SPOT: An R Package For Automatic and Interactive Tuning of Optimization Algorithms by Sequential Parameter Optimization

Thomas Bartz-Beielstein
Department of Computer Science,
Cologne University of Applied Sciences,
51643 Gummersbach, Germany

June 19, 2010

Abstract

The sequential parameter optimization (SPOT) package for R (R Development Core Team, 2008) is a toolbox for tuning and understanding simulation and optimization algorithms. Model-based investigations are common approaches in simulation and optimization. Sequential parameter optimization has been developed, because there is a strong need for sound statistical analysis of simulation and optimization algorithms. SPOT includes methods for tuning based on classical regression and analysis of variance techniques; tree-based models such as CART and random forest; Gaussian process models (Kriging) and combinations of different metamodeling approaches. This article exemplifies how SPOT can be used for automatic and interactive tuning.

1 Introduction

This article illustrates the functions of the SPOT package. The SPOT package can be downloaded from the comprehensive R archive network at `http://CRAN.R-project.org/package=SPOT`. SPOT is one possible implementation of the *sequential parameter optimization* (SPO) framework introduced in Bartz-Beielstein (2006). For a detailed documentation of the functions from the SPOT package, the reader is referred to the package help manuals.

The performance of modern search heuristics such as *evolution strategies* (ES), *differential evolution* (DE), or *simulated annealing* (SANN) relies crucially on their parametrizations—or, statistically speaking, on their factor settings. The term *algorithm design* summarizes factors that influence the behavior (performance) of an algorithm, whereas *problem design* refers to factors from the optimization (simulation) problem. Population size in ES is one typical factor which belongs to the algorithm design, the search space dimension belongs to

the problem design. We will consider SANN in the remainder of this article, because it requires the specification of two algorithm parameters only.

One major goal of SPO is to detect the importance of certain parts (subroutines such as recombination) by systematically varying the factor settings of the algorithm design. This task is related to improving the algorithm's *efficiency* and will be referred to in the following as *algorithm tuning*, where the experimenter is seeking for an improved parameter setting, say $\vec{p}^*$, for one problem instance. Varying problem instances, e.g., search space dimensions or starting points of the algorithm, are associated with *effectivity* or the algorithm's robustness. In this case, the experimenter is interested in one parameter setting of the algorithm with which the algorithm performs sufficiently good on several problem instances. SPOT can be applied for both tasks. The focus of this article lies on improving the algorithm's efficiency.

Besides an improved performance of the algorithm, SPO may lead to a better understanding of the algorithm. SPO combines several techniques from classical and modern statistics, namely *design of experiments* (DoE) and *design and analysis of computer experiments* (DACE) (Bartz-Beielstein, 2006). Basic ideas from SPO rely heavily on Kleijnen's work on statistical techniques in simulation (Kleijnen, 1987, 2008).

The paper is structured as follows: Section 2 presents an introductory example which illustrates the use of tuning. The sequential parameter optimization framework is presented in Sect. 3. Details of the sequential parameter optimization toolbox are presented in Sect. 4. SPOT uses plugins. Typical plugins are discussed in Sect. 5. How SPOT can be refined is exemplified in Sect. 6. Section 7 presents a summary and an outlook.

2 Motivation

2.1 A Typical Situation

We will discuss a typical situation from optimization. The practitioner is interested in optimizing an objective function, say f, with an optimization algorithm A. She can use the optimization algorithm with default parameters. This may lead to good results in some cases, whereas in other situations results are not satisfactory. In the latter cases, practitioners try to determine improved parameter settings for the algorithms manually, e.g., by changing one algorithm parameter at a time. Before we will discuss problems related to this approach, we will take a broader view and consider the general framework of optimization via simulation which occurs in many real-world optimization scenarios.

2.2 Optimization via Simulation

2.2.1 Modeling Layers

To illustrate the task of optimization via simulation, the following layers can be used.

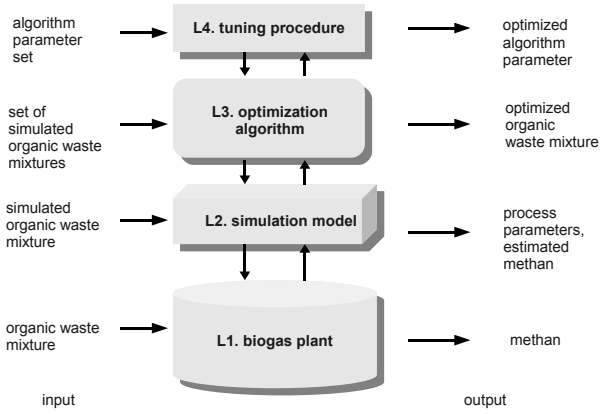

Figure 1: Optimization via simulation. Illustration taken from Ziegenhirt et al. (2010), who describe how SPOT can be applied to the optimization of a biogas-simulation model. Four different layers of the biogas simulation of shown. The first layer (L1) represents the real-world setting. Layer 2 (L2) shows the simulator. An objective function f is defined at this layer. The optimization algorithm A belongs to the third layer (L3). The fourth layer (L4) represents the algorithm tuning procedure, e.g., sequential parameter optimization.

(L1) The real-world system, e.g., a biogas plant.

(L2) The related simulation model. The objective function f is defined at this layer. In optimization via simulation, problem parameters are defined at this layer.

(L3) The optimization algorithm A. It requires the specification of algorithm parameters, say $\vec{p} \in \vec{P}$, where $\vec{P}$ denotes the set of parameter vectors.

(L4) The experiments and the tuning procedure.

Figure 1 illustrates the situation. To keep the setting as simple as possible, we consider an objective function f from layer (L2) and do not discuss interactions between (L1) and (L2). Defining the relationship between (L1) and (L2) is not a trivial task. The reader is referred to Law (2007) and Fu (2002) for an introduction.

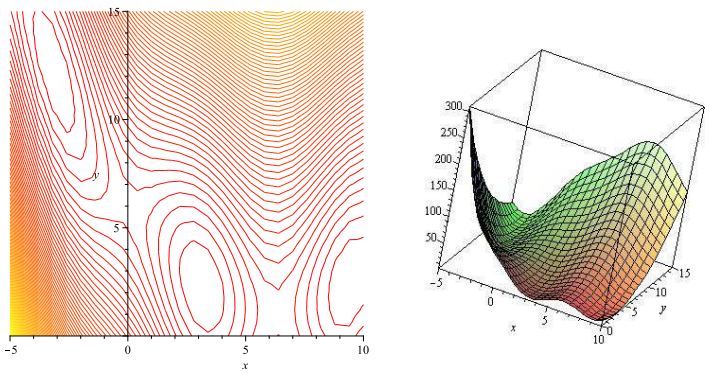

Figure 2: Plots of the Branin function. The contour plot (*left*) shows the location of the three global minima

2.3 Description of the Objective Function

The Branin function

$$f(x_1, x_2) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10 \times \left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10,$$

with

$$x_1 \in [-5, 10] \text{ and } x_2 \in [0, 15]. \tag{1}$$

was chosen as a test function, because it is well-known in the global optimization community, so results are comparable. It has three global minima, $\vec{x}_1^* = [3.1416, 2.2750]$, $\vec{x}_2^* = [9.4248, 2.4750]$ and $\vec{x}_3^* = [-3.1416, 12.2750]$ with $y^* = f(\vec{x}_i^*) = 0.39789$, $(i = 1, 2, 3)$, see Fig. 2.

