because it is better suited to the multimodal nature of Problem 1.1 due to its highly multimodal nature than TS. Note however that as displayed in Figure 30, TS performs much better for a very small number of allowed evaluations, but that its performance quickly stagnates.

Conf.	Mean $\tilde{p}$	S.d. ν	Time t
Wanderlust, $c_w = 0.1$	-1759.3	112.0	73.2
Wanderlust, $c_w = 0.3$	-1774.8	97.0	72.4
Wanderlust, $c_w = 1$	-1810.3	92.3	72.9
Wanderlust, $c_w = 2$	-1838.3	77.9	75.0
Wanderlust, $c_w = 5$	-1841.1	92.2	85.4
Wanderlust, $c_w = 10$	-1849.8	74.4	85.9
Wanderlust, $c_w = 20$	-1852.2	76.9	75.3
Wanderlust, $c_w = 50$	-1841.5	81.5	76.1
Wanderlust, $c_w = 100$	-1840.5	70.9	76.9

Figure 29: Results using the "wanderlust" option with different values of the parameter c_w.

Our own Wanderlust option performs surprisingly well, as visible in Figure 29, where the best mean performance achieved with a TS configuration is highlighted in red (-1852.2, with good standard deviation and running time). Though we do think that the intuition behind it is reasonable, we suspect that it is particularly successful here due, once again, to the nature of Problem 1.1. Indeed, the Wanderlust option has the effect of pushing the algorithm away from previously visited solutions - tests have revealed that this has the tendency to force it against the boundary, which it then has to explore thoroughly. As the best solutions are close to the boundary, this partially explains such high performance.

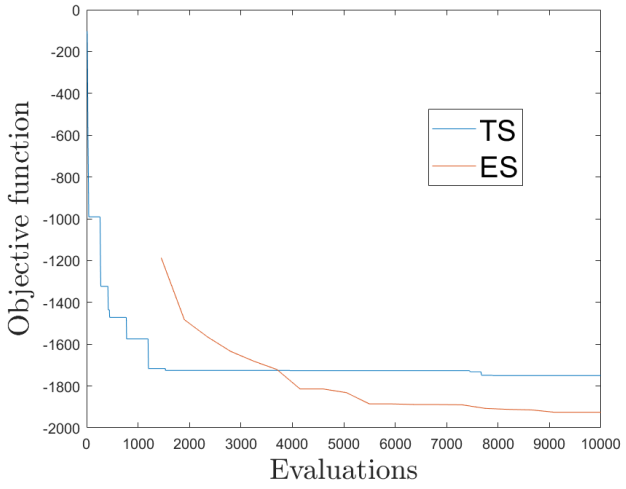

Figure 30: Best solution found as a function of the number of objective function evaluations for TS and ES.

Many surprising phenomena, such as the inefficiency of the CMA-ES algorithm, were observed, which seemed to have been caused by the nature of Problem 1.1, and more specifically the fact that it is bounded. Perhaps different implementation choices, for example using a penalty function to model the constraints, would have led to very different observations.

As far as possible improvements go, further and even more systematic testing is the first that comes to mind. For example, one could see the mean performance as a function of the hyperparameters, and try to apply optimization methods to the hyperparameters of our optimization algorithms (of course, this would mostly make sense in a research context).

A mixed algorithm, using for example ES to perform global search, then more traditional methods (Newton's method, etc.) for local search, might peform very well.

Finally, expert knowledge can always be included: in this particular case, using the fact that we knew that the optima were close to the boundaries.

References

[ADGV14] J.-M. Alliot, N. Durand, J.-B. Gotteland, and C. Vanaret. Certified global minima for a benchmark of difficult optimization problems, 2014. `https://arxiv.org/abs/2003.09867`.

[Bä96] T. Bäck. *Evolutionary algorithms in theory and practice*. Oxford University Press, 1996.

[Dre02] Z. Drezner. A new heuristic for the quadratic assignment problem. *Journal of Applied Mathematics and Decision Sciences*, 6, 2002.

[GCZ+15] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, and J.-J. Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34, 2015.

[HK04] N. Hansen and S. Kern. Evaluating the cma evolution strategy on multimodal test functions. *International Conference on Parallel Problem Solving from Nature*, 8, 2004.

[HO01] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2), 2001.

