Introduction to *SSm*

Process Engineering group
Instituto de Investigaciones Marinas (C.S.I.C.)
c/Eduardo Cabello, 6
36208, Vigo (Spain)
gingproc@iim.csic.es

January 26, 2009

Contents

1 Scatter search **1**

1.1 Scatter Search methodology 1

1.2 Scatter Search tutorial 4

1.2.1 Initialization 4

1.2.2 First *RefSet* formation 4

1.2.3 Subset generation and combination 5

1.2.4 *RefSet* update 5

1.2.5 *RefSet* regeneration 5

1.2.6 Improvement method 7

2 *SSm*: A scatter search heuristic for bioprocess optimization **7**

2.1 Methodology 9

2.1.1 *Diversification Generation Method* 9

2.1.2 Building the *RefSet* 11

2.1.3 *Subset Generation* and *Solution Combination* methods 12

2.1.4 Updating the *RefSet* 13

2.1.5 *Improvement Method* 15

2.1.6 *RefSet* Rebuilding 19

2.1.7 Intensification 21

2.1.8 The *go beyond* strategy 21

2.1.9 Constraints handling 22

2.1.10 Integer variables handling 23

2.1.11 Stopping criterion 24

2.2 Application to benchmark problems 24

2.2.1 Unconstrained problems 24

2.2.2 Constrained problems 26

2.2.3 Mixed-integer problems 27

References **30**

1 Scatter search

Scatter search is a population-based metaheuristic that has recently been shown to yield promising outcomes for solving combinatorial and nonlinear optimization problems. Based on formulations originally proposed in the 1960s for combining decision rules and problem constraints such as the surrogate constraint method, scatter search uses strategies for combining solution vectors that have proved effective in a variety of problem settings.

1.1 Scatter Search methodology

Scatter search was first introduced by Fred Glover [17] as a heuristic for integer programming. Scatter search orients its explorations systematically, relative to a set of reference points that typically consist of good solutions obtained by prior problem solving efforts. The scatter search template [19] has served as the main reference for most of the scatter search implementations to date. Scatter search methodology is very flexible, since each of its elements can be implemented in a variety of ways and degrees of sophistication. Here we give a basic design to implement scatter search based on the well-known "five-method template" [32]. The advanced features of scatter search are related to the way these five methods are implemented. That is, the sophistication comes from the implementation of the scatter search methods instead of the decision to include or exclude certain elements (as in the case of tabu search or other metaheuristics).

The fact that the mechanisms within scatter search are not restricted to a single uniform design allows the exploration of strategic possibilities that may prove effective in a particular implementation. These observations and principles lead to the following "five-method template" for implementing scatter search:

1. A *Diversification Generation Method* to generate a collection of diverse trial solutions.
2. An *Improvement Method* to transform a trial solution into one or more enhanced trial solutions. Neither the input nor the output solutions are required to be feasible, though the output solutions will more usually be expected to be so. If no improvement of the input trial solution results, the "enhanced" solution is considered to be the same as the input solution.
3. A *Reference Set Update Method* to build and maintain a reference set consisting of the b "best" solutions found, where the value of b is typically small compared to the population size of other evolutionary algorithms, organized to provide efficient accessing by other parts of the method. Solutions gain membership to the reference set according to their quality or their diversity.
4. A *Subset Generation Method* to operate on the reference set, to produce several subsets of its solutions as a basis for creating combined solutions.
5. A *Solution Combination Method* to transform a given subset of solutions produced by the *Subset Generation Method* into one or more combined solution vectors.

Figure 1 shows the interaction among these five methods and highlights the central role of the reference set (*RefSet*). This basic design starts with the creation of an initial set of solutions P, and then extracts from it the *RefSet*. The darker circles represent improved solutions resulting from the application of the *Improvement Method*.

The *Diversification Generation Method* is used to build a large set P of diverse solutions. The size of P ($PSize$) is typically at least ten times the problem size. The initial *RefSet* is built according to the *Reference Set Update Method*, which can take the b best solutions (as regards their quality in the problem solving) from P to compose the *RefSet*. However, diversity can be considered instead of or in addition to quality for the updating. For example, the *Reference Set Update Method* could consist of selecting b distinct and maximally diverse solutions from P. Regardless

of the rules used to select the reference solutions, the solutions in *RefSet* are ordered according to quality, where the best solution is the first one in the list. The search is then initiated by applying the *Subset Generation Method* which, in its simplest form, involves generating all pairs of reference solutions. The pairs of solutions in *RefSet* are selected one at a time and the *Solution Combination Method* is applied to generate one or more trial solutions. These trial solutions are subjected to the *Improvement Method*. The *Reference Set Update Method* is applied once again to build the new *RefSet* with the best solutions, according to the objective function value, from the current *RefSet* and the set of trial solutions. The basic procedure terminates after all the generated subsets are subjected to the *Solution Combination Method* and none of the improved trial solutions are admitted into the *RefSet* under the rules of the *Reference Set Update Method*. However, in advanced scatter search designs, the *RefSet* rebuilding is applied at this point and the best $b/2$ solutions are kept in the *RefSet* while the other $b/2$ are selected from P, replacing the worst $b/2$ solutions, as shown in Figure 1. For other possible advanced designs see [38].

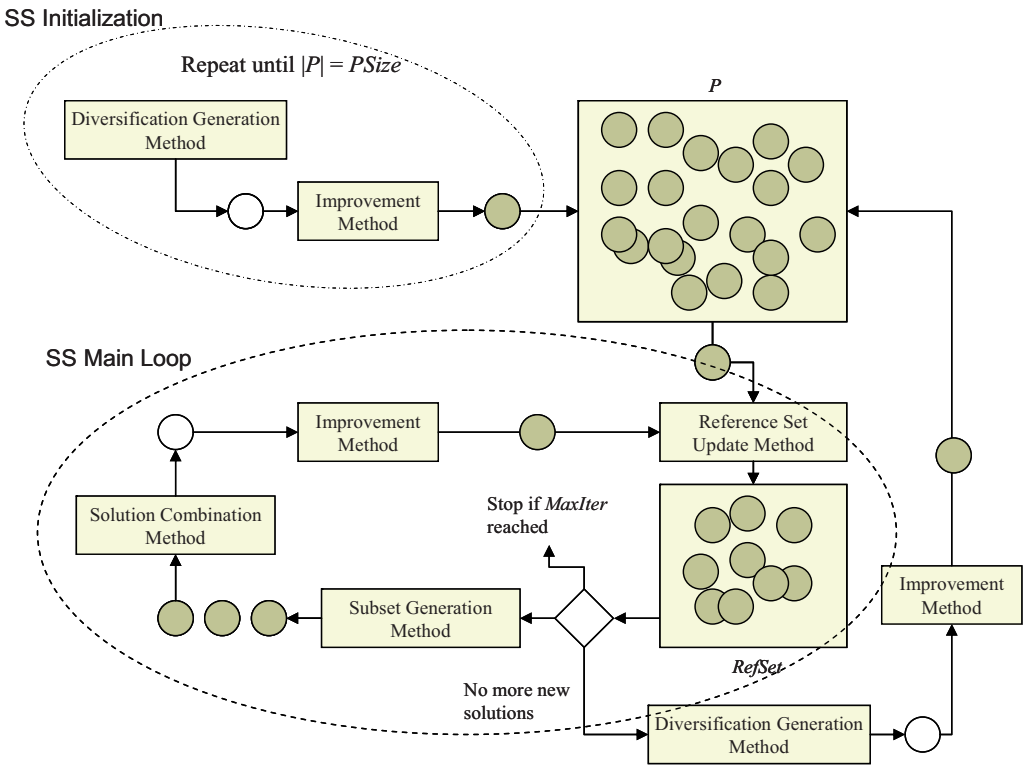


Figure 1: Schematic representation of the scatter search design

The *RefSet* is a collection of both high quality solutions and diverse solutions that are used to generate new solutions by way of applying the *Solution Combination Method*. We can use a simple mechanism to construct an initial reference set and then update it during the search. The size of the reference set is denoted by $b = b_1 + b_2$. The construction of the initial *RefSet* starts with the selection of the best b_1 solutions from P. These solutions are added to *RefSet* and deleted from P. For each solution in the updated P, the minimum of the distances to the solutions in *RefSet* is computed. Then, the solution with the maximum of these minimum distances is selected. This solution is added to *RefSet* and deleted from P, and the minimum distances are updated. The process is repeated b_2 times, where $b_2 = b - b_1$. The resulting *RefSet* has b_1 high quality solutions and b_2 diverse solutions. Algorithm 1 shows a basic scatter search procedure in pseudocode.

Of the five methods in scatter search methodology, only four are strictly required. The *Improvement Method* is usually needed if high quality outcomes are desired, but a scatter search procedure can be implemented without it as it occurs in some problems where the *Improvement Method* can not provide high quality solutions due to the problem's nature or when the computation budget is limited to a small number of function evaluations. On the other hand, hybrid scatter search designs

Algorithm 1 Basic Scatter Search procedure

Start with $P = \emptyset$
repeat
  Use the *Diversification Generation Method* to construct a solution and apply the *Improvement Method*
  Let x be the resulting solution
  if $x \notin P$ **then**
    $P = P \cup x$
  else
    Discard x
  end if
until $|P| = PSize$
Use the *Reference Set Update method* to build $RefSet = \{x^1, \ldots, x^b\}$ with the best b_1 quality solutions and b_2 diverse solutions ($b_1 + b_2 = b$) in P
Sort the solutions in *RefSet* according to their objective function value such that x^1 is the best solution and x^b the worst
Make *NewSolutions* = TRUE
while *NewSolutions* **do**
  Generate *NewSubsets* with the *Subset Generation Method*
  Make *NewSolutions* = FALSE
  while *NewSubsets* $\neq \emptyset$ **do**
    Select the next subset s in *NewSubsets*
    Apply the *Solution Combination Method* to s to obtain new trial solutions
    Apply the *Improvement Method* to the trial solutions
    Apply the *RefSet Update Method*
    if *Refset* has changed **then**
      make *NewSolutions* = TRUE
    end if
    Delete s from *NewSubsets*
  end while
end while

could incorporate a short-term tabu search or other complex metaheuristic (usually demanding more running time).

It is interesting to observe similarities and contrasts between scatter search and the original GA proposals. Both are instances of what are sometimes called "population based" or "evolutionary" approaches. Both incorporate the idea that a key aspect of producing new elements is to generate some form of combination of existing elements. However, GA approaches are predicated on the idea of choosing parents randomly to produce offspring, and further on introducing randomization to determine which components of the parents should be combined. By contrast, the scatter search approach does not emphasize randomization, particularly in the sense of being indifferent to choices among alternatives. Instead, the approach is designed to incorporate strategic responses, both deterministic and probabilistic, that take account of evaluations and history. Scatter search focuses on generating relevant outcomes without losing the ability to produce diverse solutions, due to the way the generation process is implemented.

As other metaheuristics, scatter search has been mainly applied to optimization problems involving integer variables (see [20] for a list of scatter search applications). However, some adaptations to continuous problems have arisen in the last years. Scatter search approaches for continuous optimization were presented in [16] and [50]. In [51] scatter search was combined with other metaheuristics. The OptQuest callable library, based on scatter search, was presented in [31] and used in [52]. Laguna and Martí [33] tested some advanced scatter search designs for global optimization and later Herrera el al. [25] analyzed the performance of different combination methods and improvement strategies. Other advanced implementations have been applied to parameter estimation in systems biology [47], chemical and bioprocess optimization [12] and food engineering optimization [46]. Egea et al. [13] developed a scatter search-based algorithm for computationally expensive process models.

1.2 Scatter Search tutorial

When presenting an optimization algorithm one usually tries to illustrate as deep as possible the way that it works. Detailed explanations, pseudocodes, figures and application examples are usually used for this purpose. In [21] a useful step-by-step scatter search tutorial for continuous problems is presented. It helps non-experts to start their first implementation. In the next lines, another tutorial of the scatter search basic scheme is presented, based on figures showing solutions over the search space. To follow it, some points must be considered:

1. We will consider a minimization problem.
2. Numbers inside every solution (circles) represent their objective function values.
3. Number of diverse solutions to initiate the procedure, $PSize = 10$.
4. *RefSet* dimension, $b = 4$ solutions.
5. The function to be optimized has 3 minima, 2 of them local (green squares) and 1 global (blue square).
6. For the sake of clarity, the *Improvement Method* is only applied in the last figure of this section.

1.2.1 Initialization

The algorithm starts by creating a set P of $Psize$ diverse solutions (10 in our case) with a sampling technique which can be simple randomization, latin hypercube sampling or any other strategy like those proposed in [32]. Figure 2 shows the initial set of diverse solutions at the beginning of the procedure.

