Sequential Parameter Optimization

Thomas Bartz–Beielstein
Systems Analysis Group
Department of Computer Science
University of Dortmund, Germany
thomas.bartz-beielstein@udo.edu

Christian W. G. Lasarczyk
Systems Analysis Group
Department of Computer Science
University of Dortmund, Germany
christian.lasarczyk@udo.edu

Mike Preuss
Systems Analysis Group
Department of Computer Science
University of Dortmund, Germany
mike.preuss@uni-dortmund.de

Abstract- Sequential parameter optimization is a heuristic that combines classical and modern statistical techniques to improve the performance of search algorithms. To demonstrate its flexibility, three scenarios are discussed: (1) no experience how to choose the parameter setting of an algorithm is available, (2) a comparison with other algorithms is needed, and (3) an optimization algorithm has to be applied effectively and efficiently to a complex real-world optimization problem. Although sequential parameter optimization relies on enhanced statistical techniques such as design and analysis of computer experiments, it can be performed algorithmically and requires basically the specification of the relevant algorithm's parameters.

1 Introduction

Modern search heuristics involve a set of parameters that can affect their performance drastically. We propose an approach for determining adequate parameters of optimization algorithms, tailored for the optimization problem at hand. The proposed approach employs a sequential technique from computational statistics and statistical experimental design. *Sequential parameter optimization* (SPO) can even be applied to search algorithms that produce stochastically disturbed results, especially evolutionary algorithms (genetic algorithms, evolution strategies, genetic programming, or newer approaches such as algorithmic chemistries and particle swarm optimization).

To show the universality of this approach we tackle optimization scenarios in which

- newly developed algorithms are applied to a well-known problems (Section 3.1), or
- well-known algorithms are applied to well-known problems (Section 3.2), or
- well-known algorithms are used to optimize unknown, complex real-world optimization problems (Section 3.3).

These scenarios have one thing in common: They are definitively too complex to allow for reliable manual parameter tuning, because the number of different parameter combinations grows exponentially with the number of algorithm parameters—even worse: If quantitative or continuous parameters are considered, as in the following, there are infinitively many combinations.

The major differences between SPO and other approaches that have been proposed to solve these problems, e.g. metaheuristics, can be described as follows: SPO keeps the computational cost for determining an improved parameter set relatively low. Furthermore, the optimization practitioner can learn—but she is not forced to learn—from experiments; the method is semi–automatic, the course of the tuning process is subject to change. And, this is probably the most important difference, SPO can be applied in an algorithmical manner, it requires the specification of very few parameters and no additional programming effort.

Our aims in this paper are

- to demonstrate its usefulness for very different algorithms and optimization tasks and explain how it works (*what*)
- and to shed light on possible SPO use cases, the questions it can help to answer, and the expected results (*why*).

The paper is structured as follows: Section 2 introduces sequential parameter optimization. Next, we present three application scenarios (Sect. 3). The paper closes with a discussion and an outlook.

2 Sequential Parameter Optimization

Designs are the key elements of an algorithmic SPO description, thus we provide some definitions first. An *algorithm design* $\mathcal{D}_A$ is specified by defining ranges of values for the design variables. A design point $x \in \mathcal{D}_A$ is a vector with specific settings of the considered algorithm, see e.g. Tab. 1. We will consider quantitative factors only. How qualitative factors can be included into the experimental analysis is discussed in [1, 2]. *Problem designs* $\mathcal{D}_P$ provide information related to the optimization problem, such as the available resources, e.g. the number of function evaluations $t_{\max}$, or initialization and termination criteria. An *experimental design* $\mathcal{D}$ consists of a problem design $\mathcal{D}_P$ and an algorithm design $\mathcal{D}_A$. We regard the run of a stochastic search algorithm as an experiment with stochastic output $Y(x)$, with $x \in \mathcal{D}_A$ for one problem design only. If random seeds are specified, the output would be deterministic. This case will not be considered further, because in practice it is not common to specify the seed that is used in an optimization run.

SPO can be interpreted as a search heuristic optimizing the performance of non–deterministic algorithms. Therefore it includes methods to cope with stochastically disturbed results, while keeping the number of required steps (algorithm runs) as low as possible. In most cases, time constraints rule out grid search, exhaustive local search methods, or even (evolutionary) meta–optimization algorithms: This is where