[JY13] M. Jamil and X.-S. Yang. A literature survey of benchmark functions for globaloptimization problems. *Int. Journal of Mathematical Modelling and Numerical Optimisation*, 4(2), 2013. `https://arxiv.org/abs/1308.4008`.

[Mat20] MathWorks. Matlab website. https://uk.mathworks.com, 2020.

[Par20] G.T. Parks. Lecture notes in practical optimisation, 2020.

[SK06] J. Schneider and S. Kirkpatrick. *Stochastic optimization*. Springer, 2006.

[Whi01] D. Whitley. An overview of evolutionary algorithms:practical issues and common pitfalls. *Information and Software Technology*, 43, 2001.

A General functions

```
% Computes Rana's function, expects a line vector x=[x1,...,xn]
% Simple application of the formula
function f=RanaFun(x)
    n=size(x,2);
    f=0;
    x_i=x(1:n-1);   % x_i in the formula
    x_ip1=x(2:n);   % x_i+1 in the formula
    v1=sqrt(abs(x_ip1+x_i+1));
    v2=sqrt(abs(x_ip1-x_i+1));
    % vectorized for elegance and clarity
    f=sum(x_i .* cos(v1) .* sin(v2) + (1+x_ip1) .* cos(v2) .* sin(v1));
end
% Computes Rana's function's gradient, expects a line vector x=[x1,...,xn]
function D=RanaFunGradient(x)
    n=size(x,2);
    D1=zeros(1,n);      % derivative of the ith term of the sum wrt to x_i
    D2=zeros(1,n);      % derivative of the ith term of the sum wrt to x_i+1
    parfor i=1:n-1
       s1=sign(x(i+1)+x(i)+1);
       s2=sign(-x(i+1)+x(i)-1);
       a=sqrt(abs(x(i+1)+x(i)+1));
       b=sqrt(abs(-x(i+1)+x(i)-1));
       D1(i)= cos(a)*sin(b)...
           +0.5*x(i)*((-1)*sin(a)*sin(b)*s1/a + cos(a)*cos(b)*s2/b) ...
           +0.5*(1+x(i+1))*((-1)*sin(b)*sin(a)*s2/b + cos(b)*cos(a)*s1/a);
       D2(i+1)=0.5*x(i)*(sin(a)*sin(b)*s1/a + cos(a)*cos(b)*(-1)*s2/b)...
           +cos(a)*sin(b)...
           +0.5*(1+x(i+1))*((-1)*sin(b)*sin(a)*(-1)*s2/b + cos(b)*cos(a)*s1/a);
    end
        D=D1+D2;
end
% Constraint of the problem
function f=constraintFunction(x)
    if sum(abs(x)>500)==0
        f=1;
    else
        f=-1;
    end
end
% tests whether a solution is feasible
function feas=feasible(x,Cf)
    feas=true;
    for i=1:size(Cf)
        if Cf{1,i}(x)<0
            feas= false;
        end
    end
end
```


Code 1

B Code TS

```
% The main function (parameters are described in the report)
function [X_history,Y_history,archive,graphdata]=Tabu(Cf,Of,dimension,range,gridRatio,...
    stepsize,stepsize_red_coeff,STM_size,tabu_direction,STM_direction_size,c_wanderlust,...
    MTM_size,intensify_thres,diversify_thres,reduce_thres,numMaxEvaluation,concentric,...
    tolerance,minOrMax,archive_size,D1,D2)
    if minOrMax=="min"
        Of = @(x) -Of(x);
    end
    evaluations=0;
    moves=0;
    counter=0;
    X_history=zeros(dimension,0);
    Y_history=zeros(1,0);
    archive={zeros(dimension,0),zeros(1,0)};
    areas=initialize_areas(dimension,gridRatio);

    graphdata=zeros(2,0);

    STM=NaN(dimension,0);
    MTM={zeros(1,0),zeros(dimension,0)};

    [x,areas]=GeneratePoint(range,dimension,gridRatio,Cf,areas);
    STM=[x];
    y=Of(x);
    evaluations=evaluations+1;
    center=x;
    [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
        range,dimension,gridRatio,MTM,MTM_size,moves,areas,...
        X_history,Y_history,archive,archive_size,D1,D2);
    % the main loop
    while evaluations<=numMaxEvaluation-(2*dimension+1) &  stepsize>=tolerance
        % ... the main loop is on the next page
    end
    if minOrMax=="min"
        Y_history=-Y_history;
        archive{2}=-archive{2};
    end
end
```