2.4 Description of the Optimization Algorithm

In order to improve reproducibility of the examples presented in this article, an algorithm which is an integral part of the R system, the method SANN, was chosen. It is described in R's help system as follows (R Development Core Team, 2008): Method SANN is by default a variant of simulated annealing given in Belisle (1992). Simulated annealing belongs to the class of stochastic global optimization methods. It uses only function values but is relatively slow. It will also work for non-differentiable functions. This implementation uses the Metropolis function for the acceptance probability. By default the next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. If a function to generate a new candidate point is given, method SANN can also be used to solve combinatorial optimization problems. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992); specifically, the temperature is set to

```
temp / log(((t-1) \%/\% tmax)*tmax + exp(1))
```

where t is the current iteration step and `temp` and `tmax` are specifiable via control. Note that the SANN method depends critically on the settings of the control parameters. Summarizing, there are two algorithm parameters which have to be specified before the algorithm is run:

1. `temp` controls the SANN method. It is the starting temperature for the cooling schedule. Defaults to 10.

2. `tmax` is the number of function evaluations at each temperature for the SANN method. Defaults to 10.

Note, `tmax` is an integer. How different parameter types can be handled is described in Sec. 6.2. To simply the discussion, `temp` will be treated as a numerical value in the remainder of this article.

2.5 Starting Optimization Runs

Now we discuss the typical situation from optimization: An experimenter applies an optimization algorithm A (SANN) to an objective function f (Branin function) in order to determine the minimum.

First, we will set the seed to obtain reproducible results.

```
> set.seed(1)
```

Next, we will define the objective function.

```
> spotFunctionBranin <- function(x) {
+     x1 <- x[1]
+     x2 <- x[2]
+     (x2 - 5.1/(4 * pi^2) * (x1^2) + 5/pi * x1 - 6)^2 + 10 * (1 -
+         1/(8 * pi)) * cos(x1) + 10
+ }
```

Then, the starting point for the optimization, $\vec{x}_0$, and the number of function evaluations, `maxit`, are defined:

```
> x0 <- c(10, 10)
> maxit <- 250
```

The parameters specified so far belong to the problem design. Now we have to consider parameters from the algorithm design, i.e., parameters that control the behavior of the SANN algorithm, namely `tmax` and `temp`. Default values are chosen first:

```
> tmax <- 10
> temp <- 10
```

Finally, we can start the optimization algorithm (SANN):

Table 1: Results from manually tuning SANN on Branin function. Smaller values are better. Run 1 reports results from the default configuration. Run 2 uses a different temperature and obtains a better function value. However, this result cannot be generalized, because modifying the seed leads to a worse function value

run	temp	tmax	seed	result
1	10	10	1	4.067359
2	20	10	1	0.4570975
3	20	10	1000	7.989125

```
> y1 <- optim(x0, spotFunctionBranin, method = "SANN", control =
+     list(maxit = maxit, temp = temp, tmax = tmax))
```

SANN returns the following result:

```
> print(y1$value)

[1] 4.067359
```

Since the optimum value reads $y^* = 0.39789$, the practitioner is interested in improving this result by modifying the algorithm parameters `tmax` and `temp`:

```
> tmax <- 10
> temp <- 20
> y2 <- optim(x0, spotFunctionBranin, method = "SANN",
+    control = list(maxit = maxit, temp = temp, tmax = tmax))
```

Results obtained with the new `tmax` and `temp` values look promising:

```
> print(y2$value)

[1] 0.4570975
```

However, since SANN is a stochastic algorithm, the practitioner wants to investigate the dependency of the results on the random seed. So she performs the same experiment with modified seed.

```
> set.seed(1000)
> y3 <- optim(x0, spotFunctionBranin, method = "SANN",
+       control = list(maxit = maxit, temp = temp, tmax = tmax))
> print(y3$value)

[1] 7.989125
```

This result is rather disappointing, because a worse value is obtained with this seemingly better parameter settings. Results from these experiments are summarized in Tab. 1.

The practitioner has modified one variable (`temp`) only. Introducing variations of the second variable (`tmax`) complicates the situation, because interactions between these two variables might occur. And, the experimenter has to take random effects into account. Here comes SPOT into play. SPOT was developed for tuning algorithms in a reproducible way. It uses results from algorithm runs to build up a meta model. This meta model enables the experimenter to detect important input variables, estimate effects, and determine improved algorithm configurations in a reproducible manner. Last but not least, the experimenter learns from these results.

One simple goal, which can be tackled with SPOT, is to determine the best parameter setting of the optimization algorithm for one specific instance of an optimization problem. It is not easy to define the term "best", because it can be defined in many ways and this definition is usually problem specific. Klein (2002) presents interesting aspects from practice. See also the discussion in chapter 7 of Bartz-Beielstein (2006). Therefore, we will take a naive approach by defining our tuning goal as the following hypothesis:

(H-1) "We can determine a parameter setting $\vec{p}^*$ which improves SANN's performance. To measure this performance gain, the average function values from ten runs of SANN with default, i.e., $\vec{p}^0$ and tuned parameter $\vec{p}^*$ settings are compared."

2.6 Tuning with SPOT

Before SPOT is described in detail, we will demonstrate how it can be applied to find an answer for hypothesis (H-1).

2.6.1 SPOT Projects

A SPOT *project* consists of a set of files with the same basename, but different extensions, namely CONF, ROI, and APD. Here, we will discuss the project `demo7RandomForestSann`, which is included in the SPOT package, see

```
> demo(package="SPOT")
```

for demos in the SPOT package. Demo projects, which are included in the SPOT package, can be found in the directory of your local SPOT installation, e.g., `~/Ri486-pc-linux-gnu-library/2.11/SPOT` on Linux systems.

A *configuration* (CONF) file, which stores information about SPOT specific settings, has to be set up. For example, the number of SANN algorithm runs, i.e., the available budget, can be specified via `auto.loop.nevals`. SPOT implements a sequential approach, i.e., the available budget is not used in one step. Evaluations of the algorithm on a subset of this budget, the so-called initial design, is used to generate a coarse grained meta model F. This meta model is used to determine promising algorithm design points which will be evaluated next. Results from these additional SANN runs are used to refine the meta model F. The size of the initial design can be specified via `init.design.size`.

To generate the meta model, we use random forest (Breiman, 2001). This can be specified via `seq.predictionModel.func = "spotPredictRandomForest"`. Available meta models are listed in Sect. 5.3. Random forest was chosen, because it is a robust method which can handle categorical and numerical variables. In the following example, we will use the configuration file `demo7RandomForestSann.conf`.

A *region of interest* (ROI) file specifies algorithm parameters and associated lower and upper bounds for the algorithm parameters. Values for `temp` are chosen from the interval $[1; 50]$. `TEMP 1 50 FLOAT` is the corresponding line for the `temp` parameter which is added to the file `demo7RandomForestSann.roi`.

Optionally, an *algorithm problem design* (APD) file can be specified. This file contains information about the problem and might be used by the algorithm. For example, the starting point `x0 = c(10,10)` can be specified in the apd file. The file `demo7RandomForestSann.apd` will be used in our example.