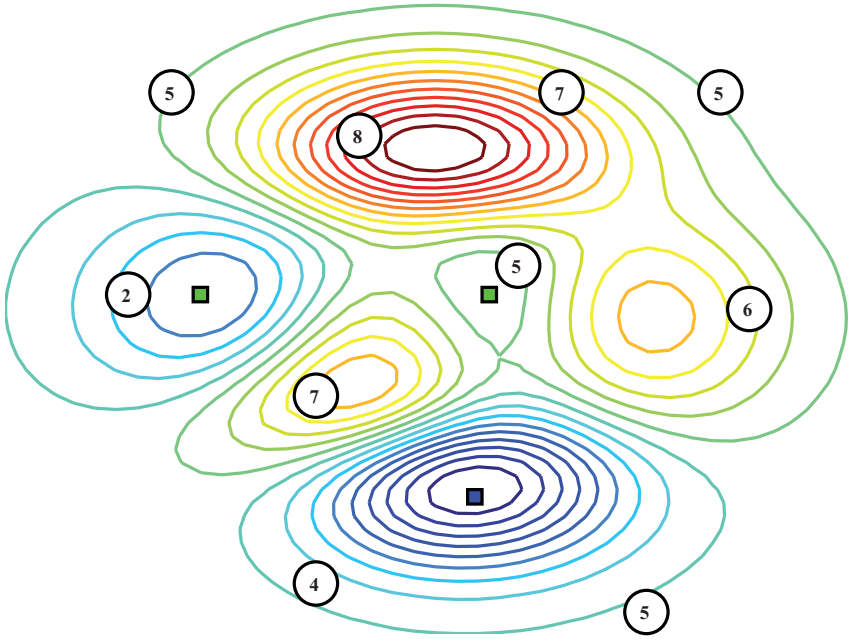


Figure 2: Initial set of diverse solutions

1.2.2 First *RefSet* formation

The initial *RefSet* must contain both quality and diverse solution. The *RefSet Update Method* is called for the first time. Thus, as explained in Section 1.1, $b_1 = 2$ solutions will be selected

by quality (i.e., those with the smallest function values since we are dealing with minimization). Then, $b_2 = 2$ solutions must be selected following a diversity criterion. In Figure 3, the first b_1 solutions (in red) are automatically selected according to their function value. The next solution (in grey) is selected by maximizing the distance to all the b_1 solutions and the next one (in black) uses the same principle taking into account the b_1 and the rest of solutions already included in b_2. The rest of solutions not included in the *RefSet* are discarded.

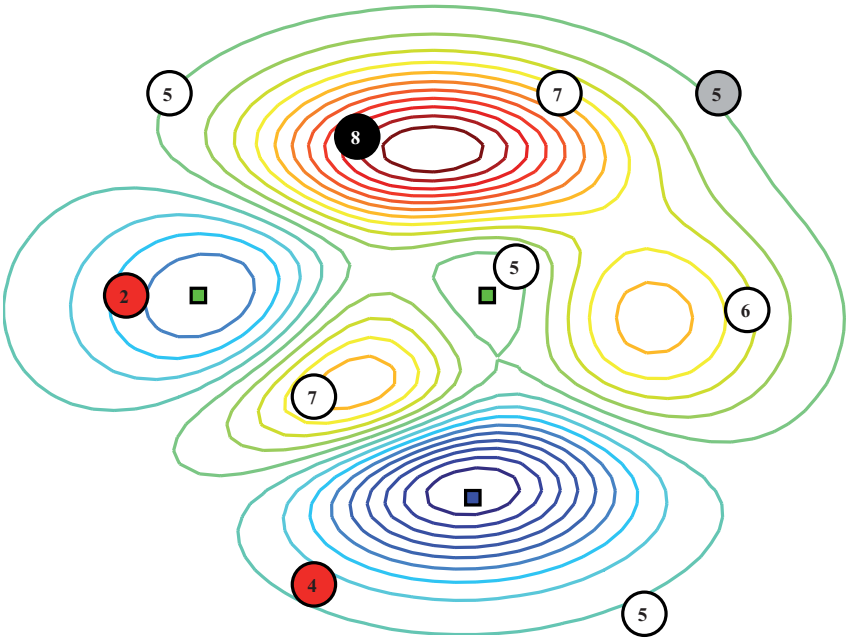


Figure 3: First *Refset* formation

1.2.3 Subset generation and combination

Once the initial *RefSet* has been formed, the *Subset Generation Method* creates different sets of solutions to be combined. Then, every set of solutions generates one or more solutions by applying the *Solution Combination Method*. In Figure 4, the sets are all the pairs of solutions in *RefSet*. The Solution Combination Method, in this case, consists of generating a solution by each pair, inside the segment linking the two solutions of the pair. The generated solutions are represented in blue in Figure 4.

1.2.4 *RefSet* update

After generating new solutions, the *RefSet Update Method* is called again to update the members of the *RefSet*. The update can be done by quality, diversity or a combination of both strategies [33]. Here we will update the *RefSet* by quality, thus the best b best solutions among the last *RefSet* members and the new generated solutions are selected as the new *RefSet* members. Those solutions with the minimum function values are selected. In our example, the new *RefSet* is composed by 2 new solutions and 2 solutions which were part of the *RefSet* in the previous iteration (see Figure 5).

1.2.5 *RefSet* regeneration

The previous steps, including the *Improvement Method*, which will be illustrated in Section 1.2.6, are repeated until a termination criterion is met. Scatter search usually makes use of a memory element to avoid performing combinations among sets of solutions already combined. It may occur

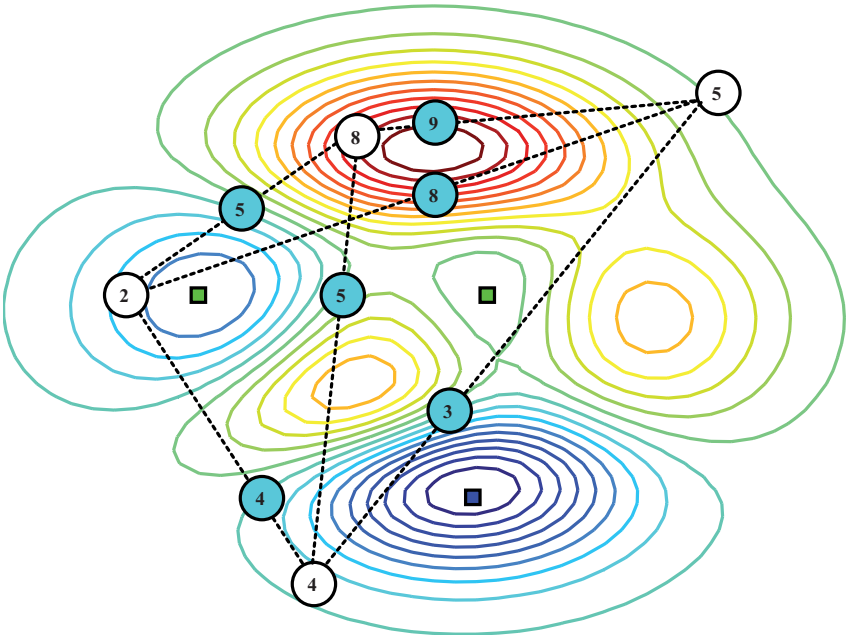


Figure 4: Combination of every pair of solutions in *RefSet*

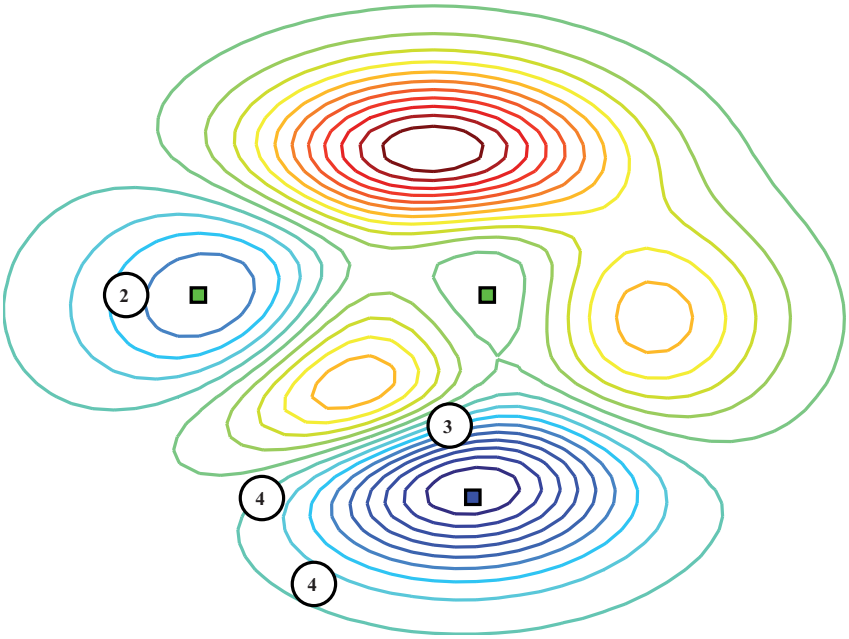


Figure 5: New *Refset*

that, at some point, no new subsets are available. At this moment, the algorithm can either stop or perform a regeneration which usually consists of deleting the worst $b/2$ solutions in the *RefSet* in terms of quality and replace them by diverse solutions. For this purpose, the *Diversification Generation Method* is used again to generate a set of diverse solutions (or we can simply use again the same set of diverse solutions used in the initialization of the method). Following the same diverse criterion as the one used in the first *RefSet* formation (see Section 1.2.2) those solutions maximizing their distance to the current *RefSet* solutions will be added to the *RefSet*. If the *RefSet* should be regenerated in a situation like that depicted in Figure 5, the 2 solutions to be deleted would be those with function value equal to 4 (since they are the b_2 worst solutions in terms of quality). The *Diversification Generation Method* would create new diverse solutions and those

maximizing their distance with respect to the remaining solutions in the *RefSet* would enter it. Note that new solutions gain the *RefSet* membership one by one in a sequential way by maximizing their distance to the current solutions in the *RefSet*. In Figure 6 the 2 solutions in red would replace the deleted solutions.

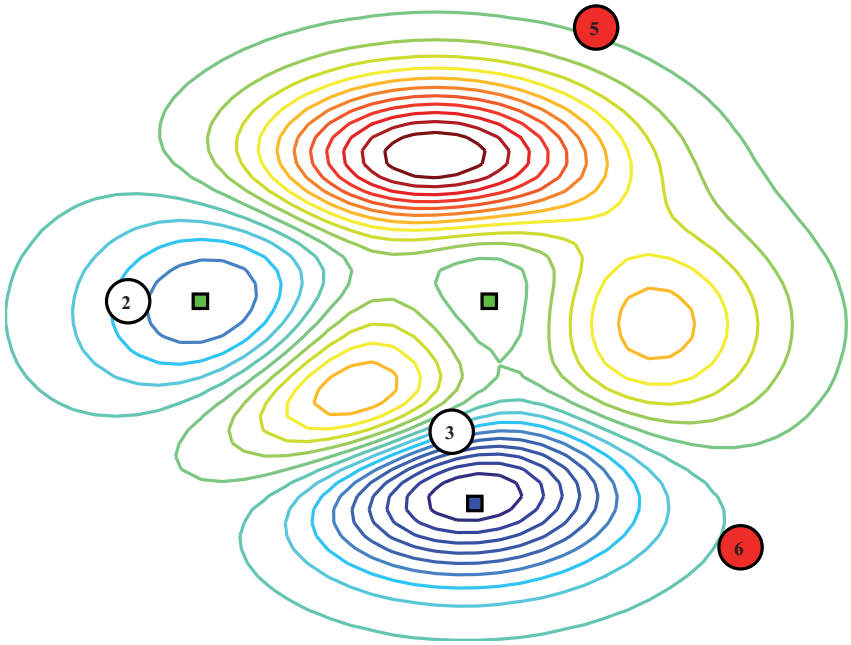


Figure 6: *Refset* regeneration

1.2.6 Improvement method

As stated in Section 1.1, during the optimization procedure all the generated solutions are subjected to the *Improvement Method* which usually consists of a local search procedure to improve the quality of the solutions. In some applications where the local search provides poor results it might be suppressed or an alternative *Improvement Method* must be designed (such as another metaheuristic hybridized with scatter search). Here we will only illustrate how the *Improvement Method* usually works by applying it to two solutions (the best 2 solutions of our particular *RefSet*). In Figure 7 the application of the *Improvement Method* to two different solutions results in finding two optima, one of them being the global optimum.

Advanced scatter search implementations take into account different parameters such as quality of the solutions and distances to found local minima in order to minimize the computational effort, avoiding to perform local searches from initial solutions which are likely to provide already known local minima.

2 *SSm*: A scatter search heuristic for bioprocess optimization

Many real world optimization problems in bioprocess engineering (and also in business or economics) are too complex to be given tractable mathematical formulations. Here we consider the general case in which there is no explicit expression of the objective function since it contains multiple nonlinearities, combinatorial relationships and uncertainties inaccessible to modeling except by resorting to more comprehensive tools like computer simulation. In the context of optimizing simulations, a "complex evaluation" refers to the execution of a simulation model (which can be extremely time-consuming).