Code 2

```
while evaluations<=numMaxEvaluation-(2*dimension+1) &  stepsize>=tolerance%the main loop
    [x,y,STM,evaluations,trapped]=move(x,y,STM,tabu_direction,STM_size,STM_direction_size,...
    c_wanderlust,Of,Cf,stepsize,evaluations,concentric,center);
    [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
        range,dimension,gridRatio,MTM,MTM_size,moves,areas,...
        X_history,Y_history,archive,archive_size,D1,D2);
    if newOptimum
        counter=0;
        center=x;
    else
        counter=counter+1;
    end
    if trapped & concentric & counter<intensify_thres
        counter=intensify_thres;
    end
    if counter==intensify_thres & feasible(mean(MTM{2},2),Cf)
        x=mean(MTM{2},2);
        STM=[x];
        y=Of(x);
        center=x;
        evaluations=evaluations+1;
        [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
            range,dimension,gridRatio,MTM,MTM_size,moves,...
            areas,X_history,Y_history,archive,archive_size,D1,D2);
        if newOptimum
            counter=0;
        else
            counter=counter+1;
        end
    end
    if counter==diversify_thres
        [x,areas]=GeneratePoint(range,dimension,gridRatio,Cf,areas);
        STM=[x];
        y=Of(x);
        evaluations=evaluations+1;
        center=x;
        [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
            range,dimension,gridRatio,MTM,MTM_size,moves,areas,...
            X_history,Y_history,archive,archive_size,D1,D2);
    end
    if counter==reduce_thres
        stepsize=stepsize*stepsize_red_coeff;
        x=MTM{2}(:,end);
        y=MTM{1}(:,end);
        STM=[x];
        center=x;
        [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
            range,dimension,gridRatio,MTM,MTM_size,moves,areas,...
            X_history,Y_history,archive,archive_size,D1,D2);
        counter=0;
    end
    graphdata=[graphdata,[evaluations;archive{2}(1)]];
end
```

```
% simple function that performs routine updating tasks
function [moves,areas,X_history,Y_history,MTM,newOptimum,archive]=general_update(x,y,...
    range,dimension,gridRatio,MTM,MTM_size,moves,areas,X_history,...
    Y_history,archive,archive_size,D1,D2)
    areas=update_areas(x,areas,range,dimension,gridRatio);
    moves=moves+1;
    X_history(:,moves)=x;
    Y_history(:,moves)=y;
    [MTM,newOptimum]=updateMTM(x,y,MTM,MTM_size);
    archive=update_archive(archive,x,y,archive_size,D1,D2);
end
% initialize the LTM
function areas=initialize_areas(dimension,gridRatio)
    numberMoves=zeros(1,gridRatio^dimension);
    indices=zeros(dimension,gridRatio^dimension);
    if gridRatio~=1
        for i=1:gridRatio^dimension
            indices(:,i)=de2bi(i-1,dimension,gridRatio);
        end
    else
       indices=0;
    end
    areas={numberMoves,indices};
end
% Generates a point based on the current LTM, and updates it
function [x,areas]=GeneratePoint(range,dimension,gridRatio,Cf,areas)
    if sum(areas{1}==0)==0
        areas{1}=zeros(1,gridRatio^dimension);
    end
    candidate_areas=areas{1}(areas{1}==0);
    candidate_areas_indices=areas{2}(:,areas{1}==0);
    acceptable=false;
    while not(acceptable)
        i=randi(numel(candidate_areas));
        index=candidate_areas_indices(:,i);
        x=(2*range/gridRatio)*(rand(dimension,1)+index)-range;
        acceptable= feasible(x,Cf);
    end
end
```