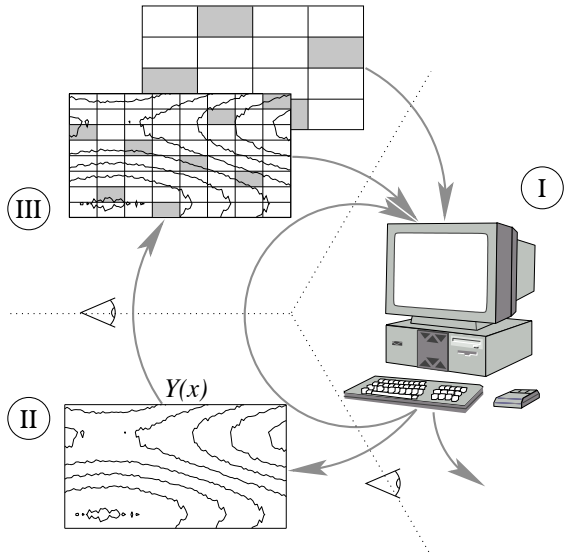

Figure 1: *SPO consists of 3 iterative stages. The first stage determines design points. Latin hypercube sampling and model based selection is used to chose new points. The best point found so far is reevaluated. During the second stage designs points are evaluated by running the algorithm using the proposed setting. During the third stage a model is build, to estimate the algorithms performance for untested settings. Results of second and third stage may be of interest to the user. As experimental results are obviously interesting for the user, also the model could show insightful properties of parameters and their relation.*

SPO may help.

Figure 1 illustrates the interaction of the three key components of SPO:

I Experimental analysis of a set of design points

II Estimation of algorithm performance my means of a stochastic process model, and

III Determination of additional design points.

These components will be discussed in the following. Assume, that an initial set of design points $\mathcal{I}$ has been determined already (how to determine $\mathcal{I}$ will be described in Section 2.3.1).

2.1 Experimental Evaluation

Experiments are performed to estimate the performance of a design point $x \in \mathcal{D}_A$ of an algorithm design for a given problem design. Let $\mathcal{I}(t)$ denote the set of design points to evaluate at sequential step t and $x^*(t)$ be the best design point at sequential step t. Due to the stochastic nature of the search algorithms considered here, performance for each design point is evaluated by means of several repeats. The best design point from the previous iteration $x^*(t-1)$ is included in the set $\mathcal{I}(t)$ and re–evaluated, thereby doubling its number of repeats.

SPO enforces fair comparison to the current best design point; newly generated design points are tested as often as the best design point has been. Incrementing the number of repeats by factor 2 each iteration gradually decreases the estimation error between the mean of measured and the true function value of a design point x.

2.2 Modeling

Following [3], the response is modeled as a realization of a regression model and a random process. A Gaussian correlation function and a regression model with polynomial of order 2 have been used. Hence, the model reads

$$Y(x) = \sum_{j=1}^{p} \beta_j f_j(x) + Z(x), \tag{1}$$

where $Z(\cdot)$ is a random process with mean zero and covariance $V(u,v) = \sigma^2 \mathcal{R}(\theta, u, v)$, and the correlation function was chosen as

$$\mathcal{R}(\theta, u, v) \prod_{j=1}^{d} \exp\left(-\theta_j (u_j - v_j)^2\right).$$

2.3 Design Point Determination

We use Latin Hypercube Sampling (LHS) [4] to determine new design points. Such points could be calculated systematically for every number k of desired design points and every dimension size d. Additionally this approach enables a good coverage of the design space.

To determine k design points we have to divide range of each dimension into k equally sized intervals. For each design point we map a range in each dimension one–to–one and draw a random value within this range. If parameter assigned to this dimension requires values to be in $\mathbb{N}$ rounding happens afterwards.

2.3.1 Initial and Sequential Design Points

The initial design points are chosen by a LHS design as described above. The number of parameters in the stochastic process model from Equation 1 determines the minimal number of design points. The set of design point in sequential steps comprehends the best point found so far and a set of expected good designs, whereby expectation is based on the model created in previous SPO step.

Therefore we first create an additional set of design point candidates. This set is typically much larger than initial set of design points. Computing the candidates model value, the generalized expected improvement criterion is determined next [5]. In short, this criterion takes into account, that we are uncertain about unknown design points and estimates probability of a candidate of being better than the known best so far, by taking the modeling error into account. So we select those design point candidates that possess the highest probability of being better than the best.

By choosing an appropriate number of design points candidates exploration and exploitation can be balanced. If the number of candidate design points is too small, the model based exploration of the search space is not satisfying because many parameter combinations remain unconsidered.

Algorithm specific parameters

- parameters: range & type
- optimization criterion

SPO specific parameters

- initial design: size & samples per design point
- sequential step:
  - number of candidates (LHS)
  - number of new and old selected design points
- termination criterion

Figure 2: *Parameters required to optimize an algorithm using sequential parameter optimization.*

On the other hand, if number of design point candidates is very high, exploration is quite good, but as selection happens, candidates selected for real experimentation may be next to each other which can lead to a "waste" of experimentation effort on a small region just expected but not ensured to be good.