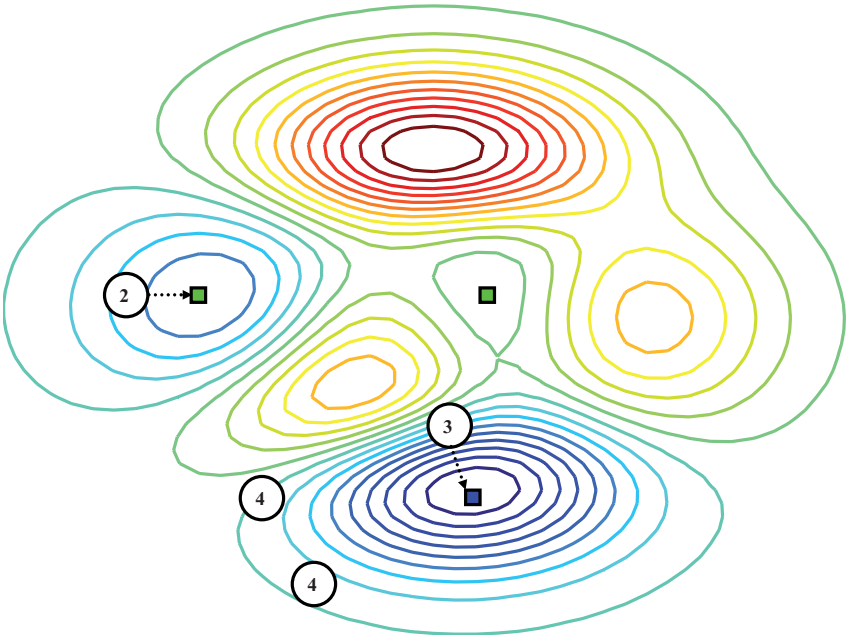


Figure 7: *Improvement Method* applied to 2 solutions in the *RefSet*

Theoretically, the issue of identifying best values for a set of decision variables falls within the realm of optimization. Until quite recently, however, the methods available for finding optimal decisions have been unable to cope with the complexities and uncertainties posed by many real world problems of the form treated by simulation. The area of stochastic optimization has attempted to deal with some of these practical problems, but the modeling framework limits the range of problems that can be tackled with such technology. The complexities and uncertainties in these systems are the primary reason to often choose simulation as a basis for handling the decision problems associated with them. Advances in the field of metaheuristics have led to the creation of optimization engines that successfully guide a series of complex evaluations with the goal of finding optimal values for the decision variables.

Most optimization problems in the chemical and bio-chemical industries are highly nonlinear in either the objective function or the constraints. Moreover, they show dynamic nature and can be formulated as nonlinear programming problems subject to differential-algebraic constraints. This set of constraints must be solved using specific mathematical techniques (e.g., initial value problem numerical methods) provided by the user or embodied in the objective function value to be optimized. Once this set of DAE's has been solved, and the objective function and additionally constraints have been evaluated, the output information is used by the optimization procedure to drive the search and choose the new solutions to be evaluated (i.e., the new inputs for the DAE's solver) in an iterative way. Figure 8 shows the interaction of the optimization procedure with the external mathematical method to solve the set of differential-algebraic constraints of the dynamic model to be optimized.

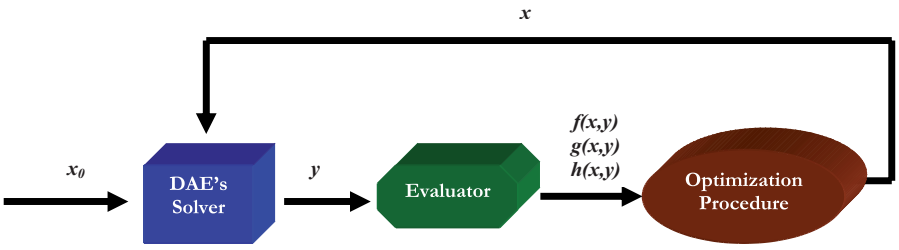


Figure 8: Interaction between the optimization procedure and the DAE's solver

In this context, optimization methods should be able to treat the optimization problems as "black-boxes". The disadvantage of "black-box" approaches is that the optimization procedure is generic and has no knowledge of the process employed to perform evaluations inside the box and therefore does not use any problem-specific information. The main advantage, on the other hand, is that the same optimizer can be applied to complex systems in many different settings. Therefore, although we have designed and tested our method in the process systems engineering environment, it can be directly applied to solve any kind of black-box optimization problems in other settings.

Deterministic (or exact) optimization algorithms work well only for small problems with some requirements (rarely met in real applications). Although they ensure the convergence to the global optimum, the computation time needed might be unaffordable. In contrast, stochastic (or heuristic) optimization methods can locate the global optimum or its vicinity in modest computation times for every type of problem. Many authors (e.g., see [5] and references therein) have successfully applied stochastic global optimization methods to process engineering optimization.

Metaheuristics are a kind of stochastic methods which arose to solve hard combinatorial optimization problems and were extended for solving continuous and mixed-integer problems in recent years. Current research in global optimization is highly devoted to metaheuristics. Here we propose a scatter search-based global optimization algorithm for continuous and mixed-integer problems which is efficient for solving optimization problems arising in bioprocess engineering. The motivation for choosing scatter search as a basis of our algorithm is that, in a recent review by [44] comparing several GO solvers over a set of 1000 constrained GO problems, the scatter search-based algorithm *OQNLP* obtained the best performance among the stochastic methods, solving the highest percentage of big problems.

2.1 Methodology

We have implemented our algorithm in Matlab under the name *SSm* (Scatter Search for Matlab). Our development goes beyond a simple exercise of applying scatter search to optimization problems in bioprocess engineering, but presents innovative mechanisms to obtain a good balance between intensification and diversification in a short-term search horizon.

2.1.1 *Diversification Generation Method*

SSm begins by generating an initial set S of m diverse vectors in the search space. Unlike other diversification strategies, *SSm* does not only generate vectors with their components uniformly distributed within the search space, but also drives the generation of values for each decision variable onto parts of the space where they have not appeared very often during the diversification process. For that, the method makes use of memory taking into account the number of times that every decision variable appears in different parts of the search space. This is usually accomplished by dividing the range of each variable into p sub-ranges of equal size. Then, a solution is constructed in two steps. First, a sub-range is randomly selected. The probability of selecting a sub-range is inversely proportional to its frequency count (which keeps track of the number of times the sub-range has been selected). Second, a value is randomly chosen from the selected sub-range.

Initially, the range of every decision variable, x_i with $i \in [1, 2, ..., n]$ (being n the problem size) defined by its lower and upper bounds, xl_i and xu_i respectively, is divided in p sub-ranges of equal size, $(xu_i - xl_i)/p$. Therefore, the limits that define each sub-range $j \in [1, 2, ..., p]$ for the variable i are given by

- Lower bound:
$$lb_{ij} = xl_i + \frac{xu_i - xl_i}{p}(j-1) \tag{1}$$
- Upper bound:
$$ub_{ij} = xl_i + \frac{xu_i - xl_i}{p}j \tag{2}$$

Frequencies, f_{ij}, are defined as the number of times that the variable i is in the sub-range j along all the generated vectors.

To initialize all the frequencies to a value of 1, p vectors are first generated, each of them having all their variables randomly generated in the same sub-range using a uniform distribution (e.g., vector 1, x^1, has all its variables in sub-range 1, and every decision variable i is randomly generated using a uniform distribution within the bounds xl_i and $xl_i + \frac{(xu_i - xl_i)}{p}$). This first set of vectors forms the initial matrix of diverse vectors $S^{p \times n}$ that will be extended up to a size of m-by-n by adding new diverse vectors.

$$S = \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^p \end{bmatrix} = \begin{pmatrix} x_1^1 & x_2^1 & \dots & x_n^1 \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^p & x_2^p & \dots & x_n^p \end{pmatrix} \tag{3}$$

The initial matrix of frequencies is defined as:

$$f = \begin{pmatrix} f_{11} & f_{12} & \dots & f_{1p} \\ f_{21} & f_{22} & \dots & f_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ f_{n1} & f_{n2} & \dots & f_{np} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix} \tag{4}$$

New vectors will be generated using the following procedure: for each new vector x^{p+t} to be generated (with $t \in [1, 2, ..., m-p]$) the probability of having its decision variable i in the sub-range j is calculated as

$$prob_{i,j}^{p+t} = \frac{\frac{1}{f_{ij}}}{\sum_{k=1}^{p} \frac{1}{f_{ik}}} \tag{5}$$

Then, a uniformly distributed random number, rnd, in the interval $[0, 1]$ is generated. The next generated vector x^{p+t} will have its i-th component in the subrange $j = a$ for the first value of a that accomplishes

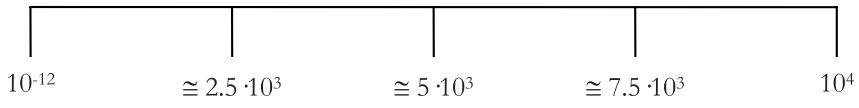
$$rnd \leq \sum_{j=1}^{a} prob_{i,j}^{p+t} \quad a = 1, 2, \dots, p \tag{6}$$

Each component, x_i^{p+t}, will take a value randomly selected using an uniform distribution in the range $[lb_{ij}, ub_{ij}]$. Thus, for a new vector to be generated, the probability of having the variable i in the sub-range j is inversely proportional to the frequency of appearance of the variables i in this sub-range considering the already created vectors. Therefore, the method has to "remember" and update these frequencies to enhance diversity. As new vectors x^{p+t} are generated, they are added to the matrix S in rows until it becomes m-by-n dimensional. The starting set of points also includes the following three solutions: the first one in which all variables are set to the lower bound, the second one in which all variables are set to the upper bound, and the third one in which all variables are set to the midpoint between both bounds. This is the standard scatter search implementation of the *Diversification Generation Method* for non-linear problems and is used by different methods like *OptQuest* [31]. However, we have found that in some instances in which variables may have values in a huge range of positive values, a logarithmic distribution usually provides better results.

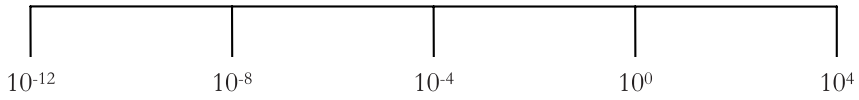
In the context of chemical and bio-process optimization, the selection of the lower bounds for the decision variables is usually quite straightforward because of their physical meaning (e.g.,

a temperature can never have a value lower than 0 Kelvin). However, the selection of the upper bounds is not so easy and they are often chosen as arbitrary large values to contain all the potential values for each variable. Therefore, it is expected that the optimal and good solutions may lie, in general, closer to the lower bounds than to the upper bounds. In this context, a uniform distribution for selecting diverse solutions within the bounds will not generate many trial points with good values. In contrast, a logarithmic distribution will generate more trial vectors close to the lower bound, thus allowing the algorithm to be initialized with high quality members in the initial population, ensuring a faster convergence. Moreover, a logarithmic distribution is also helpful in the case of variables that can intrinsically have values in different orders of magnitude (as is the case of pre-exponential factors in kinetic equations) or with variables without physical meaning, for which selecting bounds is a difficult task. In order to obtain good initial values for these cases, an option for selecting variables in different orders of magnitude has been added in our implementation under the name *log_var*.

Figure 9 illustrates this situation. Consider a variable that takes values between 10^{-12} and 10^4. If we generate a starting set of points between those bounds using a uniform distribution, we will approximately obtain the same number of values in every interval shown in Figure 9(a). Alternatively, if we select the *log_var* option for this variable, its values will be randomly selected with equal probability across the sub-ranges depicted in Figure 9(b). With this option, the number of subintervals is automatically adjusted so that there are a maximum of two orders of magnitude between the limits of each interval (e.g., for a variable between 10^{-12} and 10^4, the number of subintervals would be 8), thus generating more solutions close to the lower bound.



(a) Values uniformly distributed within the ounds



(b) Values distributed within the different orders of magnitude

Figure 9: Intervals within a variable range

2.1.2 Building the *RefSet*

As described in Section 1.1, the *RefSet Update Method* is applied in two different steps of the algorithm: when building the initial *RefSet* from the set S of diverse solutions and when updating it after applying the *Solution Combination Method*.

For building the initial *RefSet*, after generating the set S of diverse solutions, two strategies may be chosen. In the first strategy (used by default), a subset of high quality and diverse points is selected as the *RefSet*. The initial *RefSet* is built selecting the best $b/2$ solutions from S as given by the evaluation-simulation process and then making more $b/2$ selections in order to maximize the minimum distance between the candidate solution and the solutions currently in *RefSet*.

The first step consists of evaluating all diverse vectors and select the $b/2$ best ones in terms of quality. For example, in a minimization problem, provided the diverse vectors are sorted according to their function values (the best one first), the initial selection is $\left[x^1, x^2, \ldots, x^{b/2}\right]^T$ such that

$$f(x^i) \leq f(x^j) \;\; \forall \;\; j > i \;,\; i \in [1, 2, \ldots, b/2 - 1] \;,\; j \in [2, 3, \ldots, b/2] \tag{7}$$

Vectors added to the *RefSet* are deleted from S. The current number of vectors present in the *RefSet* is computed as h. Therefore, in this stage $h = b/2$ (and the maximum possible value of h is b). We complete the *RefSet* with the remaining diverse vectors in S by maximizing the minimum Euclidean distance to the included vectors in the *RefSet*.

For every diverse vector in S, x^d, with $d \in [h+1, h+2, ..., m]$, Euclidean distances to all current *RefSet* vectors are computed. The minimum of these distances, d_{min}, is stored for each vector:

$$d_{min}(x^d) = min\{d(x^d, RefSet)\} \tag{8}$$

where $d(x^d, RefSet)$, represents a vector whose components are the Euclidean distances between vector x^d and all the vectors in the *RefSet*. Then, the vector $\mathbf{x}$ having the highest minimum distance will join the *RefSet*. Therefore, $RefSet = RefSet \cup \mathbf{x}$ such that

$$d_{min}(\mathbf{x}) = max\left(d_{min}(x^d)\right) \quad \forall \; d = h+1, h+2, ..., m \tag{9}$$

and the value of h is increased one unit since a new vector has been added to the *RefSet*. This is repeated until the *RefSet* is filled with b vectors (i.e., $h = b$) so that $RefSet \in \mathbb{R}^{b \times n}$.