Code 4

```
% the main moving function, which follows the description in the notes
function [x,y,STM,evaluations,trapped]=move(x,y,STM,tabu_direction,...
STM_size,STM_direction_size,c_wanderlust,Of,Cf,stepsize,evaluations,concentric,center)
    d=size(x,1);
    neighb_x=repmat(x,1,2*d)+stepsize*[eye(d),-eye(d)];
    neighb_y=zeros(1,2*d);
    tabu=true(1,2*d);
    parfor i=1:2*d
        if not(concentric)
            tabu(1,i)=not(feasible(neighb_x(:,i),Cf)) | ( ismember(neighb_x(:,i)',...
            STM(:,1:min(size(STM,2),STM_size))','rows') ) ;
        else
            tabu(1,i)=not(feasible(neighb_x(:,i),Cf))| ...
            norm(center-neighb_x(:,i))<norm(center-x)
        end
        if not(tabu(1,i))
            neighb_y(1,i)=Of(neighb_x(:,i)');
            evaluations=evaluations+1;
        end
    end
    if sum(not(tabu))==0        % This is a rare occurence for concentric==false
        "Point is trapped"
        STM=[x];
        trapped=true;
        return
    else
        trapped=false;
    end
    feasible_x=neighb_x(:,not(tabu));
    feasible_y=neighb_y(1,not(tabu));
    %[sorted_y,I]=sort(feasible_y,'descend');
    [sorted_y,I]=selectBestMove(feasible_y,feasible_x,x,STM,tabu_direction,...
    STM_direction_size,c_wanderlust);
    x_pattern_move=2*feasible_x(:,I(1))-x;
    y=sorted_y(1);
    x=feasible_x(:,I(1));
    STM=update_STM(x,STM,max(STM_size,STM_direction_size));
    if feasible(x_pattern_move,Cf)
        y_pattern_move=Of(x_pattern_move');
        evaluations=evaluations+1;
        if y_pattern_move>y
            y=y_pattern_move;
            x=x_pattern_move;
            STM=update_STM(x,STM,max(STM_size,STM_direction_size));
        end
    end
end
```

Code 5

```
% updates the LTM
function areas=update_areas(x,areas,range,dimension,gridRatio)
    index=floor((x+range)/(2*range/gridRatio));
    if gridRatio~=1; i=bi2de(index',dimension,gridRatio)+1;
    else i=1; end
    areas{1}(i)=areas{1}(i)+1;
end
% updates the STM
function STM=update_STM(x,STM,n)
    if size(STM,2)<n      STM=[x,STM];
    else
        STM=circshift(STM,1,2);
        STM(:,1)=x;
    end
end
% selects the best move (possibly with the Wanderlust option)
function      [sorted_y,I]=selectBestMove(feasible_y,feasible_x,...
x,STM,tabu_direction,STM_direction_size,c_wanderlust)
    if tabu_direction==false | size(STM,2)<=1
        [sorted_y,I]=sort(feasible_y,'descend');
    else
        past_direction=-(x-mean(STM(:,1:min(end,STM_direction_size)),2))/...
        norm(x-mean(STM(:,1:min(STM_direction_size,end)),2));
        n=size(feasible_x,2);
        directional_aversion=zeros(1,n);
        step_size=norm(feasible_x(:,1)-x);
        for i=1:n
            directional_aversion(1,i)=-((feasible_x(:,i)-x)/step_size)'*past_direction;
        end
        modified_y=feasible_y+std(feasible_y,1)  *c_wanderlust* directional_aversion;
        [sorted_modified_y,I]=sort(modified_y,'descend');
        sorted_y=feasible_y(I);
    end
end
% MTM constantly sorted in ascending order
function [MTM,newOptimum]=updateMTM(x,y,MTM,MTM_size)
    if size(MTM{1},2)==0 | y>MTM{1}(end)
        newOptimum=true;
    else  newOptimum=false; end
    if size(MTM{1},2)<MTM_size
        [Y_MTM,I]=sort([y,MTM{1}]);
        temp_X=[x,MTM{2}];
        X_MTM=temp_X(:,I);
        MTM={Y_MTM,X_MTM};
    elseif y>MTM{1}(1,1)
        [Y_MTM,I]=sort([y,MTM{1}(:,2:end)]);
        temp_X=[x,MTM{2}(:,2:end)];
        X_MTM=temp_X(:,I);
        MTM={Y_MTM,X_MTM};
    end
end
```