2.6.2 Starting SPOT in Automatic Mode

If these files are available, SPOT can be started from R's command line via

```
> library(SPOT)
> spot(``demo7RandomForestSann.conf'')
```

SPOT is run in automatic mode, if no task is specified (this is the default setting). Result from this run reads

```
Best solution found with 236 evaluations:
    Y     TEMP     TMAX   COUNT CONFIG
0.3992229 1.283295   41    10     36
```

SPOT has determined a configuration `temp` $= 1.283295$ and `tmax` $= 41$, which gives an average function value from ten runs of $\overline{y} = 0.3998429$. SPOT uses an internal counter (COUNT) for configurations. The best solution was found with configuration 36. The tuning process is illustrated in Fig. 3. Figure 4 shows a regression tree which is generated by the default report function `spotReportDefault`.

2.7 Validating the Results

Finally, we will evaluate the result by comparing ten runs of SANN with default parameter settings, say $\vec{p}^0$ to ten runs with the tuned configurations from SPOT, say $\vec{p}^*$. The corresponding R commands used for this comparison are shown in the Appendix. First, we will set the seed to obtain reproducible results. Next, we will define the objective function. Then, the starting point for the optimization $\vec{x}_0$ and the number of function evaluations `maxit` are defined. The parameters specified so far belong to the problem design.

Now we have to consider parameters from the algorithm design, i.e., parameters that control the behavior of the SANN algorithm, namely `tmax` and `temp`. Finally, we can start the optimization algorithm (SANN). The run is finished with the following summary:

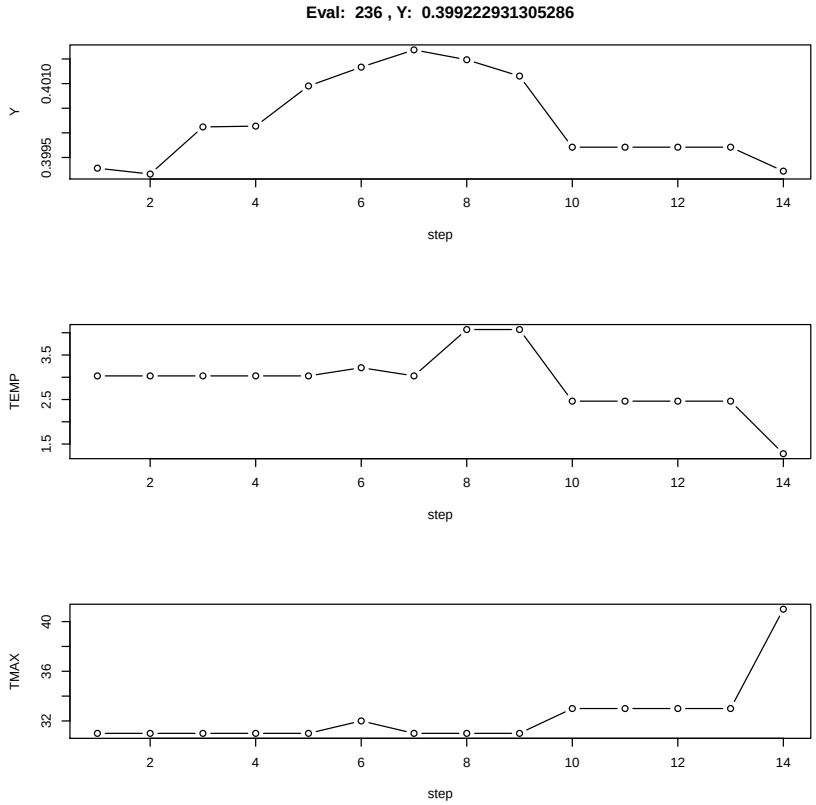

Figure 3: Tuning SANN with SPOT. Random forest was chosen as a meta model. This output can also be shown on-line, i.e., during the tuning process in order to visualize the progress. The first panel shows the average function value of the best configuration found so far. The second and third panel visualize corresponding parameter settings. These values are updated each time a new meta model (random forest) is build. Each time a meta model is build, the step counter is increased. Altogether 14 meta models (random forest) are build during this tuning process and 236 runs of the SANN algorithm were executed

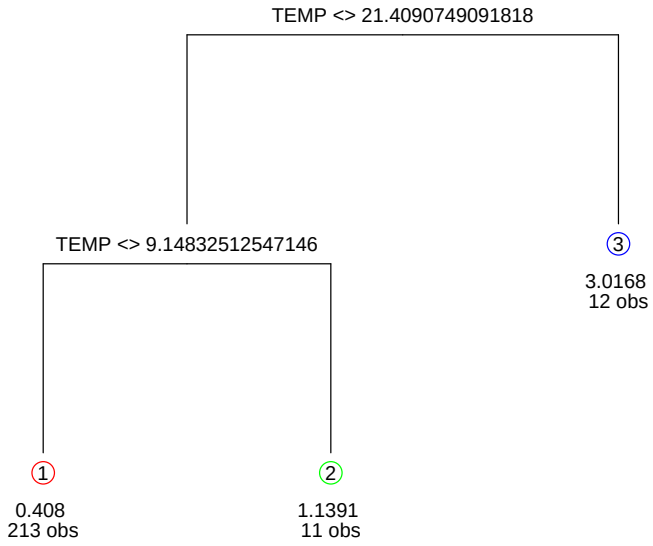

Figure 4: Tuning SANN with SPOT. Random forest was chosen as a meta model. This simple regression tree is generated by SPOT's default report function `spotReportDefault`. The tree illustrates that `temp` has the largest effect. Values at the terminal node t_i show the average function value and the number of observations (obs) which fulfill the conditions which are given by following the tree from the root node to t_i. Smaller `temp` values improve SANN's performance. A value of `temp`, which is smaller than 9.14, results in an average function value of $\overline{y} = 0.408$. This result is based on 213 observations

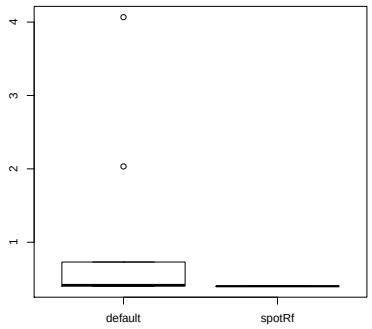

Figure 5: Comparison of SANN's default parameter values (*default*) with parameter settings obtained with SPOT, where random forest was chosen as a meta model (*spotRf*)

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3995  0.4037  0.4174  0.9716  0.6577  4.0670
```

In order to illustrate the performance gain from SPOT's tuning procedure, SANN is run with the tuned parameter configuration, i.e., `temp` = 1.283295 and `tmax` = 41. Results from these ten SANN runs can be summarized as follows:

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3980  0.3982  0.3984  0.3995  0.3989  0.4047
```

Further statistical analyses, e.g., the box plot shown in Fig. 5, reveal that this difference is statistically significant. Hence, hypothesis (H-1) cannot be rejected. After this quick introduction we will have a closer look at SPOT.