This criterion is applied in a sequential fashion. At each step we add to the *RefSet* the solution that maximizes $d_{min}(x^d)$, remove it from S, and then recalculate the Euclidean distances. Therefore, we add one solution at each step until the *RefSet* has been completed (i.e., we do it for $b/2$ steps).

This strategy requires $|S|$ simulations to identify the best $b/2$ solutions in terms of the objective function value. Unless we choose a low value for $|S|$, this can cause a waste of computational effort, especially in the case of time-consuming problems. We therefore propose an alternative strategy which does not take into account the quality of the diverse vectors. The initial *RefSet* is formed by 3 vectors: one having all the variables in their lower bounds, another one having all the variables in their upper bounds and the middle point between these two vectors. This initial $RefSet \in \mathbb{R}^{3 \times n}$ is completed using the same distance criterion described in the first strategy until it is composed of b decision vectors.

Note that the first strategy involves a higher computational cost since the all the diverse vectors have to be evaluated. However, this strategy ensures a better quality of the initial *RefSet* which can help to converge faster to the global solution. The second strategy does not involve any simulation prior to the optimization stage. We therefore have no information about the quality of these solutions and thus we expect the algorithm to converge more slowly. The first strategy combines quality and diversity in the initial *RefSet*, whereas the second focuses only on diversity (and saves computational effort).

2.1.3 *Subset Generation* and *Solution Combination* methods

After the initial *RefSet* is built, its solutions are sorted according to their quality (i.e., the best solution is the first) and we apply the *Subset Generation Method*. Laguna and Martí [37] stated that most of the quality solutions obtained by combination arise from sets of two solutions, thus, in our implementation, the *Subset Generation Method* consists of selecting all pairs of solutions in the *Refset* to combine them. To avoid repeating combinations with the same pair of solutions, we use a memory term which keeps track of the pairs previously combined.

The *Solution Combination Method* is a key element in scatter search implementations. This method is typically adapted to the problem context. Linear combinations of two solutions were suggested by Glover [18] in the context of nonlinear optimization and are a generalization of the linear or arithmetical crossover also used in continuous and convex spaces [40]. Herrera et al. [25] studied different types of combination procedures for scatter search applied to continuous problems. They concluded that the BLX-α algorithm (with $\alpha = 0.5$) is a suitable combination method for continuous scatter search. Laguna and Martí[33] already used this idea and extended it to avoid generating solutions in the same area by defining up to four different regions within and beyond the

segments linking every pair of solutions. These authors changed the number of created solutions from each pair of solutions in the *RefSet* depending on the position of the latter in the sorted *RefSet*. Here we will use the same principles, but instead of performing linear combinations between solutions, we will perform a type of combination based on hyper-rectangles, which enhances the diversification.

These combinations are of the following four types, assuming that x' and x'' are the solutions to be combined and that x' is superior in quality to x'':

$$c_1 = x' - d_1 \tag{10}$$
$$c_2 = x' + d_2 \tag{11}$$
$$c_3 = x'' - d_3 \tag{12}$$
$$c_4 = x'' + d_4 \tag{13}$$

where $d_i = r_i \bullet (x" - x')/2$ with $i = 1, 2, 3$ or 4 depending on the number of solutions generated (see below); r_i is a vector of dimension n with all its components being uniformly distributed random numbers in the interval $[0, 1]$. The notation ($\bullet$) above indicates an entrywise product (i.e., the vectors are multiplied component by component), thus it is not a scalar product. The vector d_i has the following form:

$$d_i = \begin{pmatrix} d_{i,1} \\ d_{i,2} \\ \vdots \\ d_{i,n} \end{pmatrix}^T = \begin{pmatrix} \frac{r_{i,1}(x''_1 - x'_1)}{2} \\ \frac{r_{i,2}(x''_2 - x'_2)}{2} \\ \vdots \\ \frac{r_{i,n}(x''_n - x'_n)}{2} \end{pmatrix}^T \tag{14}$$

Note that if both solutions, x' and x'', belong to the first $b/2$ elements of the sorted *RefSet*, then 4 vectors are generated: one each type. If only x' belongs to the first $b/2$ elements of the sorted *RefSet*, then 3 vectors are generated (types 1, 2 and 4). Finally, if neither x'' nor x' belong to the first $b/2$ elements of the sorted *RefSet*, then 2 vectors are generated: one of type 2 and another one of type 1 or 3 (randomly chosen). Figure 10 illustrates the type of combinations and the regions in which new solutions are created.

These vectors generated by combination of the *RefSet* members will be named x^c with $c \in [1, 2, ..., nc]$, and form a matrix $C \in \mathbb{R}^{nc \times n}$ where nc is the total number of vectors generated by combination, which is not a fixed number. It may change in every iteration depending on the number of combinations made among *RefSet* members (remember that the method avoids doing combinations with pairs of vectors already combined).

It must be noted that the *Solution Combination Method* does not have any checking mechanism to detect whether a new vector has any of its variables out of the bounds. Therefore, before evaluating these new vectors, a quick check is done by the algorithm: if any of the decision variables is out of the interval defined by its bounds, it is automatically adjusted to the value of the closest bound to it. If the *RefSet* changes after the application of the *RefSet Update Method* described in Section 2.1.4, indicating that at least one new solution has been inserted in the *RefSet*, we apply again the *Solution Combination Method* to all the pairs in *RefSet* containing at least one new element. Otherwise, as in advanced scatter search designs, we resort to the rebuilding mechanism as described in Section 2.1.6.

2.1.4 Updating the *RefSet*

In its original design, the *Reference Set Update Method* indicates that the *RefSet* is updated by selecting a set of high-quality and diverse solutions from the union of the *RefSet* and the new combined solutions. In [37], it is pointed out that the *RefSet* is usually updated considering the

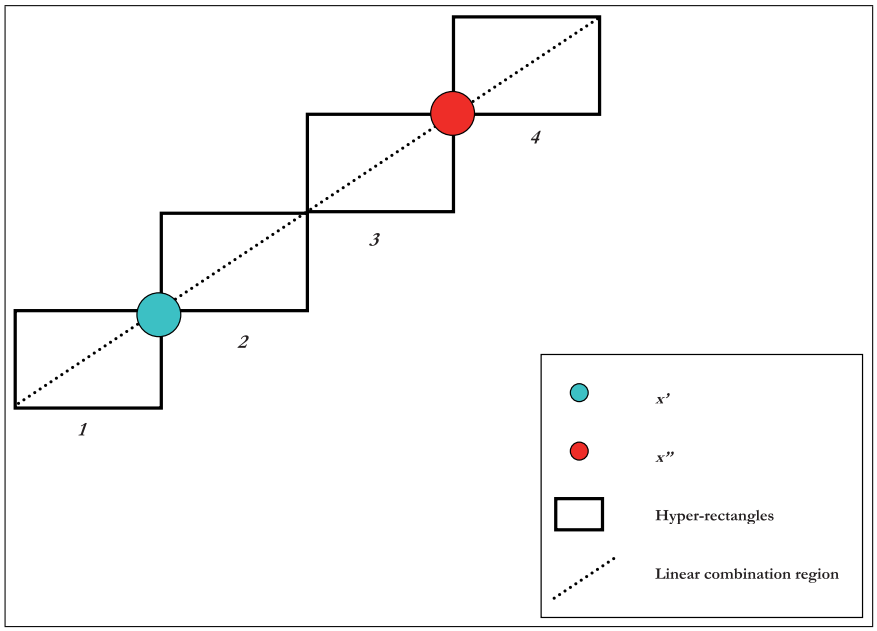


Figure 10: Combination method

quality of the elements.. However, we have empirically found that, for continuous problems, this standard mechanism tends to create clusters of solutions, which results in an intermediate *RefSet* with very similar solutions which are unlikely to produce new good solutions by combination. This effect also appears in other metaheuristics applied to continuous problems and have been overcome in different ways (see for example 24 and 49). We have added a distance filter to prevent similar solutions from becoming part of the *RefSet*. Specifically, we define a threshold value, *dth*, as a minimum Euclidean distance to be accomplished by every solution. This mechanism is illustrated in Figure 11: in a minimization problem, having 5 candidate solutions to form a *RefSet* of 4 members and a defined *dth*, we would start adding to the new *RefSet* the solution with the highest quality. We would add the rest of solutions to fill the *RefSet* regarding their quality and providing they comply with *dth*. In Figure 11, after adding the first two solutions (with function values equal to 1 and 3 respectively) we analyze the next candidate by quality (i.e., the solution with function value equal to 4). Since its distance to one of the solutions already in *RefSet* (i.e., the solution with function value equal to 3) is lower than the specified *dth*, it would not enter the *RefSet* and the procedure would continue analyzing the following candidate.

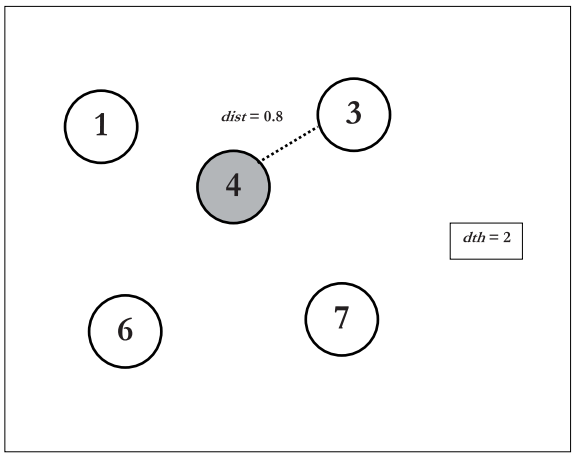


Figure 11: *RefSet* update with a threshold distance

This filter avoids clustering of the *RefSet* solutions thus preventing the search to prematurely

converge to a sub-optimal solution. The price to be paid is to diminish the average quality of the *RefSet*, which may be a drawback for problems with a small budget of simulations.

The parameter *dth* is initialized as the minimum Euclidean distance among the members of the first *RefSet*. It increases or decreases its value dynamically depending on the search history. If the best solution found does not improve in 2 consecutive iterations, *dth* decreases its value 10%. If we improve the best solution in 4 consecutive iterations, *dth* increases its value 10%. We have empirically found that, with this scheme, the *dth*-value is reduced in the last iterations, permitting the final refinement of the solutions.

A different criterion has been implemented for the cases in which the Euclidean distance criterion is inefficient. Indeed, we normalize Euclidean distances with respect to the bounds of the decision variables in order to have a similar contribution of each variable regardless their order of magnitude. If these bounds are not wisely chosen (e.g., the variables have no physical meaning or we simply have no idea of the practical range of them), this strategy can make the elements in the *RefSet* not to be as diverse as we wish. To avoid this, a second strategy based on differences in the decision variables can be chosen. According to this criterion, two solutions will be different if their relative difference in all variables is equal or greater than a value specified by the user.

We have included a second filter to prevent the method from being trapped in a region for a large number of iterations. In particular, if a solution is relatively far from the *RefSet* members but presents a very similar objective value to any of them (as it may happen in functions with flat landscape), we do not allow it to enter the *RefSet*. This prevents vectors in the same flat area from joining the *RefSet* at the same time. Provided the diversity criterion (defined by *dth*) is accomplished, the candidate vector z will join the *RefSet* only if

$$f(z) < f(x)(1-\varepsilon) \;\; \forall \;\; x \in RefSet \tag{15}$$

where ε is a small value defined by the user.

Figure 12 illustrates this situation in a minimization problem. Consider a solution x in the *RefSet* and a candidate solution z to enter it. Suppose that z, verify the distance filter according to *dth* and x has a slightly better value (say around 0.1% lower) than z. Then, instead of directly adding z to the *RefSet*, the quality filter considers that it may lie in the same flat area as x and forbids the action in order to"wait" for a better solution (thus performing a more aggressive search).

In accordance with the problem's characteristics, the user adjusts this filter value, ε, for an optimal algorithm performance. The default value for this filter is relatively conservative, but it should be changed in problems in which we want to enhance diversity (for example when there are multiple local minima and the global optimum has a small basin of attraction). When relying on local search, the search may be more aggressive, whereas if no *Improvement Method* is present, it is recommended that the default conservative value is used. In more advanced designs, this filter could be dynamic, being more relaxed at the beginning of the search in order to quickly locate the basin of attraction of the global minimum, and tighter at the end of the search to allow a specified tolerance in the solution. The evolution of this filter is not obvious and needs experimental work. The algorithm performs very well with a constant value.

Note that the two implemented filters act restricting the incorporation of solutions that contribute only slight quality and diversity to the current *RefSet*. Therefore, the scatter search design by itself will make the search more efficient over a long term horizon.

To summarize how the *RefSet Update Method* is working in our algorithm taking into account the filters described, Algorithm 2 shows a pseudocode of the procedure.