Code 6

C Code ES

```
function [X_parents_history,Y_parents_history,archive,graphdata]=...
ES(Cf,Of,my_gradient,dimension,range,scale,N,numMaxEvaluation,...
m1,m2,m3,mu,lambda,tolerance,IDCoeff,StabilizationCoeff,perturbation,...
    veteransNumber,CombinationContVarOp,CombinationStratParOp,...
    CombPairGlob,SelectionStrat,ConstraintStrat,ConvCriterion,endogamy,endogamy_scale,...
    islandsNumber,migrationRate,migrationFrequency,minOrMax,archive_size,D1,D2)
    if minOrMax=="min"
        Of = @(x) -Of(x);
    end
    graphdata=zeros(2,0);
    count=0;
    generation=0;
    X_parents_history=cell(1,0);
    Y_parents_history=cell(1,0);
    archive={zeros(dimension,0),zeros(1,0)};
    X=GeneratePopulation(N*islandsNumber,range,dimension,Cf);
    [Y,count] = evaluate(X,Of,count);
    Theta=GenerateStrategyParameters(N*islandsNumber,dimension,scale);
    if IDCoeff ==0
        margin=lambda*islandsNumber;
    else
        margin=lambda*islandsNumber*(1+dimension);
    end
    while count<=numMaxEvaluation-margin &  TestConvergence(Y,tolerance,ConvCriterion) == false
        % ... the main loop, in the next page
    end
    if minOrMax=="min"
        for i=1:size(Y_parents_history,2)
            Y_parents_history{1,i}=-Y_parents_history{1,i};
        end
        archive{2}=-archive{2};
    end
end
```

Code 7

```
% the main loop
while count<=numMaxEvaluation-margin &  TestConvergence(Y,tolerance,ConvCriterion) == false
        [selectedIndices,failure]=ESSelect(Y,mu,SelectionStrat,islandsNumber);
        if failure
            "Population had died out"
            return
        end
        X_parents_history{1,generation+1}=X(:,selectedIndices);
        Y_parents_history{1,generation+1}=Y(:,selectedIndices);
        archive=update_archive(archive,X(:,selectedIndices),...
        Y(:,selectedIndices),archive_size,D1,D2);
        if mod(generation,migrationFrequency)==0
            [X_migrated,Theta_migrated]=Migrate(X(:,selectedIndices),...
            Theta(1,selectedIndices),migrationRate);
            [X_children,Theta_children]=Reproduce(X_migrated,...
            Theta_migrated,lambda,CombinationContVarOp,...
            CombinationStratParOp,CombPairGlob,endogamy,endogamy_scale,islandsNumber);
        else
            [X_children,Theta_children]=Reproduce(X(:,selectedIndices),...
            Theta(1,selectedIndices),lambda,CombinationContVarOp,,,,
            CombinationStratParOp,CombPairGlob,endogamy,endogamy_scale,islandsNumber);
        end
        [X_children,Theta_children,count]=Mutate(X_children,Theta_children,,,,
        count,m1,m2,m3,IDCoeff,perturbation,my_gradient,ConstraintStrat,Cf,StabilizationCoeff);
        [Y_children,count] = evaluate(X_children,Of,count);
        if count<=numMaxEvaluation   % to be commented
            [veteransIndices,failure]=ESSelect(Y,veteransNumber,SelectionStrat,islandsNumber);
            [X,Y,Theta]=newGeneration(X,Y,Theta,X_children,Y_children,...
            Theta_children,veteransIndices,veteransNumber,islandsNumber);
            generation=generation+1;
        end

        graphdata=[graphdata,[count;-archive{2}(1)]];
end
```

Code 8

```
% generates initial population
function X=GeneratePopulation(N,range,dimension,Cf)
    X=zeros(dimension,N);
    n=0;
    while n<N
        x=2*range*rand(dimension,1)-range;
        if feasible(x,Cf)    % Redundant with the previous line in our special case
            X(:,n+1)=x;
            n=n+1;
        end
    end
end
% generates initial SP
function Theta=GenerateStrategyParameters(N,dimension,scale)
    Theta=cell(1,N);
    Theta(:)={eye(dimension)*scale};
end
% selects the best solutions
function [selectedIndices,failure]=ESSelect(Y,mu,SelectionStrat,islandsNumber)
    n=size(Y,2);
    k=n/islandsNumber;
    if n==0
        failure=true;
    else
        failure=false;
    end
    selectedIndices=logical(zeros(1,n));
    for j=1:islandsNumber
        if SelectionStrat=="mufirst"
            [B,I]=sort(Y(1+(j-1)*k:j*k),'descend');
            %selectedIndices=I(1:min(mu,n));     % better, automatically ordered
            selectedIndices((j-1)*k+I(1:min(mu,k)))=true;
        else
            "Choose selection strategy"
            return
        end
    end
end
```