3 Sequential Parameter Optimization

3.1 Definition

Definition 3.1 (Sequential Parameter Optimization)**.** Sequential parameter optimization *(*SPO*) is a framework for tuning and understanding of algorithms by active experimentation. SPO employs methods from error statistics to obtain reliable results. It comprises the following elements:*

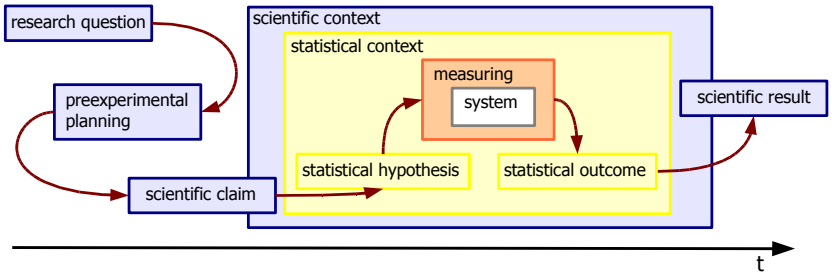

Figure 6: Steps and contexts of performing an experiment from research question to scientific result

SPO-1: *Scientific questions*
SPO-2: *Statistical hypotheses*
SPO-3: *Experiments*
SPO-4: *Scientific meaning*

□

These elements can be explained as follows. Starting point of the investigation is a scientific question (SPO-1). This question often deals with assumptions about algorithms, e.g., influence of parameter values or new operators. This (complex) question is broken down into (simple) statistical hypotheses (SPO-2) for testing, see Bartz-Beielstein (2008) for an example. Next, experiments can be performed for each hypothesis, e.g., (H-1) as defined in Sect. 2.5.

a) Select a model F (e.g., random forest) to describe a functional relationship.

b) Select an experimental design, e.g., Latin hypercube design.

c) Generate data, i.e., perform experiments.

d) Refine the model until the hypothesis can be accepted/rejected.

Performing these experiments will be referred to as step (SPO-3). Finally, to assess the scientific meaning of the results from an experiment, conclusions are drawn from the hypotheses. This is step (SPO-4) in the sequential parameter optimization framework, see Definition 3.1. Figure 6 illustrates the SPO framework. SPOT implements the steps from the statistical context.

This article describes one specific instance of (SPO-3), which implements the corresponding software programs in R. It will be referred to as SPOT.

3.2 Sequential Parameter Optimization Toolbox

We introduce R's SPOT package as one possible implementation of step (SPO-3) from the SPO framework. Implementations in other programming languages, e.g., MATLAB, are also available but are not subject of this article.

The SPO *toolbox* was developed over recent years by Thomas Bartz-Beielstein, Christian Lasarczyk, and Mike Preuss (Bartz-Beielstein et al., 2005). Main goals of SPOT are (i) the determination of improved parameter settings for optimization algorithms and (ii) to provide statistical tools for analyzing and understanding their performance.

Definition 3.2 (Sequential Parameter Optimization Toolbox)**.** *The sequential parameter optimization toolbox implements the following features, which are related to step (SPO-3) from the* SPO *framework.*

SPOT-1: *Use the available budget (e.g., simulator runs, number of function evaluations) sequentially, i.e., use information from the exploration of the search space to guide the search by building one or several meta models. Choose new design points based on predictions from the meta model(s). Refine the meta model(s) stepwise to improve knowledge about the search space.*

SPOT-2: *If necessary, try to cope with noise by improving confidence. Guarantee comparable confidence for search points.*

SPOT-3: *Collect information to learn from this tuning process, e.g., apply exploratory data analysis.*

SPOT-4: *Provide mechanisms both for interactive and automatic tuning.*

□

The article entitled "sequential parameter optimization" (Bartz-Beielstein et al., 2005) was the first attempt to summarize results from seminars and tutorials given at conferences such as CEC and GECCO and make this approach known to and available for a broader audience (Beielstein, 2002; Bartz-Beielstein and Preuß, 2004; Bartz-Beielstein, 2005; Bartz-Beielstein and Preuß, 2005a,b).

SPOT was successfully applied in the fields of bioinformatics (Volkert, 2006; Fober et al., 2009), environmental engineering (Konen et al., 2009; Flasch et al., 2010), shipbuilding (Rudolph et al., 2009), fuzzy logic (Yi, 2008), multimodal optimization (Preuss et al., 2007), statistical analysis of algorithms (Lasarczyk, 2007; Trautmann and Mehnen, 2009), multicriteria optimization (Bartz-Beielstein et al., 2009), genetic programming (Lasarczyk and Banzhaf, 2005), particle swarm optimization (Bartz-Beielstein et al., 2004; Kramer et al., 2007), automatic and manual parameter tuning (Fober, 2006; Smit and Eiben, 2009; Hutter et al., 2010a,b), graph drawing (Tosic, 2006; Pothmann, 2007), aerospace and shipbuilding industry (Naujoks et al., 2006), mechanical engineering (Mehnen et al., 2007), and chemical engineering (Henrich et al., 2008). Bartz-Beielstein (2010) collects more than 100 publications related to the sequential parameter optimization.

3.3 Elements of the SPOT Framework

3.3.1 The General SPOT Scheme

Algorithm 1 presents a formal description of the SPOT scheme. The *utility* is used to measure algorithm's performance. Typical measures are the estimated mean

Algorithm 1: SPOT

```
   // phase 1, building the model:
1  let A be the tuned algorithm;
2  generate an initial population P⃗ = {p⃗^1, ..., p⃗^m} of m parameter vectors;
3  let k = k_0 be the initial number of tests for determining estimated utilities;
4  foreach p⃗^i ∈ P⃗ do
5      run A with p⃗^i k times to determine the estimated utility u^i (e.g., average
       function value from 10 runs) of p⃗^i;
   // phase 2, using and improving the model:
6  while termination criterion not true do
7      let p⃗^* denote the parameter vector from P⃗ with best estimated utility;
8      let k the number of repeats already computed for p⃗^*;
9      build prediction model F based on P⃗ and u^1, ..., u^|P⃗|};
10     generate a set P⃗' of l new parameter vectors by random sampling;
11     foreach p⃗^i ∈ P⃗' do
12         calculate f(p⃗^i) to determine the predicted utility F(p⃗^i);
13     select set P⃗'' of d parameter vectors from P⃗' with best predicted utility
       (d ≪ l);
14     run A with p⃗^* once and recalculate its estimated utility using all k + 1 test
           results; // (improve confidence)
15     update k, e.g., let k = k + 1;
16     run A k times with each p⃗^i ∈ P⃗'' to determine the estimated utility F(p⃗^i);
17     extend the population by P⃗ = P⃗ ∪ P⃗'';
```

or median from several runs of A. Algorithm 1 consists of two phases, namely the first construction of the model (lines 1–5) and its sequential improvement (lines 6–17). Phase 1 determines a population of initial designs in algorithm parameter space and runs the algorithm k times for each design. Phase 2 consists of a loop with the following components:

1. Update the meta model F (or several meta models F_i) by means of the obtained data.

2. Generate a (large) set of design points and compute their utility by sampling the model.

3. Select the seemingly best design points and run the algorithm for these.

4. The new design points are added to the population and the loop starts over if the termination criterion is not reached.

A counter k is increased in each cycle and used to determine the number of repeats that are performed for each setting to be statistically sound in the obtained results. In consequence, this means that the best design points so far are also run again to obtain a comparable number of repeats. These reevalua-

tions may worsen the estimated performance and explains increasing Y values in Fig. 3.