2.1.5 *Improvement Method*

The *Improvement Method* consists of a local search with the appropriate algorithm, using a carefully selected solution as the starting point. One of the advantages of implementing our optimization

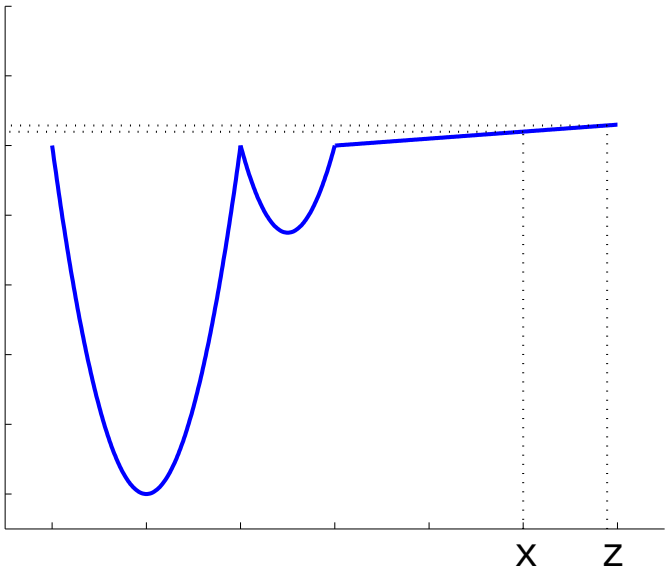


Figure 12: Two solutions in a flat zone of the objective function

Algorithm 2 *SSm RefSet Update*

Create a set, *tempset*, with the *RefSet* and the generated solutions by combination: $tempset \in \mathbb{R}^{b+nc\times n} = RefSet \cup C$
Sort solutions in *tempset* by quality
Clear the *RefSet*: $RefSet = \emptyset$
Add the best solution in *tempset* to the *RefSet* and clear it from *tempset*
while $|RefSet| < b$ **do**
  for $i = 1$ to $|tempset|$ **do**
    $x_t = x^1_{tempset}$
    Delete $x^1_{tempset}$ from *tempset*
    if $\min\{d(x_t, x^i)\} > dth$ **and** $f(x_t) < f(x^i)(1-\varepsilon)\ \ \forall\ \ x^i \in RefSet$ **then**
      $RefSet = RefSet \cup x_t$
      break for
    end if
  end for
end while

method in the Matlab environment is that we can easily apply any improvement method available in one of the many existing libraries. We have considered the following methods:

- *fmincon*: this is a local gradient-based method, implemented as part of the Matlab optimization Toolbox, which finds a local minimum of a constrained multivariable function by means of a SQP (Sequential Quadratic Programming) algorithm. The method uses numerical or, if available, analytical gradients.
- *solnp*: the SQP method by Ye [54].
- *fsqp*: this algorithm is a SQP method for minimizing smooth objective functions subject to general smooth constraints. The successive iterates generated by the algorithm all satisfy the constraints [45].
- *ipopt*: Interior Point OPTimizer is a software package for large-scale nonlinear optimization. It is designed to find (local) solutions of nonlinear programs [53].

- *misqp*: the solver called Mixed-Integer Sequential Quadratic Programming (MISQP) is a SQP Trust-Region method, recently developed by Exler and Schittkowski [14], [15], which handles both continuous and integer variables.
- *n2fb*: this algorithm was specially designed for non-linear least squares problems by Dennis et al. [9]. The method is based on a combined approximation of a Gauss-Newton and quasi-Newton algorithm.
- *lsqnonlin*: this method is also designed for non-linear least squares problems. It can use different algorithms such as the interior-reflective Newton method [7], the Levenberg-Marquardt method with line search [43] or a Gauss-Newton method with line search [10].
- *fminsearch*: this is a direct search method implemented in Matlab, that uses the simplex search method of Lagarias et al. [30]. It does not use numerical or analytic gradients. We have adapted the original code to handle nonlinear constraints.
- *NOMADm*: Nonlinear Optimization for Mixed variables And Derivatives-Matlab, abbreviated as NOMADm [1], is a Matlab code that runs various Generalized Pattern Search (GPS) algorithms to solve nonlinear and mixed variable optimization problems. This solver is suitable when local gradient-based solvers do not perform well.
- *dhc*: the Dynamic Hill Climbing algorithm by de la Maza and Yuret [8] is a direct search algorithm which explores every dimension of the search space using dynamic steps. Only the local phase of the algorithm has been implemented.
- *Hooke & Jeeves*: The Hooke & Jeeves algorithm [26] is a pattern search method for optimization of function where gradient is not available or it is expensive to calculate. We have used the Matlab implementation of Prof. C.T. Kelley*.

In a classical implementation of scatter search, the improvement method is applied to a large number of solutions (all the initial solutions in S and all the combined solutions from the *RefSet*). However, in applications related to chemical and bio-process engineering, we often face time-consuming evaluation problems (i.e., every function evaluation can consume several minutes) or complex topologies which can make the local search inefficient. This implies that the application of the *Improvement Method* should be restricted to a low number of promising solutions. This idea has also been used by other authors in memetic algorithms, assigning different probabilities to the individuals to be subject to a local search (see for example [36],[42]). It is expected that in the first iterations of the search process the solutions generated will be of a relatively poor quality. Therefore, we have implemented a parameter that determines the iteration number in which the *Improvement Method* is applied for the first time (i.e., defining a number of previous function evaluations before calling the *Improvement Method*). Then, once this is satisfied, both quality and diversity filters are applied. These filters were successfully applied in [52] and they do not allow the *Improvement Method* to be applied from a solution of a low quality (merit filter), or from a solution close to other from which the *Improvement Method* was applied in previous iterations (diversity filter). As documented by these authors, the filters significantly reduce the computational time with good results.

The merit filter ensures that no local search will be performed unless we do not find a better initial point than those found before. However, this filter is flexible since finding high quality solutions might be a time-consuming task for some problems and calling the local search from other points may be useful. An initial threshold is defined in the first call to the local search (e.g., as the function value of the first initial point). If no good initial points are found after a pre-defined number of iterations, the filter is relaxed (i.e., points with worse objective function values can be chosen as initial points to perform the local search) by means of a relaxation parameter th_{factor}. The threshold is relaxed calculating a new threshold from the existing one according with:

$$th_{new} = th_{old} + th_{factor}(1 + |th_{old}|); \tag{16}$$

*http://www4.ncsu.edu/∼ctk/darts/hooke.m

The distance filter computes the Euclidean distance between the local minima and the initial points used to locate them. This filter prevents the algorithm from doing local searches from initial points that might lead to already found local optima. It assumes spherical basins of attraction for the optima and defines the radius of a circle as the Euclidean distance from the initial point used for the local search and the optimum found. In practice, basins of attraction are not spherical, thus the distance filter often needs to be relaxed. If no initial points being far enough from found optima are found after a number of iterations, this filter is relaxed by multiplying the radii by a parameter in the interval $[0, 1]$.

To avoid overlapping of the circles defining the initial points and their respective local minima, a correction of this filter has been implemented. Figure 13 represents two local searches which lead to two different optima, activating the distance filters (dotted circles). To prevent other vectors leading to different optima from being discarded as initial points for the following local searches, a correction of the filters is applied (solid circles).

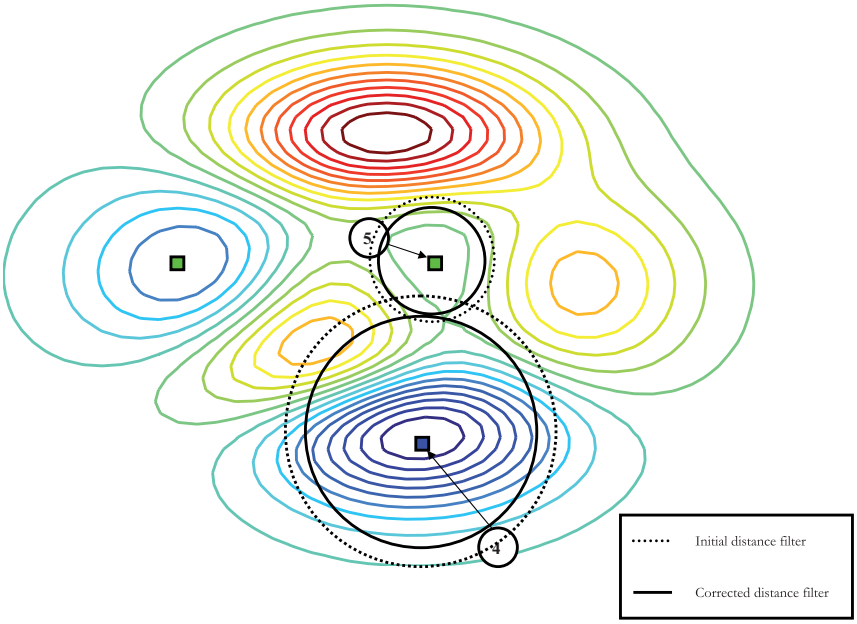


Figure 13: Correction of the distance filter overlap

For two different local minima, $\mathbf{x_i}$ and $\mathbf{x_j}$ their respective radii defining the distance filters are r_i and r_j, which must satisfy:

$$r_i + r_j \leq d(i, j) \tag{17}$$

where $d(i, j)$ is the Euclidean distance between $\mathbf{x_i}$ and $\mathbf{x_j}$.

The distance filter should be relaxed or even deactivated when many local solutions are closely located, as it is the case of some problems in bioprocess engineering optimization.

An alternative strategy has been implemented for applying the *Improvement Method* in our algorithm: instead of using filters, a local search is performed every time that the algorithm finds a better solution than the best current one, using as initial point this new found best solution. To avoid performing many local searches from similar initial points, a minimum number of function evaluations between two local searches is fixed. This allows the algorithm to search more globally without spending computation time in intermediate local searches without improving the best solution. Algorithm 3 shows a pseudocode of this procedure.

Algorithm 3 Alternative *Improvement Method* strategy

```
Set n1, n2
Set neval = 0
Apply the Diversification Generation Method
Build the initial RefSet
Perform a local search from the best solution so far after n1 evaluations
xbest = Local solution obtained
Start the SSm main routine
while not end of the optimization do
  if a solution, x*, better than xbest found and neval ≥ n2 then
    Perform a local search from x*
    xbest = Local solution obtained
    neval = 0
  end if
  neval = neval + function evaluations of current iteration
end while
```

2.1.6 *RefSet* Rebuilding

Rebuilding is a key operation associated with the *RefSet*. It implements a mechanism to partially rebuild the *RefSet* when none of the new trial solutions generated with the *Combination Method*, x^c, qualifies for addition to the *RefSet*. In advanced scatter search designs, the method is usually the same as that used to create the initial *RefSet*, in the sense that it uses the max-min distance criterion for selecting diverse solutions. Typically, the worst g vectors in *RefSet* (in terms of quality) are deleted. New diverse vectors are generated using the *Diversification Generation Method* and the *RefSet* is refilled according to the diversity criterion of maximizing Euclidean distances performed in the first *RefSet* formation. Normally, g is equal to $b/2$ but in aggressive implementations it can be set to b-1 (i.e., all the solution vectors in the *RefSet* except the best one are deleted).

We have modified the standard implementation of the rebuilding mechanism to incorporate the notion of orthogonality. Over a long-term horizon, the purpose of adding diverse solutions to the *RefSet* is to generate new search directions. It is therefore interesting not only to get scattered solutions in the search space, but also solutions that are able to create new search directions. Then, instead of selecting the solutions in S with the max-min distance, we select those with min-max cosine with the solutions already in the *RefSet*. Specifically, we choose the best element in *RefSet* as the center of gravity and in the first iteration apply the standard criterion to add the first diverse solution to the *RefSet*. Consider now the vector linking this new solution with the center of gravity. In subsequent iterations, instead of considering distances between the solutions in S and the *RefSet* members, we consider the vectors that the former define with the center of gravity and select the solution associated with the vector that minimizes the maximum value of the cosine among the vectors of the solutions already in the *RefSet*. In this strategy, the vectors refilling the *RefSet* are chosen to maximize the number of relative directions defined by them and the existing vectors in the *RefSet*.

Figure 14 illustrates both types of *RefSet Rebuilding*, the classical one, by distance, and our strategy, by direction. In Figure 14(a) two solutions of the *RefSet* (in white) have been kept, and the rest have been deleted. The *Diversification Generation Method* has created a set of diverse solutions (in grey) from which we will take two to refill the *RefSet*. The classical criterion of maximizing the Euclidean distance would select the yellow solution in Figure 14(b) as the next *RefSet* member. Our criterion selects the best solution remaining in the *RefSet* as a center of gravity and would generate all the vectors linking it to the rest of solutions in the *RefSet* (in this case just a vector since there are only two solution). Among the candidate solutions generated by the *Diversification Generation Method*, the solution defining a vector with the center of gravity which is as orthogonal as possible as the previously defined vectors, is selected as the next *RefSet* member (in Figure 14(b), the green solution). The process is repeated with the following solutions to be included in the *RefSet*. In Figure 14(c), the classical criterion would select again the furthest solution from the current *RefSet* members (in yellow again). Our criterion would analyze all the possible vectors

defined by linking the candidate solutions with the center of gravity and would select that solution defining a vector as orthogonal as possible to the rest of vectors defined by the center of gravity and the rest of solutions in the *RefSet* (in green again).