2.4 Configuration

We distinguish two aspects of configuration. The first aspect considers settings of the algorithm to be optimized, the second aspect considers settings of the SPO approach. The algorithm interface requires the specification of an algorithm design as shown in the first 4 columns of Tab. 1. For each parameter the user has to specify minimal and maximal values and a parameter type. SPO allows quantitative factors only, because it uses space filling designs and Kriging methods. By rounding decimals we additionally allow integer values. Rounding happens right after creating a new design point using a space filling design, so prediction, modeling, and execution is based on rounded values.

In addition to these values, we have to specify a performance criterion, e.g. the average fitness of best individual or worst fitness of best individual on optimizing an online algorithm. Our SPO implementation is based on the DACE toolbox developed by Lophaven et al. [6] and extends heuristics (e.g. the expected improvement) proposed by Schonlau [7].

Furthermore, there are some parameters needed by the sequential approach. The number of initial design points and the number of samples per point can be mentioned here as well as the number of design points that will be evaluated in each sequential step. The determination of the latter is based on the stochastic process model introduced in Eq.(1) and the number of best evaluated points out of those we choose for real experimentation. Additionally we have to specify the number of best so far design point we want to run further experiments in the next sequential step. Finally, a termination criterion has to be selected, e.g. a number of sequential steps or number of repetitions for the best setting. Figure 2 lists required configuration parameters.

3 Application

3.1 Algorithmic Chemistries

3.1.1 Algorithm

Algorithmic chemistries are *artificial chemistries* that aim at algorithms. An algorithmic chemistry is a multiset of instructions, which consist of an opcode and three addresses targeting sources and destination register. Instructions cannot be accessed in a specific order. To execute an algorithmic chemistry they are drawn randomly and executed in an environment containing a common set of registers. We distinguish two kinds of registers. Instructions interact with each other via connection registers, by reading results of other instructions and feeding their computation into registers possibly read by one or more other instruction. A second, readonly set of registers contains input values. At the end of execution a predefined connection register is interpreted as the chemistry's output.

While different multisets are considered as different programs, different executions of a single program show different behavioral variants due to random execution order. Concentration of instructions mainly influences variant's probability. If two instructions share the same register as their target register and one of both is necessary to compute a desired value, chemistries can increase their chance for successful execution by increasing the frequency of the required instruction or by decreasing/removing the not required one.

We use *genetic programming* (GP) to adjust these concentrations by selecting those chemistries that show preferable variants. An GP individual consist of an algorithmic chemistry and the address of the register interpreted as the individual's output. Individuals are initialized by generating a random set of valid instructions.

As we use a (μ, λ)–strategy, a set of μ parents produce λ offspring. Crossover rate determines the proportion of offsprings generated by recombination. To recombine two individuals, the offspring's chemistry is formed by a random subset of both parents chemistries and the address of the result register is randomly take by on of both parents. Remaining offspring are generated by reproducing a random parent.

Before offspring are evaluated, they get mutated. With a probability called mutation rate we change each entity under evolution independent from one another by setting it to random, valid value.

Evaluation happens by executing an algorithmic chemistry on a set of fitness cases. Thereby the randomized way of execution introduces a new source of noise. To reduce noise we simply increase the set of fitness cases used for evaluation. If number of fitness cases is small, we increase training set by duplication. Apparently this increases the computational costs of evaluation, so we want to keep the number of fitness cases as low as possible.

For a detailed description of algorithmic chemistries see [8, 9].

Table 1: *Algorithm design for genetic programming of algorithmic chemistry solving the 4–Bit odd parity problem.*

Parameter	$\mathbb{N}/\mathbb{R}$	Min	Max	4–Bit
Population size μ	$\mathbb{N}$	1000	5000	4651
Selection pressure (λ/μ)	$\mathbb{R}$	2	20	10.8
Crossover rate	$\mathbb{R}$	0%	100%	2.6%
Mutation rate	$\mathbb{R}$	0.1%	10%	2.6%
Initial length	$\mathbb{N}$	1	50	29
Connection registers	$\mathbb{N}$	10	50	28
Training size	$\mathbb{N}$	16	256	117

3.1.2 Design Considerations

The first time we used GP to evolve algorithmic chemistries we tried to adopt as many settings as possible from a linear GP systems. As random execution of instructions differs much from executing linear individuals, we soon felt uncomfortable with simply adopting settings.

So we tried a to improve performance by adjusting settings more or less systematically. In [8] we use an iterative latin hypercube sampling designs to adjust parameters. In each iteration we narrow the parameters ranges by examining the results manually. Obviously its difficult to grasp complex interactions between parameters this way. As problems became more difficult, ranges of good settings decreases and importance of parameter interactions increases, so iterative latin hypercube designs where not suitable anymore. Starting with [9] we use SPO to adjust parameter settings.