Sequential approaches can be more efficient than approaches that evaluate the information in one step only (Wald, 1947). This presumes an experienced operator who is able to draw the right conclusions out of the first results. In case the operator is new to SPOT the sequential steps can be started automatically. Compared to interactive procedures, performance in the automatic tuning process may decrease. However, results from different algorithm runs, e.g., ES and SANN, will be comparable in an objective manner if data for the comparison is based on the same tuning procedure.

Extensions to the SPOT approach were proposed by other authors, e.g., Lasarczyk (2007) integrated an *optimal computational budget allocation* procedure, which is based on ideas by Chen et al. (2003). Due to SPOT's plugin structure, see Sect. 5, further extensions can easily be integrated.

3.3.2 Running SPOT

In Sect. 2.6.2, SPOT was run as an automatic tuner. Steps from the automatic mode can be used in an interactive manner. SPOT can be started with the command

```
spot (<configurationfile>,  <task>)
```

where `configurationfile` is the name of the SPOT configuration file and `task` can be one of the tasks `init`, `seq`, `run`, `rep` or `auto`. SPOT can also be run in a `meta` mode to perform tuning over a set of problem instances.

Files Used During the Tuning Process Each configuration file belongs to one SPOT project, if the same basename is used for corresponding files. SPOT uses simple text files as interfaces from the algorithm to the statistical tools.

1. The user has to provide the following files:
   (i) *Region of interest* (ROI) files specify the region over which the algorithm parameters are tuned. Categorical variables such as the recombination operator in ES, can be encoded as factors, e.g., "intermediate recombination" and "discrete recombination."
   (ii) *Algorithm design* (APD) files are used to specify parameters used by the algorithm, e.g., problem dimension, objective function, starting point, or initial seed.
   (iii) *Configuration* files (CONF) specify SPOT specific parameters, such as the prediction model or the initial design size.
2. SPOT will generate the following files:
   (i) *Design* files (DES) specify algorithm designs. They are generated automatically by SPOT and will be read by the optimization algorithms.

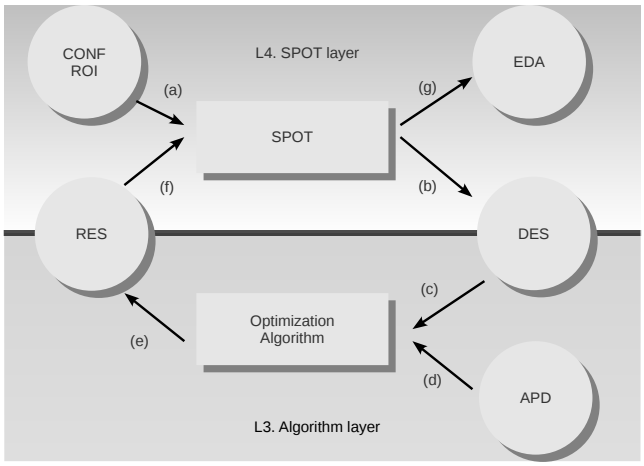

Figure 7: SPOT interfaces. The SPOT loop can be described as follows: *Configuration* (CONF) and *region-of-interest* (ROI) files are read by SPOT (a). SPOT generates a *design* (DES) file (b). The algorithm reads the design file and (c) extra information, e.g., about the problem dimension from the *algorithm-problem design* (APD) file (d). Output from the optimization algorithm are written to the *result* (RES) file (e). The result file is used by SPOT to build the prediction model (f). Data can be used by *exploratory data analysis* (EDA) tools to generate reports, statistics, visualizations, etc. (g)

(ii) After the algorithm has been started with a parametrization from the algorithm design, the algorithm writes its results to the *result file* (RES). Result files provide the basis for many statistical evaluations/visualizations. They are read by SPOT to generate prediction models. Additional prediction models can easily be integrated into SPOT.

Figure 7 illustrates SPOT interfaces and the data flow. The acronym EDA (exploratory data analysis) summarizes additional information that can be used to add further statistical tools. For example, SPOT writes a best file (BST), which summarizes information about the best configuration during the tuning process. Note, that the problem design can be modified, too. This can be done to analyze the robustness (effectivity) of algorithms.

SPOT Tasks SPOT provides tools to perform the following tasks (see also Fig. 8):

1. *Initialize*. An initial design is generated. This is usually the first step

during experimentation. The employed parameter region (ROI) and the constant algorithm parameters (APD) have to be provided by the user. SPOT's parameters are specified in the CONF file. Although it is recommended to use the same basename for CONF, ROI, and APD files in order to define a project, this is not mandatory. SPOT allows a flexible combination of different filenames, e.g., one APD file can be used for different projects.

2. *Run.* This is usually the second step. The optimization algorithm is started with configurations of the generated design. Additionally information about the algorithms problem design are used in this step. The algorithm writes its results to the result file.

3. *Sequential step.* A new design, based on information from the result file, is generated. A prediction model is used in this step. Several generic prediction models are available in SPOT by default. To perform an efficient analysis, especially in situations when only few algorithms runs are possible, user-specified prediction models can easily be integrated into SPOT.

4. *Report.* An analysis, based on information from the result file, is generated. Since all data flow is stored in files, new report facilities can be added very easily. SPOT contains some scripts to perform a basic regression analysis and plots such as histograms, scatter plots, plots of the residuals, etc.

5. *Automatic* mode. In the automatic mode, the steps *run* and *sequential* are performed after an initialization for a predetermined number of times.

6. *Meta* mode. In the `meta` mode, the tuning process is repeated for several configurations. For example, tuning can be performed for different starting points $\vec{x}_0$, several dimensions, or randomly chosen problem instances.

As stated in Sect. 3.2, SPOT has been applied to several optimization tasks which might give further hints how SPOT can be used. Bartz-Beielstein Bartz-Beielstein and Preuss (2010); Bartz-Beielstein et al. (2010) present case studies that may serve as good starting points for SPOT applications.

4 Details

We will discuss functions which are used during the four SPOT steps initialize, run, sequential, and report.

4.1 Initialize

During this step, the initial design is generated and written to the design file. `spotCreateDesignLhs`, which is based on R's `lhs` package, is recommended as a simple space filling design.