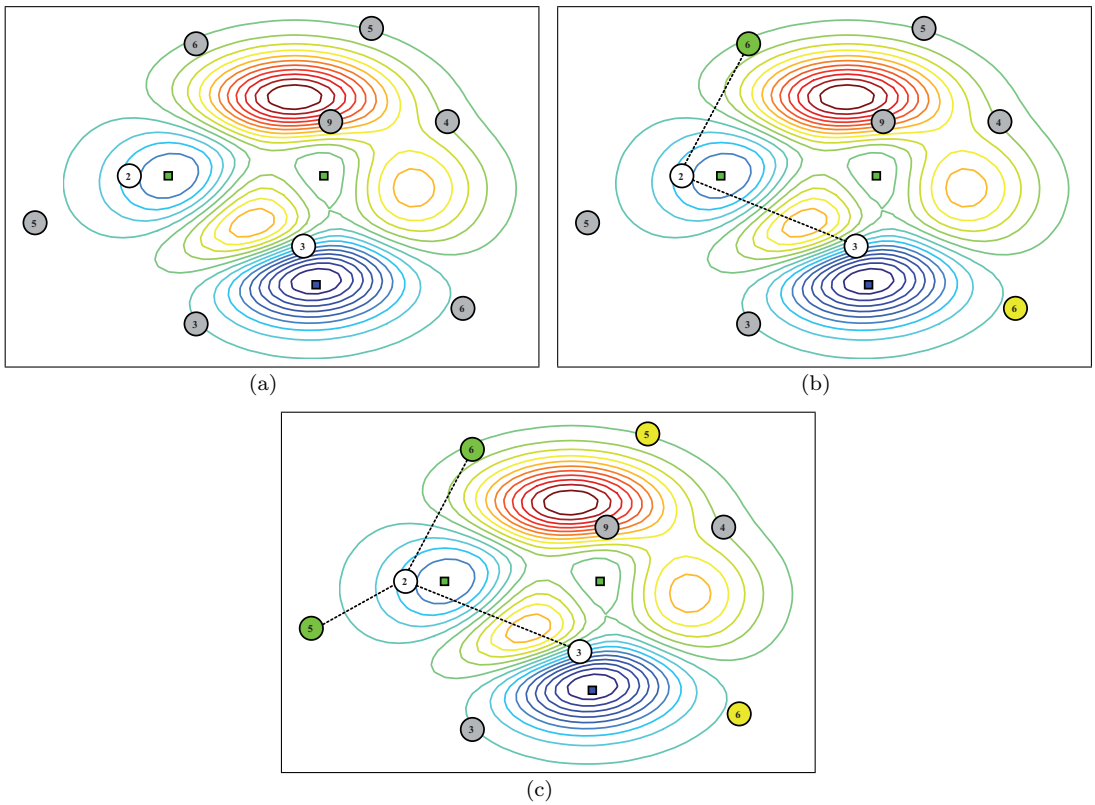


Figure 14: *RefSet* rebuilding by distance (yellow) and by direction (green)

In other words, after deleting the g worst solutions, the *RefSet* is $(b-g) \times n$ dimensional. Let $j = b - g$ be the number of existing vectors in the current *RefSet*. We introduce the new matrix $M^{(j-1)\times n}$ containing the vectors that define the segments formed by the best vector in *RefSet* (i.e., the center of gravity) and the rest of vectors in it. The $(k-1)th$ row of M is $x^1 - x^k$, being x^1 the center of gravity, and x^k $(k = 2, \ldots, j)$ the rest of the elements in it (note that the *RefSet* is sorted according to quality). For every diverse vector created with the *Diversification Generation Method*, x^d with $d \in [1, 2, \ldots, m]$ in the regeneration phase, a vector Q^d of scalar products is also defined

$$Q^d = (x^1 - x^d)M^T \tag{18}$$

where x^1 is again the center of gravity and M^T is the transpose matrix of M. For every x^d, the maximum value of its corresponding vector Q^d is computed as $msp(x^d)$. The solution $y \in x^d$ will join the *RefSet* in the regeneration phase if

$$msp(y) = \min\{msp(x^d)\} \qquad \forall d \in [1, 2, \ldots, m] \tag{19}$$

At this stage, the value of j is increased one unit, the value of m is decreased one unit and the process continues until $j = b$. The application of this strategy results in a maximum diversity in search directions on the regenerated *RefSet*. After every time the *RefSet Rebuilding* is carried out, the center of gravity is allowed to be combined again with all the rest of *RefSet* solutions, regardless the fact that it was previously combined with any of them. This has similarities with the *aspiration criterion* of tabu search, in which a forbidden movement is allowed if a predefined condition is met.

2.1.7 Intensification

The inclusion of the distance filter in the *RefSet Update Method* (Section 2.1.4) could be too restrictive if the parameter *dth* takes relatively large values, (or if the tolerance chosen by the user when using the other strategy is too high), thus rejecting too many solutions to become part of the *RefSet*. Instead of keeping this parameter under low values to prevent this effect, we have experimentally found that if we store the rejected solutions with good values in a secondary reference set, $RefSet_2$, we can treat them differently from the other solutions in the *RefSet*. $RefSet_2$ stores the solutions that do not qualify to enter into the *RefSet* and present a value close to the value of the best solution found (specifically, better than the second best solution in the *RefSet*). During the *Solution Combination Method*, we combine the best solution in the *RefSet* with all the solutions in $RefSet_2$ and add all the resulting solutions to the *RefSet Update Method* phase. This intensification mechanism is performed every $Intens_{freq}$ iterations.

Figure 15 illustrates those combinations in an example with four solutions in the *RefSet* (white and red circles) and two solutions in $RefSet_2$ (grey circles). The blue square represents the global optimum. In this example this intensification strategy makes the process converge faster since the combination of solutions in $RefSet_2$ (in grey) with the best in *RefSet* (red circle) generates solutions which are very close to the global optimum of the function.

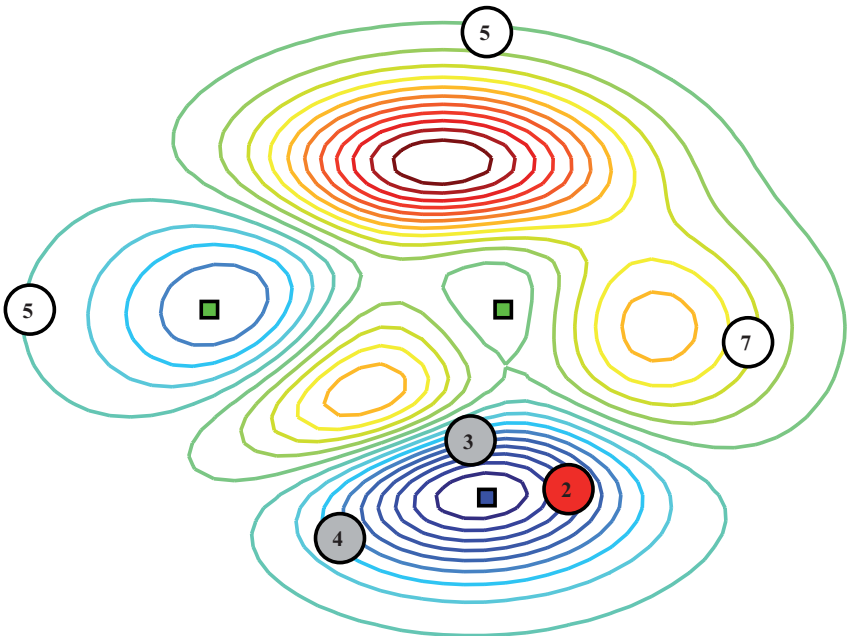


Figure 15: Intensification strategy

2.1.8 The *go beyond* strategy

Another advanced strategy to enhance the intensification of the search has been implemented in our algorithm. It has been named *go beyond* strategy and consists of exploiting promising directions. When performing the *Solution Combination Method* every generated vector is compared with its *parent*. If a new vector outperforms its parent in terms of quality, a new non-convex solution in the direction defined by the child and its parent is created. The child becomes the new parent and the new generated solution is the new child. If the improvement continues, we might be in a very promising area, thus the area of generation of new solutions is increased. This procedure is limited to the first $b/2$ elements of the *RefSet*.

A straightforward question arises from the last paragraph: *how do we identify the parent of a generated solution*? As explained in Section 2.1.3, the new solutions are created in hyper-rectangles

defined by the pair of solutions combined (see Figure 10). The parent of a solution will be the *RefSet* solution laying in one of the vertex of the hyper-rectangle in which it has been created. Figure 16 depicts how the *go beyond* strategy works: from a pair of *RefSet* solutions (in red), some new solutions are generated in the corresponding hyper-rectangles. The pink solution is the child whose parent is the *RefSet* solution in the vertex of its hyper-rectangle. Since the child outperforms the parent in quality, a new hyper-rectangle (in yellow) is defined by the distance between the parent and the child. A new solution (in orange) is created in this hyper-rectangle. This new solution becomes the child and the old child (i.e., the pink circle) becomes the parent. Since the new child (orange) outperforms again its parent (pink), the process is repeated, but the size of the new hyper-rectangle created (in green) is doubled because there has been improvement during two consecutive children generation.

As we can see in Figure 16, the new hyper-rectangle contains the global optimum, thus this strategy may locate it in a lower number of iterations than a scatter search without it. Algorithm 4 shows a pseudocode of the *go beyond* strategy procedure.

Algorithm 4 *go beyond* strategy

```
Call the Solution Combination Method
Identify children, x_children, outperforming their parents, x_parent
for i=1 to |x_children| do
  x_ch = x^i_children
  x_pr = x^i_parent
  improvement = 1
  denom = 1
  while f(x_ch) < f(x_pr) do
    Create a new solution, x_child_new, in the rectangle defined by [x_ch − (x_pr − x_ch)/denom, x_ch]
    x_pr = x_ch
    x_ch = x_child_new
    improvement = improvement + 1
    if improvement = 2 then
      denom = denom/2
      improvement = 0
    end if
  end while
end for
```

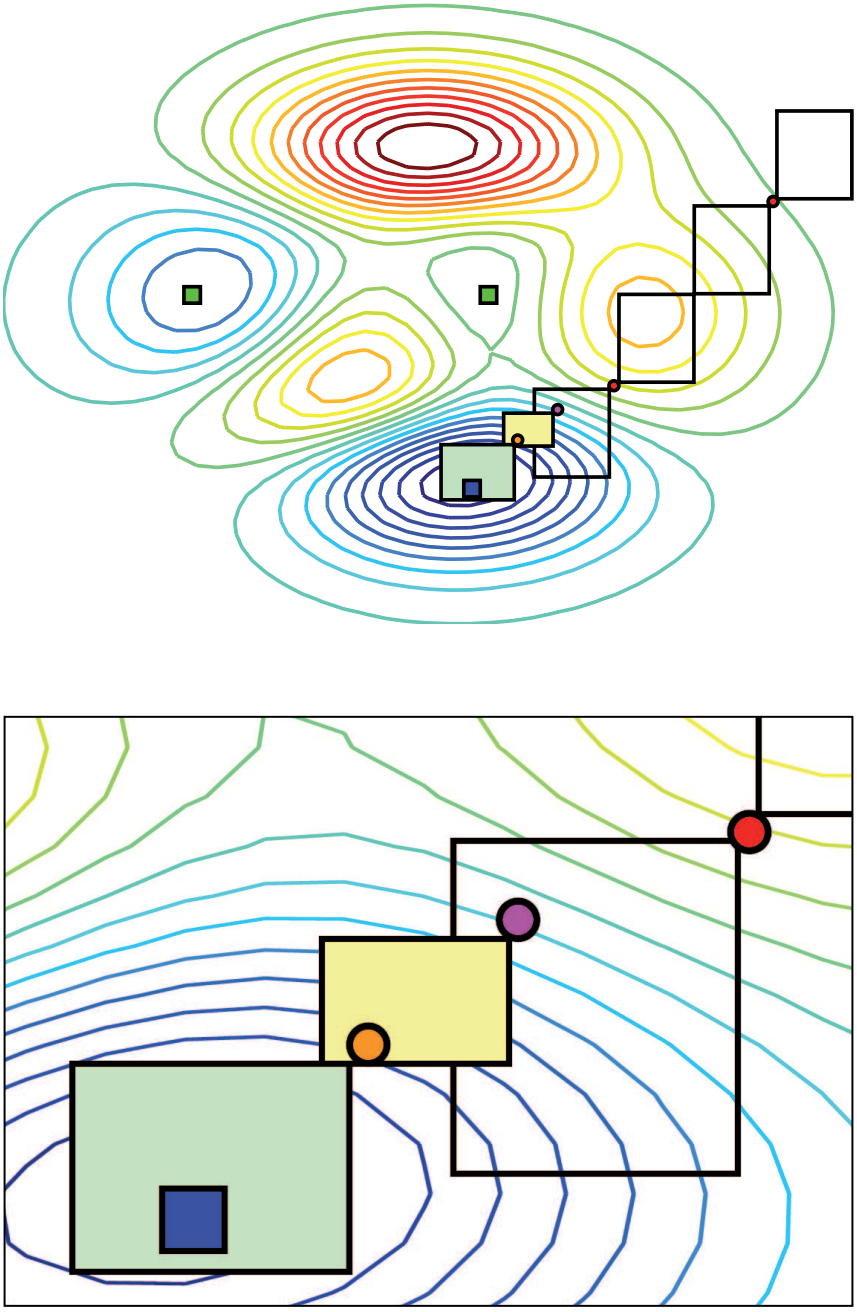
Even if the *go beyond* strategy has been mainly designed to enhance the intensification, the fact that the hyper-rectangles areas are increased if the new solutions improve the old ones during at least two consecutive iterations may make the search be more diverse, exploring regions where different minima can be found.