Table 1 shows parameters, that are part of our algorithmic design and their range of values. The first four parameters are well known from other evolutionary algorithms, so we do not explain in detail, why they belong to out algorithm design. To ensure that design points do not violate the $\mu < \lambda$ constraint, we adjust population size μ and selection pressure (λ/μ) with $\lambda/\mu > 1$. Further we adjust mutation and crossover rate. In following paragraphs, we explain the reason for considering the three other parameters and the optimization criterion we use.

If the initial chemistry size is too small, then the initial diversity is small. If initial size is large, probability of a single instruction to be drawn is low, an individual's number of behavioral variants increases and results are less reproducible. Therefore we design for initial size too.

Obviously we need a minimum number of connection registers to organize chemistry's data flow, but an increasing number of connection registers also increases search space. So its necessary to find an adequate number of connection registers

Training set size determines the number of fitness cases used to evaluate an individual and is a part of our algorithms design. As we terminate runs after a limited number of executed instruction, there is an implicit pressure for smaller training set sizes. While smaller sets increase noise, they also decrease the number of executed instruction per generation and therefore increase the number of possible generations if number of instruction executions is limited.

The task (problem design) has been to evolve a boolean function returning true iff parity of 4 input bits is odd. Evolving parity functions using operation set {`AND`, `OR`, `NAND`, `NOR`} is known as one of the most difficult boolean functions to evolve. While we first tried to optimize for success rate, it soon turns out, that success are very rare events within the initial designs, unable to guide the further optimization process. Therefore we adjust the problem design by optimizing for the proportion of correctly classified inputs, which leads to 2^4 fitness levels instead of a single one.

In spite of this "understated" optimization criterion SPO has been able to detect very promising designs (probability of success: 80%) for evolving a perfect solution of the 4–bit odd parity problem. The last column in Tab. 1 shows the best setting found during optimization process.

3.2 Classical Test Suites

The previous section described how to apply SPO to newly developed algorithms. Next we report how PSO can be used to improve well–known algorithms.

Two important aspects in optimization will be discussed in this section: (i) effectivity and (ii) efficiency. First, we will demonstrate how SPO can improve the effectivity of algorithms. Effectivity can be characterized as the algorithm's ability to obtain the best goal for a given resource limit (budget), e.g. maximum number of function evaluations $t_{\max}$. Efficiency is the ability to reach a pre-specified goal, e.g. the known best function value, with the least amount of resources.

It is still a common practice in evolutionary computation not to tune the parameters of each heuristic for each problem. Some authors claim that the parameter settings used in their experimental study are based on experience and turned out to yield very satisfying performance results for a broad class of optimization problems (some authors have become more cautious because of "no free lunch").

SPO has the capability to tune even complex optimization algorithms during the pre-experimental planning phase. The choice of an adequate experimental setup—and this includes algorithm parameters that have been chosen with respect to the optimization problem—is as important as a sound statistical interpretation of the experimental results. What is the use of comparing algorithms with parameter settings that are totally inadequate for the underlying problem?

3.2.1 Particle Swarm Optimization

As there are no theoretical results for many evolutionary algorithms availabale, "default" settings from the literature are chosen as starting points. We will analyze the particle swarm optimizer (PSO) [10]. Assume a d-dimensional search space, $S \subset \mathbb{R}^d$, and a swarm consisting of s particles. The i-th particle is a d-dimensional vector,

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T \in S.$$

The velocity of this particle is also a d-dimensional vector,

$$v_i = (v_{i1}, v_{i2}, \ldots, v_{id})^T.$$

Table 2: *Default algorithm design of the PSO algorithm. Similar designs have been used in [12] to optimize well–known benchmark functions. Some authors use $x_{\max}$as an additional parameter, that specifies the search interval in the objective variable space, e.g. $x_i \in [-x_{\max}, x_{\max}]$. Default and tuned values are shown in Tab. 4.*

Parameter	$\mathbb{N}/\mathbb{R}$	Min	Max
Swarm size: s	$\mathbb{N}$	5	100
Cognitive parameter: c_1	$\mathbb{R}_+$	0	4
Social parameter: c_2	$\mathbb{R}_+$	0	4
Starting value of the inertia weight w: $w_{\max}$	$\mathbb{R}_+$	0.5	1
Final value of w in percentage of $w_{\max}$: w_{scale}	$\mathbb{R}_+$	0	1
Percentage of iterations, for which $w_{\max}$ is reduced: w_{iterSc}	$\mathbb{R}_+$	0	1
Velocity, maximum value: $v_{\max}$	$\mathbb{R}_+$	10	1000