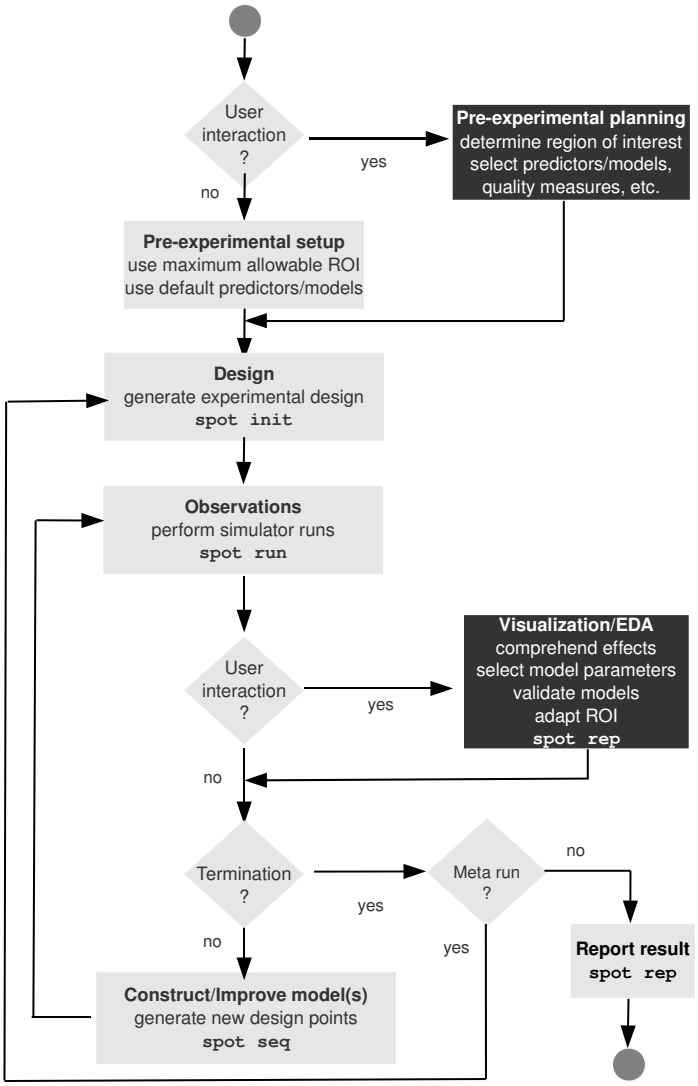

Figure 8: The SPOT process. *White font color* indicates steps used in the interactive process only. A *typewriter font* indicates the corresponding SPOT commands. To start the automatic mode, simply use the task `auto`. Note that the interaction points are optional, so SPOT can be run without any user interaction. `meta` runs perform tuning processes for several configurations

Alternatively, factorial designs can be used. `spotCreateDesignFrF2`, which is based on Groemping's `FrF2` package, generates a fractional factorial design with center point.

Furthermore, the number of initial design points, the type of the experimental design etc. have to be specified before the first SPOT run is performed. These information are stored in the *configuration file*(CONF), see Listing 1.

Listing 1: demo7RandomForestSann.conf

```
alg.func = "spotAlgStartSann"
auto.loop.nevals = 100
init.design.func = "spotCreateDesignLhs"
init.design.size = 10
init.design.repeats  = 2
seq.predictionModel.func = "spotPredictRandomForest"
```

The configuration file plays a central role in SPOT's tuning process. It stores information about the optimization algorithm (`alg.func`) and the meta model (`seq.predictionModel.func`). SPOT uses a classification scheme for its variables: `init` refers to variables which were used during the initialization step, `seq` are variables used during the sequential step, and so forth.

The experimental region is specified in the *region of interest* (ROI) file, see Listing 2. In the `demo7RandomForestSann` project, two numerical variables with values from the interval $[1; 50]$ are used. SPOT employs a mechanism which adapts the region of interest automatically. Information about the actual region of interest are stored in the `aroi` file, which is handled by SPOT internally.

Listing 2: demo7RandomForestSann.roi

```
name low high type
TEMP 1 50 FLOAT
TMAX 1 50 FLOAT
```

Now all the source files are available. In order to generate the initial design, simply call SPOT as follows.

```
spot("demo7RandomForestSann.conf","init")
```

The "init" call generates a *design file* (DES), which is shown in Listing 3.

Listing 3: Design file demo7RandomForestSann.des generated by SPOT

```
TEMP                TMAX              CONFIG REPEATS STEP SEED
35.6081542731496    20.5193298289552     1   2     0 1235
3.03074611076154    30.9299647800624     2   2     0 1235
35.0430960887112    11.6981793923769     3   2     0 1235
18.7132237656275    49.8082008753903     4   2     0 1235
13.9964890434407    35.3148515065433     5   2     0 1235
26.2654501354089    25.7817166472087     6   2     0 1235
24.1049260527361    3.67511987574399     7   2     0 1235
7.34134425916709    25.289775890857      8   2     0 1235
49.035177000449     42.5625789203448     9   2     0 1235
42.5434358732775    7.34647449669428    10   2     0 1235
```

Since we have chosen the SPOT plugin `spotCreateDesignLhs`, a Latin hypercube design is generated. Each configuration is labeled with a configuration number. The column REPEATS contains information from the variable `init.design.repeats`. Since no meta model has been created yet, STEP is set to 0 for each configuration. Finally, the SEED, which is used by the algorithm, is shown in the last column.

4.2 Run

Parameters from the design file are read and the algorithm is executed. Each run results in one fitness value (single-objective optimization) or several values (multi-objective optimization). Fitness values with corresponding parameter settings are written to the result file. The user has to set up her own interface for her algorithm A. Examples are provided, see Sect. 5.2. The command

```
spot("demo7RandomForestSann.conf","run")
```

executes the `run` task. Results from this run are written to the *result file*(RES), which is shown in Listing 4.

Listing 4: demo7RandomForestSann.res

```
Y TEMP TMAX FUNCTION DIM SEED CONFIG STEP
3.30597157332377 35.6081542731496 21 Branin0.0 2 1235 1 0
5.0386282545543 35.6081542731496 21 Branin0.0 2 1236 1 0
0.4003069540417 71 3.03074611076154 31 Branin0.0 2 1235 2 0
0.398257998573415 3.03074611076154 31 Branin0.0 2 1236 2 0
0.445520325383134 35.0430960887112 12 Branin0.0 2 1235 3 0
2.226649807247 35.0430960887112 12 Branin0.0 2 1236 3 0
0.497684708317145 18.7132237656275 50 Branin0.0 2 1235 4 0
1.83495383747858 18.7132237656275 50 Branin0.0 2 1236 4 0
0.399119231011623 13.9964890434407 35 Branin0.0 2 1235 5 0
0.542312919825205 13.9964890434407 35 Branin0.0 2 1236 5 0
2.72322377892531 26.2654501354089 26 Branin0.0 2 1235 6 0
0.417190547410852 26.2654501354089 26 Branin0.0 2 1236 6 0
4.7533857289203 24.1049260527361 4 Branin0.0 2 1235 7 0
1.61981739130332 24.1049260527361 4 Branin0.0 2 1236 7 0
0.403447629608046 7.34134425916709 25 Branin0.0 2 1235 8 0
0.411160853072728 7.34134425916709 25 Branin0.0 2 1236 8 0
2.98491602241286 49.035177000449 43 Branin0.0 2 1235 9 0
5.24485760127901 49.035177000449 43 Branin0.0 2 1236 9 0
6.91947230812718 42.5434358732775 7 Branin0.0 2 1235 10 0
0.521764744656569 42.5434358732775 7 Branin0.0 2 1236 10 0
```

4.3 Sequential

Now that results have been written to the result file, the meta model can be build.

```
spot("demo7RandomForestSann.conf","seq")
```

The sequential call generates a new *design file* (DES), which is shown in Listing 5.

Listing 5: Design file demo7RandomForestSann.des generated by SPOT

```
TEMP TMAX CONFIG REPEATS repeatsLastConfig STEP SEED
3.03074611076154 31 2 1 2 1 1237
6.13796767716994 31.5014664068934 11 3 3 1 1235
1.50800024462165 28.6290849421476 12 3 3 1 1235
```

In order to improve confidence, the best solution found so far is evaluated again. To enable fair comparisons, new configurations are evaluated as many times as the best configuration found so far. Note, other update schemes are possible.