2.1.9 Constraints handling

Constraint handling of stochastic optimization methods has been subject of research since these algorithms arose. Many different techniques have been proposed to handle problems with constraints (e.g., see reviews in [39] [6], [55]). In our algorithm, we have implemented a simple strategy consisting of a static penalty function. The objective function evaluated by the algorithm has the following form:

$$C(\mathbf{x}) = f(\mathbf{x}) + w \cdot \max\{\max\{viol(\mathbf{h}), viol(\mathbf{g})\}\} \tag{20}$$

where $\mathbf{x}$ is the solution being evaluated, $f(\mathbf{x})$ is the original objective function value, $\mathbf{h}$ is the set of equality constraints and $\mathbf{g}$ is the set of inequality constraints. w is a penalization parameter, which is constant during the optimization procedure (and usually has a high positive value). We use the $L-\infty$ norm of the constraints set to penalize the original objective function. Other penalty function approaches usually use the $L-1$ (exact penalization) or the $L-2$ (quadratic penalization).



(b) (Zoom)

Figure 16: *go beyond* strategy

More sophisticated strategies might be implemented although we strongly rely on the local solvers used by our algorithm to achieve feasible optimal points.

2.1.10 Integer variables handling

Many problems in the chemical and biotechnological industry such as process design, process synthesis and multi-component blended-flow problems, lead to mixed integer nonlinear models [2, 3, 27, 28].

Although our algorithm is mainly designed for continuous problems, a rounding operator has

been implemented for handling integer and binary variables. Glover [22] introduced an operator in scatter search to generate MIP solutions. Here we will use the rounding operator described by Ugray et al. [52]. The *Solution Combination Method* used by our algorithm does not take into account whether a decision variable is integer or not. Thus, a rounding method has to be considered for this kind of variables before evaluating new solutions. For a decision variable x_i with $i \in [1, 2, ..., n]$, we transform it to its closest allowed integer y_i value by:

$$y_i = lb_i + \left[0.5 + \frac{x_i - lb_i}{st_i}\right] st_i \tag{21}$$

where lb_i is the lower bound for the decision variable i, and st_i is the step between two consecutive integer values (usually 1). If the calculated y_i is lower than the upper bound for that decision variable, ub_i, then y_i is accepted as the rounded value. Otherwise we make $y_i = ub_i$.

2.1.11 Stopping criterion

The stopping criterion of our algorithm is taken as a combination of three conditions:

- Maximum number of evaluations exceeded.
- Maximum computational time exceeded.
- Value to reach of the cost function satisfied.

By default, the algorithm will stop when any of these conditions is satisfied. Since the *Improvement Method* is selectively applied, when the scatter search algorithm is over, before abandoning the search, we apply the *Improvement Method* to the best solution found so far, just to be sure that it is not skipped, or simply to refine the best solution. In this final local search, the stopping tolerance assigned to the local solver is tighter than in the rest of local searches performed along the optimization procedure.

2.2 Application to benchmark problems

As a first test of our algorithm's performance, it has been applied to a set of well known unconstrained and constrained global optimization problems that have usually been used as benchmark problems in the literature for testing optimization software. Additionally, to check its applicability to mixed-integer optimization a set of this type of problems from the process engineering area has also been considered. For every problem tested in this section, the local solver chosen was *misqp*.

2.2.1 Unconstrained problems

We have tested our algorithm over 40 unconstrained problem of different dimensions. Table 1 provides information about all these problems.

Following the same procedure as in [33], we have defined an optimality gap as:

$$GAP = |f(x) - f(x^*)| \tag{22}$$

where x is a heuristic solution and x^* is the optimal solution. We say that a heuristic solution is satisfactory if:

$$GAP \leq \begin{cases} \varepsilon & \text{if} \quad f(x^*) = 0 \\ \varepsilon\,|f(x^*)| & \text{if} \quad f(x^*) \neq 0 \end{cases} \tag{23}$$

Number of variables	Problem Number	Problem Name	x^*	$f(x^*)$
2	1	Branin	(9.42478, 2.475)[a]	0.397887
	2	B2	(0, 0)	0
	3	Easom	(π, π)	-1
	4	Goldstein and Price	(0, -1)	3
	5	Shubert	(-7.7083, -7.0835)[a]	-186.7309
	6	Beale	(3, 0.5)	0
	7	Booth	(1, 3)	0
	8	Matyas	(0, 0)	0
	9	SixHumpCamelback	(0.089840, -0.712659)[a]	-1.03163
	10	Schwefel(2)	(420.9687, 420.9687)	-837.9658
	11	Rosenbrock(2)	(1, 1)	0
	12	Zakharov(2)	(0, 0)	0
3	13	De Joung	(0, 0, 0)	0
	14	Hartmann(3,4)	(0.114614, 0.555649, 0.852547)	-3.86278
4	15	Colville	(1, 1, 1, 1)	0
	16	Shekel(5)	(4, 4, 4, 4)	-10.1532
	17	Shekel(7)	(4, 4, 4, 4)	-10.4029
	18	Shekel(10)	(4, 4, 4, 4)	-10.5364
	19	Perm(4,0.5)	(1, 2, 3, 4)	0
	20	Perm0(4,10)	(1, 1/2, 1/3, 1/4)	0
	21	Powersum	(1, 2, 2, 3)	0
6	22	Hartmann(6,4)	(0.20169, 0.150011, 0.47687, 0.275332, 0.311652, 0.6573)	-3.32237
	23	Schwefel(6)	(420.9687,. . . , 420.9687)	-2513.897
	24	Trid(6)	$x_i = i * (7 - i)$	-50
10	25	Trid(10)	$x_i = i * (11 - i)$	-210
	26	Rastrigin(10)	(0,. . . , 0)	0
	27	Griewank(10)	(0,. . . , 0)	0
	28	Sum Squares(10)	(0,. . . , 0)	0
	29	Rosenbrock(10)	(1,. . . , 1)	0
	30	Zakharov(10)	(0,. . . , 0)	0
20	31	Rastrigin(20)	(0,. . . , 0)	0
	32	Griewank(20)	(0,. . . , 0)	0
	33	Sum Squares(20)	(0,. . . , 0)	0
	34	Rosenbrock(20)	(1,. . . , 1)	0
	35	Zakharov(20)	(0,. . . , 0)	0
>20	36	Powell(24)	(3, -1, 0, 1, 3,. . . , 3, -1, 0, 1)	0
	37	Dixon and Price(25)	$x_i = 2^{-\frac{z-1}{z}}$, $z = 2^{i-1}$	0
	38	Levy(30)	(1,. . . , 1)	0
	39	Sphere(30)	(0,. . . , 0)	0
	40	Ackley(30)	(0,. . . , 0)	0

[a]This is one of several multiple optimal solutions.

Table 1: Unconstrained test problems

We set $\varepsilon = 0.0001$. For each test function we perform 30 independent runs with a maximum number of function evaluations of 10^6. In any case, the optimization stops before achieving the maximum number of function evaluations if a satisfactory heuristic solution is found. The results obtained for this set of unconstrained problems are shown in Table 2.

Our algorithm was able to find satisfactory solutions for most of the problems with a high probability. However, it was not able to find good solutions for problem 31 in none of the 30 runs. For problem 26 (which is actually the same as problem 31 but in lower dimension: the *Rastrigin* function) only one run out of 30 was successful. Changing the type of local search we were able to solve these two problems. Instead of a gradient-based algorithm as *misqp*, a direct search method (*NOMADm*) was used. This algorithm may be able to overcome small local minima in the surroundings of the global optimum, thus increasing the probability of finding satisfactory solutions.

Problem Number	$f(x^*)$	Results with SSm				
		Best	**Mean**	**Worst**	**% Success**	**Mean Evaluations**
1	0.397887	0.397887	0.397889	0.397896	100	236
2	0	1.10693e-009	5.11987e-006	7.90261e-005	100	3366
3	-1	-1	-9.99994e-001	-9.99935e-001	100	3949
4	3	3	3	3.00001	100	258
5	-186.7309	-186.7309	-186.7309	-186.7309	100	300
6	0	1.87494e-008	3.60511e-006	4.08401e-005	100	247
7	0	7.57705e-012	1.76946e-006	8.62196e-006	100	245
8	0	2.45572e-008	1.70451e-005	7.61722e-005	100	273
9	-1.03163	-1.03163	-1.03163	-1.03162	100	246
10	-837.9658	-837.9658	-837.9623	-837.9326	100	683
11	0	1.44532e-008	2.69161e-006	1.05570e-005	100	278
12	0	3.64625e-012	1.60548e-006	7.99405e-006	100	256
13	0	1.09026e-018	1.08278e-006	7.57465e-006	100	340
14	-3.86278	-3.86278	-3.86277	-3.86250	100	367
15	0	2.03938e-007	3.03223e-006	9.55533e-006	100	608
16	-10.1532	-10.1532	-10.1532	-10.1532	100	586
17	-10.4029	-10.4029	-10.4029	-10.4029	100	649
18	-10.5364	-10.5364	-10.5364	-10.5364	100	649
19	0	2.46504e-006	5.92432e-003	1.30655e-001	37	635689
20	0	7.20106e-007	2.63873e-005	9.61754e-005	100	4097
21	0	6.55970e-007	2.18288e-005	9.71297e-005	100	6118
22	-3.32237	-3.32237	-3.31044	-3.20316	90	132328
23	-2513.897	-2513.897	-2458.564	-2277.021	60	408437
24	-50	-50	-50	-50	100	752
25	-210	-210	-210	-210	100	1223
26	0	2.85055e-006	2.45423	6.96471	3	969903
26[a]	0	2.57876e-005	4.01953e-005	5.11601e-005	100	2943
27	0	1.60214e-006	1.16388e-005	2.80477e-005	100	18434
28	0	9.77524e-007	3.83435e-006	7.64475e-006	100	1381
29	0	1.06683e-006	1.06683e-006	1.06683e-006	100	2089
30	0	2.24968e-007	3.020977e-006	7.94442e-006	100	1248
31	0	3.97984	7.528523	15.9193	0	1000140
31[a]	0	4.51242e-005	7.08546e-005	9.79561e-005	100	10432
32	0	6.44905e-006	2.04312e-005	4.08787e-005	100	2753
33	0	1.00472e-006	3.88733e-006	6.90130e-006	100	2934
34	0	5.22079e-007	5.22079e-007	5.22079e-007	100	5573
35	0	2.31524e-008	3.49472e-006	9.00947e-006	100	2466
36	0	9.72116e-006	2.67765e-005	5.45857e-005	100	3772
37	0	1.72965e-006	5.55556e-001	6.66667e-001	17	835211
38	0	1.09245e-007	5.71577e-006	4.01198e-005	100	84167
39	0	5.47119e-015	5.647805e-007	1.50361e-006	100	3341
40	0	8.23867e-006	1.215999e-005	1.81072e-005	100	172538

[a]Solution found using a direct local search ($NOMADm$).

Table 2: Unconstrained test problems results

2.2.2 Constrained problems

The next set of problems used for testing our algorithm's performance is a collection of constrained problems usually used for testing new optimization software. Table 3 provides information about all these problems.

As for unconstrained problems, we perform 30 independent runs for each problems with the default parameter values of our algorithm. In this case, we fix a maximum number function evaluations of 100000 in order to compare our results with those presented by Landa-Becerra and Coello [34]. In their study, these authors compare the results obtained with a *Cultured Differential Evolution, CDE*, applied over the same set of problems. They outperform other optimization algorithms and carry out 30 independent runs per problem with a limit of 10^5 function evaluations each run. Table 4 shows our results compared with those reported by these authors.

Results obtained by our algorithm are competitive compared with those obtained by Landa-Becerra

Problem Name	Number of variables	Number of inequality constraints	Number of equality constraints	$f(x^*)$
g01	13	9	0	-15
g02	20	2	0	-0.803619
g03	10	0	1	-1
g04	5	6	0	-30665.54
g05	4	2	3	5126.489
g06	2	2	0	-6961.814
g07	10	8	0	24.30621
g08	2	2	0	-0.095825
g09	7	4	0	680.6301
g10	8	6	0	7049.25
g11	2	0	1	0.75
g12	3	729	0	-1
g13	5	0	3	0.0539498

Table 3: Constrained test problems

Problem Name	$f(x^*)$	Results with *SSm*		Results with *CDE*	
		Best	**Mean**	**Best**	**Mean**
g01	-15	-15.00001	-14.66668	-15.00000	-15.00000
g02	-0.803619	-0.794662	-0.699783	-0.803619	-0.724886
g03	-1	-1.000049	-1.000034	-0.995413	-0.788635
g04	-30665.54	-30665.55	-30665.54	-30665.54	-30665.54
g05	5126.489	5126.498	5126.498	5126.571	5207.411
g06	-6961.814	-6961.821	-6961.815	-6961.814	-6961.814
g07	24.30621	24.30621	24.30621	24.30621	24.30621
g08	-0.095825	-0.095825	-0.095825	-0.095825	-0.095825
g09	680.6301	680.6300	680.6300	680.6301	680.6301
g10	7049.25	7049.25	7049.25	7049.25	7049.25
g11	0.75	0.749990	0.749991	0.749900	0.757995
g12	-1	-1.000000	-1.000000	-1.000000	-1.000000
g13	0.0539498	0.0539495	0.143760	0.056180	0.288324

Table 4: Results for constrained problems, comparing with *CDE*

and Coello [34], which, at the same time, outperformed other state-of-the-art algorithms. Except in *g02* and the mean value of *g01*, our results are the same quality or even better (see *g05* and specially *g13*) than those reported by these authors, which indicates that our algorithm is competitive for constrained problems. Note that in some problems (e.g., *g03* and *g06*) our algorithm reports better values than the global optimum. This is because we allow by default a maximum constraint violation of 10^{-5} in the global phase. Besides, the local solver has its own tolerance which may allow a slight violation of constraints too.