The best previous position encountered by the i-th particle (i.e., its memory) in S is denoted by,

$$p_i^* = (p_{i1}^*, p_{i2}^*, \ldots, p_{id}^*)^T \in S.$$

Assume b to be the index of the particle that attained the best previous position among all the particles in the swarm, and t to be the iteration counter. Then, the resulting equations for the manipulation of the swarm are [11],

$$\begin{aligned} v_i(t+1) =& w v_i(t) + c_1 r_1 \left(p_i^*(t) - x_i(t)\right) \\ &+ c_2 r_2 \left(p_b^*(t) - x_i(t)\right), \\ x_i(t+1) =& x_i(t) + v_i(t+1), \end{aligned} \tag{2}$$

where $i = 1, 2, \ldots, s$; w is a parameter called the *inertia weight*; c_1 and c_2 are positive constants, called the *cognitive* and *social* parameter, respectively; and r_1, r_2 are vectors with components uniformly distributed in $[0, 1]$, $x_{\min}$ and $x_{\max}$ define the bounds of the search space, and $v_{\max}$ is a parameter that was introduced in early PSO versions to avoid swarm explosion. Table 2 summarizes the design variables of particle swarm optimization algorithms.

3.2.2 SPO to Improve Effectivity

Our first comparison is based on the results from an experimental study of particle swarm optimization that comprehends four different benchmark problems to show that PSO is "a promising optimization approach"[12]. We report results from the dimension 10, results from other dimensions are similar. The first design comprehends 80 points that have been generated with Latin hypercube sampling. Each design point represents one parameter setting and has been initially evaluated two times. Several evaluations are neccessary because of the stochastic nature of the PSO algorithm. This number can be kept low, because only a few evaluations are necessary to detect worse design points that should not be considered further. A design correction mechnism was included to prevent outliers that disturb the analysis. It ensures that the relation $c_1 + c_2 \leqslant 4.0$ holds. We re–implemented a PSO based on the description given in [12]. Table 3 shows results published in [12] denoted by "Default I", the results from our impementation are denoted by "Default II". Due to PSO's stochastic nature (and the experimental setup) the variances of the experimental data are relatively high.

Table 3: *A comparison of the results published in [12] using default settings(Default I), own experimental results using these settings (Default II), and results from the tuned version of the PSO. Note, that [12] report only four digits after the decimal and no standard deviations. The standard deviation for the results from the Rosenbrock function is 298.3827.*

Problem	Default I	Default II	Tuned
Sphere	0	2.82e-09	1.66e-21
Rosenbrock	96.17	148.84	4.20
Rastrigin	5.56	10.43	0.98
Griewangk	0.09	0.12	0.07

Table 4: *A comparison of the default and the tuned parameter settings. Tuned parameters for the Rosenbrock and Rastrigin function are identical.*

Problem	s	c_1	c_2	$w_{\max}$	w_{scale}	w_{iterSc}	$v_{\max}$
Default	20	2	2	0.9	0.44	1	100
Sphere	29	0.75	2.49	0.73	0.34	0.64	6.64
Rosenbrock	5	0.69	3.32	0.76	0.48	0.85	1.54
Rastrigin	5	0.69	3.32	0.76	0.48	0.85	1.54
Griewangk	32	1.80	2.20	0.86	0.45	0.53	9.66

However, the performance improvements caused by SPO are significant. Are there any explanations for these significant differences between results from default and tuned algorithm designs? Comparing the default and the tuned algorithm designs (Tab. 4), the following observations can be made. Small $c_1$values are preferable, that means that the influence of the local best position should be reduced. The influence of the global best position is rather strong. The default and the tuned $v_{\max}$values show the most striking difference: smaller values are definitively better. This is not a big surprise—it is a consequence of the experimental setup that avoids search positions outside the interval $[-x_{\max}, x_{\max}]$, the algorithm is "forced to be good". Results from this experimental setup are biased and highly questionable. Other designs should be preferred [2]. SPO was able to reduce the variance in the results drastically, therefore the tuned parametrizations are only better but also more robust than the default settings. Compared to other algorithms (evolution strategies or classical optimizers such as Quasi–Newton methods) the performance of the PSO is rather poor. For example, a function value of 4.2 after 10,000 function evaluations for Rosenbrock's function is not efficient. However, sequential parameter optimization can reduce this value as will be demonstrated next.