Code 9

```
% migrates solutions between islands
function [X_migrated,Theta_migrated]=Migrate(X,Theta,migrationRate)
    X_migrated=X;
    Theta_migrated=Theta;
    n=size(X,2);
    k=floor(migrationRate*n);    % the number of migrators
    my_perm=randperm(n);
    migrators_indices=my_perm(1:k);     % randomly pick the indices
                                        % of the solutions that will migrate
    X_migrators=X(migrators_indices);   % the migrators
    Theta_migrators=Theta(migrators_indices);   % the migrators's parameters
    my_perm_2=randperm(k);
    % randomly migrating the migrators
    X_migrated(migrators_indices)=X_migrators(my_perm_2);
    Theta_migrated(migrators_indices)=Theta_migrators(my_perm_2);
end
% Recombination operator
function [X_children,Theta_children]=Reproduce(X_parents,Theta_parents,...
lambda,CombinationContVarOp,CombinationStratParOp,CombPairGlob,endogamy,endogamy_scale,islandsNum
   k=size(X_parents,2);
   l=k/islandsNumber;
   X_children=zeros(size(X_parents,1),0);
   Theta_children=cell(1,0);
       for t=1:islandsNumber      % reproduction only inside islands (migration occurs before)
           temp_X=zeros(size(X_parents,1),lambda);% necessary to avoid indexing issues within pa
           temp_Theta=cell(1,lambda);
           parfor i=1:lambda
                if k==1
                    X_children(:,i)=X_parents(:,1);
                else
                        if CombPairGlob=="pair"
                            [a,b]=SelectParents(X_parents(:,1+(t-1)*l:t*l),...
                            endogamy,endogamy_scale);
                            temp_X(:,i)= CombControlVariables([X_parents(:,a+(t-1)*l),...
                            X_parents(:,b+(t-1)*l)],CombinationContVarOp);
                            temp_Theta{1,i}=CombStratPar({Theta_parents{1,a+(t-1)*l},...
                            Theta_parents{1,b+(t-1)*l}},CombinationStratParOp);
                        elseif CombPairGlob=="global"
                            temp_X(:,i)= CombControlVariables(X_parents(:,...
                            1+(t-1)*l:t*l),CombinationContVarOp);
                            temp_Theta{1,i}=CombStratPar(Theta_parents(1,...
                            1+(t-1)*l:t*l),CombinationStratParOp);
                        end
                end
           end
           X_children=[X_children,temp_X];
           Theta_children=[Theta_children,temp_Theta];
       end
end
```

```
% the main mutation operator
function [X_mutated,Theta_mutated,count]=Mutate(X,Theta,count,m1,m2,m3,...
IDCoeff,perturbation,my_gradient,ConstraintStrat,Cf,StabilizationCoeff)
    n=size(X,2);
    d=size(X,1);
    feasible_indices=true(1,n);
    X_mutated=zeros(d,n);
    Theta_mutated=cell(1,n);
    parfor i=1:n
            redraw=true;
            redrawing_count=0;
            while redraw==true
                if IDCoeff==0
                    if perturbation=="normal"
                        X_mutated(:,i)=X(:,i)+Theta{1,i}*normrnd(0,1,d,1);
                    elseif perturbation=="student"
                        X_mutated(:,i)=X(:,i)+Theta{1,i}*trnd(1,d,1);
                    end
                else
                    if perturbation=="normal"
                        X_mutated(:,i)=X(:,i)+Theta{1,i}*normrnd(0,1,d,1)+...
                        IDCoeff*my_gradient(X(:,i)')'*exprnd(1);
                    elseif perturbation=="student"
                        X_mutated(:,i)=X(:,i)+Theta{1,i}*trnd(1,d,1)+...
                        IDCoeff*my_gradient(X(:,i)')'*exprnd(1);
                    end
                    count=count+d;
                end
                redraw = not(feasible(X_mutated(:,i),Cf));
                redrawing_count=redrawing_count+1;
                if ConstraintStrat=="accept"
                    redraw=false;
                    feasible_indices(1,i)=feasible(X_mutated(:,i),Cf);
                end
                if redrawing_count >2^d*10
                    Theta{1,i}=Theta{1,i}*eye(d)*0.1;
                    redrawing_count=0;
                    "covariance rescaled"
                end
            end
            Theta_mutated{1,i}=rotationPerturbation(d,m3)*Theta{1,i}*...
            variancePerturbation(d,m1,m2,StabilizationCoeff);
    end
    X_mutated=X_mutated(:,feasible_indices);
end
```