If SPOT's budget is not exhausted, the new configurations are evaluated, i.e., `run` is called again, which updates the result file. In the following step, `seq` is called again etc.

To support exploratory data analysis, SPOT also generates a best file, which is shown in Listing 6.

Listing 6: Best file demo7RandomForestSann.bst generated by SPOT

```
Y TEMP TMAX COUNT CONFIG
0.398592091098704 3.03074611076154 31 2 2
0.399500174065 72 1.8086370804091 31 3 11
0.399448475332373 1.8086370804091 31 4 11
0.399732585200016 1.8086370804091 31 5 11
0.399443742300062 1.8086370804091 31 6 11
0.400057920171103 3.03074611076154 31 3 2
0.400239513036257 1.8086370804091 31 7 11
0.400845187050665 1.8086370804091 31 9 11
0.400892337114249 3.82960996863898 30 4 13
0.401138794573396 1.8086370804091 31 10 11
0.399842856102580 1.20055107064079 31 10 29
0.399842856102580 1.20055107064079 31 10 29
0.399842856102580 1.20055107064079 31 10 29
```

The best file is updated after each SPOT iteration and be be used for an on-line visualization tool, e.g., to illustrate search progress or stagnation, see Fig. 3. The variable COUNT reports the number of REPEATS used for this specific configuration.

4.4 Report

If SPOT's termination criterion is fulfilled, a report is generated. By default, SPOT provides as simple report function which reads data from the res file and produces the following output:

```
Best solution found with 223 evaluations:
     Y     TEMP TMAX COUNT CONFIG
0.3998429 1.200551   31    10     29
```

4.5 Automatic

SPOT's `auto` task performs steps `init`, `run`, `seq`, `run`, `seq`, etc. until the termination criterion is fulfilled, see Fig. 8. It can be invoked from R's command line via

```
spot("demo7RandomForestSann.conf","auto")
```

5 Plugins

SPOT comes with a basic set of R functions for generating inital designs, starting optimization algorithms, building meta models, and generating reports. This set can easily be extended with user defined R functions, so called plugins. Further plugins will be added in forthcoming SPOT versions. Here, we describe the interfaces that are necessary for integrating user-defined plugins into SPOT.

5.1 Initialization Plugins

The default plugin for generating an initial design is `init.design.func = "spotCreateDesignLhs"`. It uses information about the size of the initial design `init.design.size`. The number n of design variables x_i ($i = 1, \ldots, n$), their names `pNames`, and their ranges $a_i \leq x_i \leq b_i$ can be determined with SPOT's internal `alg.roi` variable, which is passed to the initialization plugin.

```
> pNames <- row.names(alg.roi);
> a <-  alg.roi[ ,"low"];
> b <-  alg.roi[ ,"high"];
```

Note, `pNames`, a, and b are vectors of size n. Based on this information, a data frame with inital design points is generated. For example, the data frame from the `demo7RandomForestSann` project reads:

```
        TEMP      TMAX
1  35.608154 20.519330
2   3.030746 30.929965
3  35.043096 11.698179
4  18.713224 49.808201
5  13.996489 35.314852
6  26.265450 25.781717
7  24.104926  3.675120
8   7.341344 25.289776
9  49.035177 42.562579
10 42.543436  7.346474
```

These values are written to the initial design file, see Listing 3. The reader is referred to the `spotCreateDesingLhs` function for further details.

The plugin `spotCreateDesignFrF2` generates a central composite design and can be used as a template for fractional factorial design plugins. Currently, SPOT implements the following `init` plugins:

- `spotCreateBasicDoe3R`: creates a fractional-factorial design (resolution III)

- `spotCreateFrF2`: creates a resolution III design with center point and star points
- `spotCreateLhs`: creates a Latin hypercube design

Note, these plugins should be used as templates and can be easily adopted to specific situations.

5.2 Run Plugins

The `run` plugin `spotAlgStartSann`, which is used as an interface to R's SANN algorithm, is shown in the Appendix, see Listing 8. Basically, the user has to specify variable names to be read from the design file, see Sect. 5.2.1, and written to the result file, see Sect. 5.2.3, and the call of the algorithm, see Sect. 5.2.2.

5.2.1 Reading Values From the Design File

To add a new variable, say COLOR, the user simply adds the following line of code to the run file:

```
if (is.element("COLOR", pNames)){color <- des$COLOR[k]}
```

5.2.2 Executing the Algorithms

Next, the call of the algorithm has to be specified. In our example,

```
y <- optim(x0, spotFunctionBranin, method="SANN",
control=list(maxit=maxit, temp=temp, tmax=tmax, parscale=parscale,
color=color))
```

5.2.3 Writing Results to the Result File

And finally, in order to write the variable to the result file, it has to be added to the following list:

```
res <- list(Y = y, TEMP = temp, TMAX = tmax,
COLOR = color, SEED = seed, CONFIG = conf)
```

5.2.4 Interfacing With Algorithms Written in Other Programming Languages

We will demonstrate how JAVA programs can be called from SPOT. The procedure consists of two steps: First, a call string is build. Then, R's `system` function is used for executing the callString.

```
callString <- paste("java -jar simpleOnePlusOneES.jar",
seed, steps, target, f, n, xp0, sigma0, a, g, px, py, sep = " ")
y <-system(callString, intern= TRUE)
```

This procedure can be applied to any optimization algorithm. Templates for state-of-the-art optimization algorithms will be added to forthcoming SPOT version. Users are encouraged to submit interfaces to their algorithms to the SPOT development team.

Currently, SPOT implements the following `run` plugins:

- `spotAlgStartSann:` Interface to R's simulated annealing SANN
- `spotAlgStartES:` Interface to an ES based on Beyer and Schwefel (2002)
- `spotFuncStartBranin:` Interface to the Branin function. SPOT is used as an optimizer, not as a tuner, see also Sect. 6.4

Additional `run` packages are available, e.g.,

```
> demo(spotDemo11Java)
```

demonstrates how a (1+1)-ES, which is implemented in Java, can be tuned with SPOT.

5.3 Sequential Plugins

During SPOT's sequential step one or several meta models are generated. These models use information from the result file. New, promising design points are generated. Therefore, a large number of randomly generated design points are evaluated on the meta model. Configurations with the best estimated objective function values are written to the design file and will be evaluated during the `run` step, see line 11 in Algorithm 1.

SPOT provides two types of data assembled from the result file: *Raw* data comprehend parameter values $\vec{x}$ (configurations) and related objective function values y, whereas *merged* data map same $\vec{x}_i$ configurations to one configuration $\vec{x}_j$. The corresponding y_i values are merged according to the merge function (default: mean), e.g., $y_j = \sum_1^n y_i/n$. In our example, the random forest is generated with raw data. R's generic `predict` function is used to evaluate new data on the meta model (random forest). Finally, the best design points are determined.

The random forest meta model is implemented as shown in Listing 7.