3.2.3 SPO to Improve Efficiency

SPO is an integrated approach that combines several methods from classical and modern statistics. Results can be used to (i) generate new design points and to improve the algorithm's performance and (ii) to understand how the algorithm works. Results from the previous experiments will be used to generate run–length distributions (RLD). Figure 3 illustrates how SPO improves the efficiency: only 2000 func-

tion evaluations are necessary to nearly complete 100 % of the runs successfully, the default configuration requires more than 10,000 function evaluations to complete less than 80 % of the runs successfully. Imagine an experimental setup

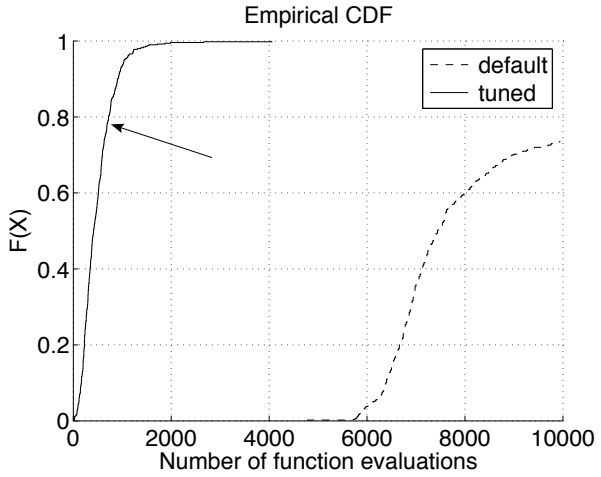

Figure 3: *RLD comparing the default and the tuned PSO on the Rosenbrock function. This is only one example that demonstrates how SPO improves the efficiency of search heuristics. Consider the arrow: 80 % of the runs from the tuned algorithm were able to complete the run successfully after 1000 function evaluations, whereas none of the PSO with the default configuration was able to reach the pre–specified goal with the budget.*

with $t_{\max} = 5000$ function evaluations. Based on the parameterization of the PSO the experimenter can demonstrate, that (a) PSO fails completely, or (b) PSO is able to solve the problem in any case. Both experimental results are statistically sound—however, whether the results are scientifically meaningful is another question. The new experimentalism provides tools to tackle these questions [13].

3.3 Evolutionary Algorithms

Evolutionary algorithms are direct search heuristics and thus easily applied to all kinds of optimization problems. Their enormous flexibility is also one of their weaknesses because it stems from utilization of many different operators that come together with a lot of parameters. However, it is difficult to choose the right operators and/or parameters when problem knowledge is very limited.

In this section, the optimization problem is a simplified real–world problem from the chemical engineering domain: the design of a non–sharp separation sequence. Such a separation process is needed if the available feed stream (raw material) is a mixture of different substances of which only some combinations are acceptable for further processing. A descriptive example in this context is crude oil; it is partitioned into several fractions, e.g. kerosene, gas, and diesel oil. One way to perform separation is by distillation where the feed is heated and pumped into a column (large vertical tube), so that materials with lower/higher boiling point drift to the top/bottom and can be extracted as separate streams. The columns are laid out such that minimum requirements concerning cleanness of the products are fulfilled, these 2 design variables are refered to as light key and heavy key recovery. In previous work, separation has often been considered ideal (sharp), resulting in pure products. However, in many industrial environments, absolute cleanness is not necessary and remixing pure components after first producing them requires very expensive equipment; this is clearly not desirable. We therefore already take the non–sharpness into account during optimization of the separation sequence design. This problem and the appropriate simulation techniques have been provided by the Chair of Technical Thermodynamics of the RWTH Aachen.

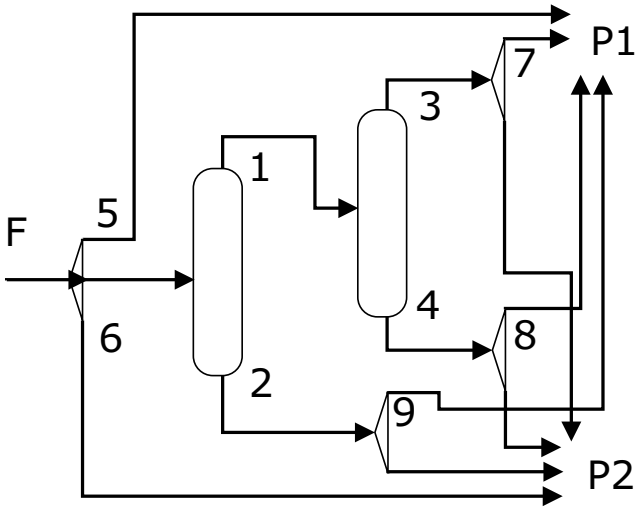

Figure 4: *Used separation problem: The feed (F) is split into two product streams P1 and P2 by use of 2 columns. Numbers indicate optimization variables controlling column requirements (1-4) and stream divider proportions (5-9).*

The problem instance dealt with in the following has been described by Aggarwal and Floudas [14] and tackled with several deterministic algorithms (mainly MINLP). A three–component feed stream has to be separated into two three–component products with defined component proportions. We use two columns and 4 stream dividers, resulting in 9 real–valued optimization variables. The optimization task is to minimize the yearly cost of a facility that performs this separation (see Fig. 4).