2.2.3 Mixed-integer problems

To finish our algorithm's testing we have selected a set of constrained mixed-integer optimization problems arising from chemical engineering. The models used are written in AMPL code and can be found in Sven Leyffer's web page[†]. Table 5 shows information about this set of problems. Since, in some cases, the mathematical models describing the problems are quite large, they are not included here. We instead provide the original reference for each problem, where the model equations can be found.

Problems 1-5 names use the same notation as in [14]. Problems 6-13 names were taken from [35] and from the same author's web page. For this set of mixed-integer problem we followed the same procedure as in Section 2.2.1. The same optimality *GAP* is defined and 30 independent runs with a maximum number of 10^6 function evaluations were fixed. The results obtained for this set of mixed-integer problems are shown in Table 6.

[†]http://www-unix.mcs.anl.gov/∼leyffer/macminlp/

Problem Number	**Problem Name**	**Ref.**	ncv	niv	nbv	$f(x^*)$
1	mitp4	[4]	1	3	0	-40.957
2	mitp6	[4]	3	4	0	694.9
3	mitp8	[4]	4	6	0	37.219
4	mitp47	[29]	2	0	3	7.6672
5	mitp49	[56]	3	0	4	4.5796
6	windfac	[41]	11	3	0	0.254487
7	synthes1	[11]	5	0	3	6.00976
8	synthes2	[11]	6	0	5	73.0353
9	synthes3	[11]	9	0	8	68.0097
10	optprloc	[11]	5	0	25	-8.06414
11	trimloss2	[23]	6	0	31	5.3
12	batch	[29]	22	0	24	285507
13	spring	[48]	5	1	11	0.846246

ncv = Number of continuous variables

niv = Number of integer variables

nbv = Number of binary variables

Table 5: Mixed-integer test problems

As shown in Table 6, our algorithm is able to solve this set of mixed-integer benchmark problems. Due to some errors in the dynamic library function calling problem 11, only 20 optimizations instead of 30 were performed for it. Problem 13 was the hardest to be solved. Increasing the frequency of the local search call (i.e., making the search more aggressive), we achieved a higher percentage of success. Note that in problems 6-8 the best value obtained by our algorithm is better than the global solution. This is caused again by the default small constraint violation allowed by our algorithm.

With this set of benchmark problems we finish the testing of our algorithm performance with benchmark functions. We were able to solve all the proposed problem with high probability of success. Default parameter values were used for the test, although in some cases special settings had to be considered to increase the success rate.

Problem Number	$f(x^*)$	Results with SSm				
		Best	**Mean**	**Worst**	**% Success**	**Mean Evaluations**
1	-40.957	-40.957	-40.957	-40.957	100	541
2	694.9	694.9	694.9	694.9	100	1043
3	37.219	37.219	37.219	37.219	100	1467
4	7.6672	7.6672	7.6672	7.6672	100	53019
5	4.5796	4.5796	4.5796	4.5796	100	843
6	0.254487	0.254484	0.254487	0.254487	100	57954
7	6.00976	6.00972	6.00975	6.00976	100	1345
8	73.0353	73.0351	73.0353	73.0353	100	4188
9	68.0097	68.0097	68.0097	68.0097	100	14925
10	-8.06414	-8.06414	-8.06414	-8.06414	100	12359
11[a]	5.3	5.3	5.3	5.3	100	30639
12	285507	285507	285507	285507	100	46526
13	0.846246	0.846246	0.940063	1.82256	53	575033
13[b]	0.846246	0.84624	0.847983	0.859276	87	202028

[a]Results in 20 runs

[b]Increasing the local search frequency

Table 6: Mixed-integer test problems results

References

[1] M. A. Abramson. *Pattern Search Algorithms for Mixed Variable General Constrained Optimization Problems*. PhD thesis, Rice University, 2002.

[2] C. S. Adjiman, I. P. Androulakis, and C. A. Floudas. Global optimization of MINLP problems in process synthesis and design. *Computers and Chemical Engineering*, 21(SUPPL.1):S445–S450, 1997.

[3] C. S. Adjiman, I. P. Androulakis, and C. A. Floudas. Global optimization of mixed-integer nonlinear problems. *AIChE Journal*, 46(9):1769–1797, 2000.

[4] J. Asaadi. A computational comparison of some non-linear programs. *Mathematical Programming*, 4(1):144–154, 1973.

[5] J. R. Banga, C. G. Moles, and A. A. Alonso. Global optimization of bioprocesses using stochastic and hybrid methods. In C. A. Floudas and P. M. Pardalos, editors, *Frontiers In Global Optimization*, volume 74 of *Nonconvex Optimization and Its Applications*, pages 45–70. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[6] C. A. Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, 2002.

[7] T. F. Coleman and Y. Li. On the convergence of interior-reflective newton methods for nonlinear minimization subject to bounds. *Mathematical Programming*, 67(1-3):189–224, 1994.

[8] M. de la Maza and D. Yuret. Dynamic hill climbing. *AI Expert*, 9(3):26–31, 1994.

[9] J. E. Dennis, D. M. Gay, and R. E. Welsch. An adaptive non-linear least-squares algorithm. *ACM Transactions on Mathematical Software*, 7(3):348–368, 1981.

[10] J. E. J. Dennis. Nonlinear least squares and equations. In D. Jacobs, editor, *State of the Art in Numerical Analysis*, pages 269–312. Academic Press, 1977.

[11] M. A. Duran and I. E. Grossmann. Outer-approximatino algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36(3):307–339, 1986.

[12] J. A. Egea, M. Rodríguez-Fernández, J. R. Banga, and R. Martí. Scatter search for chemical and bio-process optimization. *Journal of Global Optimization*, 37(3):481–503, 2007.

[13] J. A. Egea, E. Vazquez, J. R. Banga, and R. Martí. Improved scatter search for the global optimization of computationally expensive dynamic models. *Journal of Global Optimization*, In press(doi:10.1007/s10898-007-9172-y), 2007.

[14] O. Exler and K. Schittkowski. MISQP: A fortran implementation of a trust region SQP algorithm for mixed-integer nonlinear programming - user's guide, version 2.1-. Technical report, Department of Computer Science University of Bayreuth, Bayreuth, Germany, 2006.

[15] O. Exler and K. Schittkowski. A trust region SQP algorithm for mixed-integer nonlinear programming. *Optimization Letters*, 1(3):269–280, 2007.

[16] C. Fleurent, F. Glover, P. Michelon, and Z. Valli. Scatter search approach for unconstrained continuous optimization. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 643–648, 1996.

[17] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[18] F. Glover. Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). *Discrete Applied Mathematics*, 49(1-3):231–255, 1994.

[19] F. Glover. A template for scatter search and path relinking. In J. K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume 1363 of *Lecture Notes in Computer Science*, pages 13–54. Springer Verlag, 1998.

[20] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):652–684, 2000.

[21] F. Glover, M. Laguna, and R. Martí. Scatter search. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computation: Theory and Applications*, pages 519–537. Springer-Verlag, New York, 2003.

[22] F. Glover, A. Løkketangen, and D. L. Woodruff. Scatter search to generate diverse MIP solutions. In M. Laguna and J. L. González Velarde, editors, *Computing tools for modeling optimization and simulation*, pages 299–320. Kluwer Academic Publishers, Boston, 2000.

[23] I. Harjunkoski, T. Westerlund, R. Pörn, and H. Skrifvars. Different transformations for solving non-convex trim-loss problems by MINLP. *European Journal of Operational Research*, 105(3):594–603, 1998.

[24] F. Herrera and M. Lozano. Gradual distributed real-coded genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 4(1):43–62, 2000.

[25] F. Herrera, M. Lozano, and D. Molina. Continuous scatter search: An analysis of the integration of some combination methods and improvement strategies. *European Journal of Operational Research*, 169(2):450–476, 2006.

[26] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. 1961.

[27] J. Kallrath. Mixed integer optimization in the chemical process industry: Experience, potential and future perspectives. *Chemical Engineering Research and Design*, 78(6):809–822, 2000.

[28] J. Kallrath. Solving planning and design problems in the process industry using mixed integer and global optimization. *Annals of Operations Research*, 140(1):339–373, 2005.

[29] G. R. Kocis and I. E. Grossmann. Global optimization of nonconvex mixed-integer nonlinear programming (MINLP) problems in process synthesis. *Industrial and Engineering Chemistry Research*, 27(8):1407–1421, 1988.

[30] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the nelder-mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1999.

[31] M. Laguna and R. Martí. The OptQuest callable library. In S. Voss and W. D. L., editors, *Optimization Software Class Libraries*. Kluwer Academic Publishers, Boston, 2002.

[32] M. Laguna and R. Martí. *Scatter Search: Methodology and Implementations in C*. Kluwer Academic Publishers, Boston, 2003.

[33] M. Laguna and R. Martí. Experimental testing of advanced scatter search designs for global optimization of multimodal functions. *Journal of Global Optimization*, 33(2):235–255, 2005.

[34] R. Landa Becerra and C. A. Coello-Coello. Cultured differential evolution for constrained optimization. *Computer Methods in Applied Mechanics and Engineering*, 195(33-36):4303–4322, 2006.

[35] S. Leyffer. Integrating SQP and branch-and-bound for mixed integer nonlinear programming. *Computational Optimization and Applications*, 18(3):295, 309 2001.

[36] M. Lozano, F. Herrera, N. Krasnogor, and D. Molina. Real-coded memetic algorithms with crossover hill-climbing. *Evolutionary Computation*, 12(3):273–302, 2004.

[37] R. Martí and M. Laguna. Scatter search: Diseño básico y estrategias avanzadas. *Revista Iberoamericana de Inteligencia Artificial*, 19:123–130, 2003.

[38] R. Martí, M. Laguna, and F. Glover. Principles of scatter search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[39] Z. Michalewicz. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[40] Z. Michalewicz, T. Logan, and S. Swaminathan. Evolutionary operators for continuous convex parameter spaces. *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 84–97, 1994.

[41] M. Michna. Model of the winding factor of the electrical machines. Technical report, Polytechnika Gdanska, 2000.

[42] D. Molina, F. Herrera, and M. Lozano. Adaptive local search parameters for real-coded memetic algorithms. In *IEEE Congress on Evolutionary Computation, IEEE CEC 2005*, volume 1, pages 888–895, 2005.

[43] J. J. Moré. The levenberg-marquardt algorithm: implementation and theory. In G. A. Watson, editor, *Numerical Analysis*, number 630 in Lecture Notes in Mathematics, pages 105–116. Springer-Verlag, Berlin, 1978.

[44] A. Neumaier, O. Shcherbina, W. Huyer, and T. Vinkó. A comparison of complete global optimization solvers. *Mathematical Programming*, 103(2), 2005.

[45] E. Panier and A. L. Tits. On combining feasibility, descent and superlinear convergence in inequality constrained optimization. *Mathematical Programming*, 59:261–276, 1993.

[46] M. Rodríguez-Fernández, E. Balsa-Canto, J. A. Egea, and J. R. Banga. Identifiability and robust parameter estimation in food process modeling: Application to a drying model. *Journal of Food Engineering*, 83(3):374–383, 2007.

[47] M. Rodríguez-Fernández, J. A. Egea, and J. R. Banga. Novel metaheuristic for parameter estimation in nonlinear dynamic biological systems. *BMC Bioinformatics*, 7:483+, 2006.

[48] E. Sandgren. Nonlinear integer and discrete programming in mechnical design optimization. *Journal of Mechanisms, Transmissions, and Automation in Design*, 112(2):223–229, 1990.

[49] W. Tfaili and P. Siarry. A new charged ant colony algorithm for continuous dynamic optimization. *Applied Mathematics and Computation, In press, Corrected Proof*, 2008.

[50] T. B. Trafalis and S. Kasap. An affine scaling scatter search approach for continuous global optimization problems. In C. H. Dagli, M. Akay, C. L. P. Chen, B. R. Fernandez, and J. Ghosh, editors, *Intelligent Engineering Systems Through Artificial Neural Networks*, volume 6, pages 1027–1032. ASME Press, 1996.

[51] T. B. Trafalis and S. Kasap. A novel metaheuristics approach for continuous global optimization. *Journal of Global Optimization*, 23(2):171–190, 2002.

[52] Z. Ugray, L. Lasdon, J. Plummer, F. Glover, J. Kelly, and R. Martí. A multistart scatter search heuristic for smooth NLP and MINLP problems. In C. Rego and B. Alidaee, editors, *Adaptive Memory and Evolution: Tabu Search and Scatter Search*, pages 25–58. Kluwer Academic Publishers, 2005.

[53] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[54] Y. Ye. *Interior algorithms for linear, quadratic and linearly constrained non-linear programming*. PhD thesis, Stanford University, 1987.

[55] O. Yeniay. Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and Computational Applications*, 10(1):45–56, 2005.

[56] X. Yuan, S. Zhang, L. Piboleau, and S. Domenech. Une methode d'optimization nonlineare en variables mixtes pour la conception de procedes. *RAIRO*, 22:31, 1988.