Code 11

```
% selects parents, potentially with the endogamy option
function [a,b]=SelectParents(X_parents,endogamy,endogamy_scale)
    k=size(X_parents,2);
    my_perm=randperm(k);
    a=my_perm(1);
    if endogamy==false       b=my_perm(2);
    else
        probas=zeros(1,k);
        for t=1:k
            if t~=a
                probas(1,t)=min(0.5,1/...
                (1+ min(9, norm(X_parents(:,a)-X_parents(:,t))/endogamy_scale)));
            end
        end
        b=genBern(probas);
    end
end
% generates random rotation matrices
function R=rotationPerturbation(d,m3)
    R=eye(d);
    for i=1:d
        for j=i+1:d
            xji=normrnd(0,1)*m3;
            R(i,:)=cos(xji)*R(i,:)-sin(xji)*R(j,:);
            R(j,:)=sin(xji)*R(i,:)+cos(xji)*R(j,:);
        end
    end
end
% generates random diagonal matrices
function D=variancePerturbation(d,m1,m2,StabilizationCoeff)
    D=zeros(d);
    x0=normrnd(0,1);
    for i=1:d
        xl=normrnd(0,1);
        D(i,i)=exp(m1*x0+m2*xl+StabilizationCoeff);
    end
end
```

Code 12

```
% evaluates a solution
function [Y,count] = evaluate(X,Of,count)
    n=size(X,2);
    Y=zeros(1,n);
    parfor i=1:size(X,2)
        Y(i)=Of(X(:,i)');
    end
    count = count+n;
end
% combines the control variables
function x= CombControlVariables(X,CombOp)
    if CombOp=="Intermediate"
        x=mean(X,2);
    elseif CombOp=="Discrete"
        [d,k]=size(X);
        x=zeros(d,1);
        for i=1:d
            j=genBern(ones(1,k));
            x(i)=X(i,j);
        end
    end
end
% combines the strategy parameters (intermediate only)
function Theta=CombStratPar(THETA,CombOp)
d=size(THETA{1,1},1);
Theta=zeros(d);
k=size(THETA,2);
if CombOp=="Intermediate"
    Theta=zeros(d);
    k=size(THETA,2);
    for t=1:k
        Theta=Theta+THETA{1,t}*THETA{1,t}'/k;
    end
    Theta=chol(Theta)';
end
end
% convergence test
function hasConverged=TestConvergence(Y,tolerance,ConvCriterion)
    hasConverged=false;
    if ConvCriterion=="AbsDiff"
        if abs(max(Y)-min(Y))<tolerance
            hasConverged=true;
            "ES has converged"
        end
    else ConvCriterion=="RelDiff"
        if abs(max(Y)-min(Y))<tolerance*abs(mean(Y))
            hasConverged=true;
            "ES has converged"
        end
    end
end
```


Code 13

```
% mixes the veterans with the offsprings
function [X,Y,Theta]=newGeneration(X,Y,Theta,X_children,Y_children,Theta_children,...
veteransIndices,veteransNumber,islandsNumber)
       X_veterans=X(:,veteransIndices);
       Y_veterans=Y(1,veteransIndices);
       lambda=size(X_children,2)/islandsNumber;
       Theta_veterans=Theta(1,veteransIndices);
       X=zeros(size(X_children,1),islandsNumber*(lambda+veteransNumber));
       Y=zeros(1,islandsNumber*(lambda+veteransNumber));
       Theta=cell(1,islandsNumber*(lambda+veteransNumber));
       for i=1:islandsNumber
            X(:,1+(i-1)*(lambda+veteransNumber):i*(lambda+veteransNumber))=...
            [X_veterans(:,1+(i-1)*veteransNumber:i*veteransNumber),...
            X_children(:,1+(i-1)*lambda:i*lambda)];
            Y(1,1+(i-1)*(lambda+veteransNumber):i*(lambda+veteransNumber))=...
            [Y_veterans(1,1+(i-1)*veteransNumber:i*veteransNumber),...
            Y_children(1,1+(i-1)*lambda:i*lambda)];
            Theta(1,1+(i-1)*(lambda+veteransNumber):i*(lambda+veteransNumber))=...
            [Theta_veterans(1,1+(i-1)*veteransNumber:i*veteransNumber),...
            Theta_children(1,1+(i-1)*lambda:i*lambda)];
       end
end
```

Code 14