Search points generated by the EA are first put into a shortcut simulator that computes the layout and checks if the described process is possible at all. E.g., column pressures cannot exceed certain limits, and mass and heat balances must be valid. If so, the design can be evaluated rigorously with a much more time consuming commercial simulator. But even at this stage simulation can fail unexpectedly, meaning that the suggested design is not realizable. The problem is thus heavily constrained: only one inequality constraint can be given in algebraic form but 18 others are hidden in the shortcut simulator.

As can be expected, it is not easy for an EA to generate valid solutions at all. To enable testing different parameters without spending much time on simulation, we utilize the non–linear fitness function defined in [14] for search points regarded as valid by the shortcut simulator. In fact, first

Table 5: *Algorithm design for optimizing the separation problem with an ES, other parameters are kept at default values. Original values correspond to manually tested parameter settings, the best found configuration (last column) is well beyond these ranges for three parameters.*

Parameter name	$\mathbb{N}/\mathbb{R}$	Min	Max	Original	Best
Population size μ:	$\mathbb{N}$	10	100	10-20	98
Maximum age κ:	$\mathbb{N}$	1	100	1-20	1
Selection pressure λ/μ:	$\mathbb{R}_+$	0.1	10	1-5	7.76
Learning rate τ:	$\mathbb{R}_+$	0.0	1.0	0.05-0.2	0.6

tests with a multi–membered evolution strategy (ES) and several default parameter settings (column original in Tab. 5) revealed that the valid search space volume is small and seemingly unconnected (non–convex). Although discretized penalties had been added, the ES mostly failed to generate even one valid solution throughout a whole optimization run.

Random search delivers estimates for the probability to generate a valid solution by chance (ρ metric, see [15]), it is below 10^{-5} for this problem. At this point, we applied SPO to obtain a set of ES parameters—if any—that guarantees valid solutions with high probability. Table 5 enumerates minimum and maximum values for algorithm designs tried by SPO. Note that chosen ranges are quite large to allow for testing non–standard parameter settings. The remaining ES parameters are kept at standard values (discrete recombination, one step size, initial mutation strengths 0.25) throughout all runs. We used the mean best fitness of all runs of a design point as experiment outcome Y, whereas each run was assigned a budget of 10000 evaluations which is quite large considered that the shortcut simulator is able to perform only ≈ 20 evaluations per second.

After only three sequential steps with 25 initial design points (152 runs alltogether), SPO found that despite the small budget allowed, relatively large population sizes and high selection pressures work best (last column in Tab. 5). The most surprising finding is that tuned self–adaptation learning rates are much higher than recommended values $\tau = 1/\sqrt{2N} \approx 0.24$ [16] for multimodal functions. The best found variant employs $\tau = 0.6$, other good configurations choose $\tau \in [0.3, 0.5]$.

A possible explanation for these values is that large selection pressures and high learning rates together enable fast response to better (lesser constrained) newly found search points, their neighborhood is explored more intense. The increased population size probably helps maintaining diversity much longer than with default values $\mu \in [10, 20]$.

We also performed a control experiment to verify that the new parameter settings are not an artifact of 'lucky' sampling. Of 40 runs, the tuned ES found valid solutions 26 times, corresponding to a 65% success rate, compared to a success rates of less than 10% for the manually tested parameters.

4 Discussion and Outlook

In the previous section, we have presented three different applications of SPO. They are different in the following sense: in Sec. 3.1, the algorithm-problem interaction is unknown because the technique is relatively new; however, the test problem is well-known. In Sec. 3.2, comparison of different algorithms has been the main task. Even though both problems and algorithms have been tested thoroughly before, their interaction is obviously not understood well. Otherwise, good parameter values could have been estimated beforehand. In Sec. 3.3, the problem features are relatively unknown so that the interaction is once more unpredictable. In all three cases, SPO found parameter values that led to increased performance, thereby utilizing the algorithms potential to a much higher extent.

Summarizing, we can state that whenever parameters for an algorithm-problem combination have not been thoroughly searched before, application of SPO makes sense because otherwise one cannot be sure to have a competitive parameter set. As bad configurations of an optimization method can lead to drastic performance losses, it can render a whole empirical study worthless.

The remainig question is: are there useful alternatives to SPO? Our experience indicates that classical regression techniques such as linear models can be applied too [17]. Some approaches exist to automate classical statistical methods. But these methods are much more complicated compared to SPO. A visualization of the response surface may indicate where linear regression models fail: the response surface may be highly multi-modal or even chaotic. Systematic (grid) search is also not an option because its cost increases exponentially in the number of parameters.

However, there are some drawbacks, too. SPO is constrained to decimal and integer values which is an obvious limitation. If number of combinations of nominal or ordinal parameters is low, you can optimize for each combination and choose the best one. Nevertheless, sometimes it is possible to transform nominal or ordinal scales into continuous ones, e.g. a (μ, κ, λ)–strategy allows a stepwise transition from (μ, λ) to $(\mu + \lambda)$–selection. Another difficulty when applying SPO stems from the fact that it also requires the specification of some — fortunately few — parameter values. Concerning this issue, there are currently no theoretical results available, so that one has to rely on experience from previous experimental studies.

It is often easy to find algorithm designs that do not work—or, alternatively, to find a problem instance that disturbs the algorithm's performance. However, SPO makes the determination of working algorithm designs for specific problems easy, too. Thereby, it enables fair comparisons between algorithms, hopefully resulting in increased knowledge gain for empirical studies in evolutionary computation yet to come.

Acknowledgment

T. Bartz–Beielstein's research was supported by the DFG as part of the collaborative research center "Computational Intelligence" (531). Christian W.G. Lasarczyk gratefully acknowledges support from a DFG grant to W.B. under Ba 1042/7–3. Mike Preuss gratefully acknowledges DFG grant SCHW 361/13-1, in close cooperation with the Chair of Technical Thermodynamics (LTT) of the RWTH Aachen,

namely Prof. Dr. Klaus Lucas and Frank Henrich.

Bibliography

[1] T. Bartz-Beielstein and S. Markon, "Tuning search algorithms for real-world applications: A regression tree based approach," in *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*. Portland, Oregon: IEEE Press, 20-23 June 2004, pp. 1111–1118.

[2] T. Bartz-Beielstein, "New Experimentalism Applied to Evolutionary Computation," Ph.D. dissertation, University of Dortmund, April 2005.

[3] T. Santner, B. Williams, and W. Notz, *The Design and Analysis of Computer Experiments*. Springer, 2003.

[4] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[5] M. Schonlau, W. Welch, and R. Jones, "Global versus local search in constrained optimization of computer models," in *New developments and applications in experimental design*, N. Flournoy, W. Rosenberger, and W. Wong, Eds. Institute of Mathematical Statistics, 1998, vol. 34, pp. 11–25.

[6] S. Lophaven, H. Nielsen, and J. Søndergaard, "DACE - A Matlab Kriging Toolbox," Informatics and Mathematical Modelling, Technical University of Denmark, Tech. Rep. IMM-REP-2002-12, 2002.

[7] M. Schonlau, "Computer experiments and global optimization," Ph.D. dissertation, University of Waterloo, Ontario, Canada, 1997.

[8] C. W. G. Lasarczyk and W. Banzhaf, "An algorithmic chemistry for genetic programming," in *Proceedings of the 8th European Conference on Genetic Programming*, ser. LNCS, M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447. Berlin Heidelberg: Springer, 2005, pp. 1–12.

[9] ——, "Total synthesis of algorithmic chemistries," in *Gecco 2005*, 2005, (accepted).

[10] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proc. Sixth International Symposium on Micro Machine and Human Science (Nagoya, Japan)*. Piscataway, NJ: IEEE Service Center, 1995, pp. 39–43.

[11] R. Eberhart and Y. Shi, "Comparison between genetic algorithms and particle swarm optimization," in *Evolutionary Programming*, V. Porto, N. Saravanan, D. Waagen, and A. Eiben, Eds. Springer, 1998, vol. VII, pp. 611–616.

[12] Y. Shi and R. Eberhart, "Empirical study of particle swarm optimization," in *Proceedings of the Congress of Evolutionary Computation*, P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, Eds., vol. 3, 1999, pp. 1945–1950.

[13] T. Bartz-Beielstein, K. E. Parsopoulos, and M. N. Vrahatis, "Design and analysis of optimization algorithms using computational statistics," *Applied Numerical Analysis & Computational Mathematics (ANACM)*, vol. 1, no. 2, pp. 413–433, 2004.

[14] A. Aggarwal and C. A. Floudas, "Synthesis of general distillation sequences — nonsharp separations," *Computers chem. Engng.*, vol. 14, pp. 631–653, 1990.

[15] Z. Michalewicz and M. Schoenauer, "Evolutionary algorithms for constrained parameter optimization problems," *Evolutionary Computation*, vol. 4, no. 1, pp. 1–32, 1996.

[16] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies: A comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3–52, 2002.

[17] T. Bartz-Beielstein, "Experimental analysis of evolution strategies - overview and comprehensive introduction," University of Dortmund, Technical Report of the Collaborative Research Center 531 *Computational Intelligence* CI–157/03, 2003.