Listing 7: spotPredictRandomForest.R

```
spotPredictRandomForest <- function(rawB,mergedB,largeDesign,
    spotConfig){
spotInstAndLoadPackages("randomForest")
xNames <- setdiff(names(rawB),"y")
x <- rawB[,xNames]; y <- rawB$y
fit <- randomForest(x, y)
res <- predict(fit,largeDesign)
largeDesign <-  largeDesign[order(res,decreasing=FALSE),]
newDesign <- largeDesign[1:spotConfig$seq.design.new.size,]}
```

Currrently (June 2010), SPOT provides interfaces to the following meta modeling approaches:

- Regression models (`lm`; `rsm`):
  1. `spotPredictLm`
  2. `spotPredictLmOptim`
- Tree based models (`tree`; `randomForest`)
  1. `spotPredictTree`
  2. `spotPredictRandomForest`
- Gaussian process models (`mlegp`; `tgp`)
  1. `spotPredictTgp`
  2. `spotPredictMlegp`

The Appendix presents an example (Listing 9) how several meta models can be combined. Interfaces to further meta models will be provided in future releases of the SPOT package.

5.4 Report Plugins

SPOT comes with a simple report plugin `spotReportDefault.R`. It reports the best configuration from the tuning procedure, illustrates the tuning process (evolution of the best solution as shown in Figs. 4 and 13), and generates a simple regression tree as shown in Fig. 9.

User defined report functions can easily be added. Wolfgang Konen has written a report plugin which uses `randomForest` to visualize factor effects, see Fig. 10. Figure 11 demonstrates how EDA tools can be applied to analyse effects and interactions. Note, results from the result file can be used for detailed reports. At this stage, EDA tools are the method of choice.

6 Refinement

6.1 Combining Meta Models And Adaptation of the Region of Interest

During the sequential step, SPOT can use different meta models. The following example demonstrates how results from tree based regression and response surface modeling can be combined.

6.1.1 Designs

A *central composite design* (CCD) was chosen as the starting point of the tuning process. SPOT's `spotCreateDesignFrF2` plugin can be used to generate design points. After the first run is finished, we can use SPOT's report facility to analyze results. Since we have chosen a classical factorial design, we will use *response-surface methodology* (RSM). Lenth (2009) describes an implementation of RSM

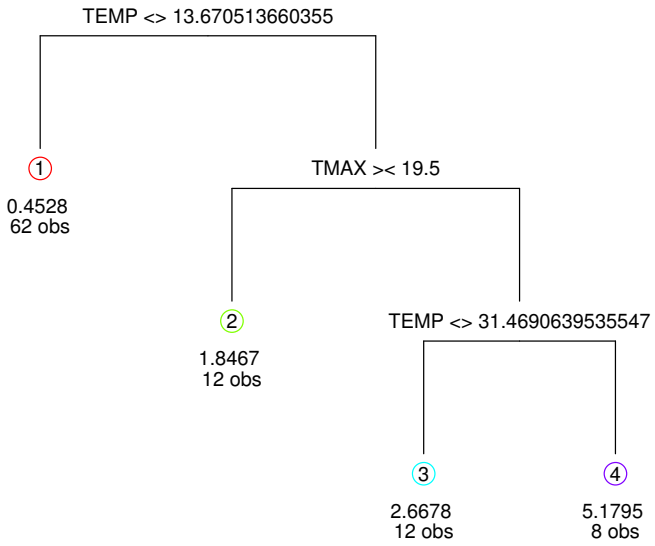

Figure 9: Tuning SANN with SPOT. An `rsm` and `tree` based approach are combined. Similar to the random forest based meta modeling, TEMP has the largest effect

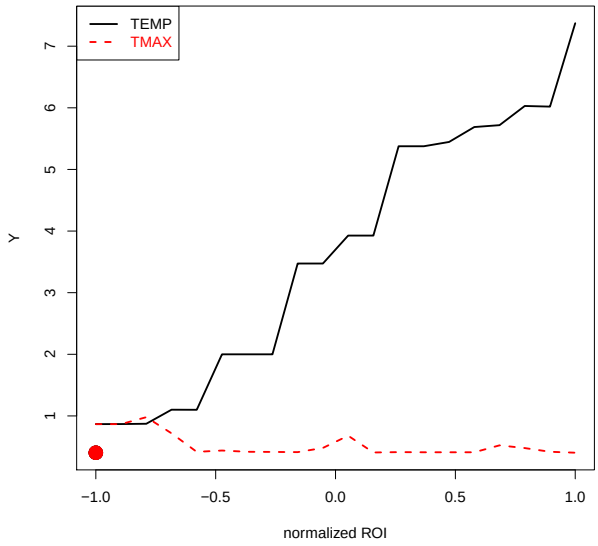

Figure 10: Results from the SANN tuning procedure. Function values Y plotted versus parameter values. `randomForest` was used to predict values for one variable, say `temp`, while the other variable (`tmax`) was set its optimal value. Values for both variables were normalized. This plot was generated with the `spotReportSens` plugin

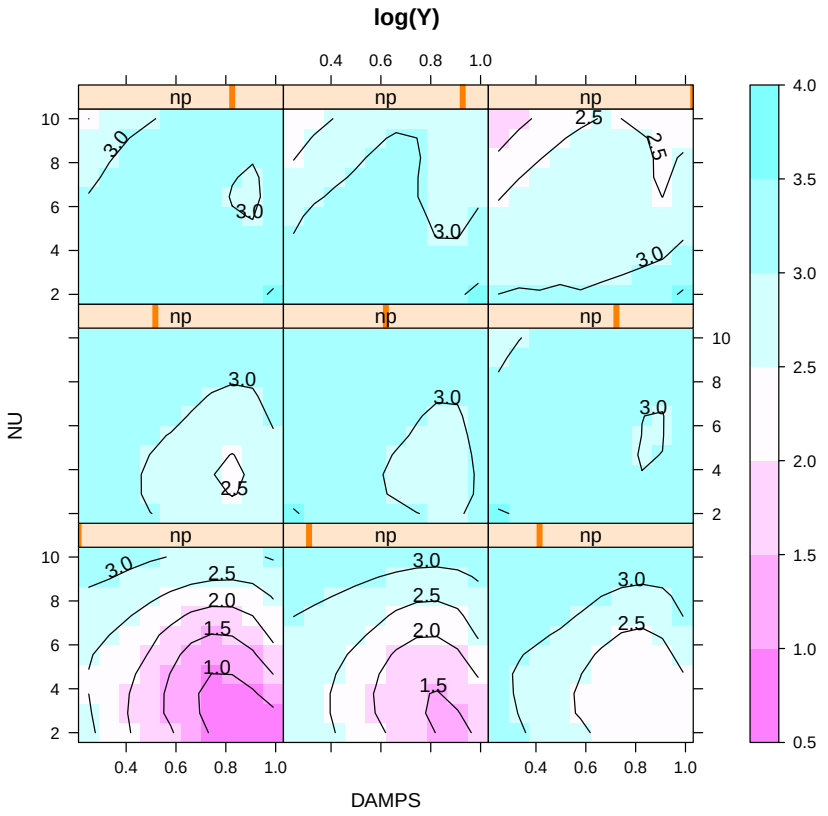

Figure 11: This figure shows an EDA example taken from Bartz-Beielstein et al. (2010). Contour plots based on 82 function evaluations of the CMA evolution strategy (CMA-ES) optimizing the Ackley function are shown (Hansen, 2006). Smaller values are better. Better configurations are placed in the lower area of the panels. The CMA-ES has four algorithm parameters (CS, NU, DAMPS, and NPARENTS). The parameter CS is held constant. NU is plotted versus DAMPS, while values of the parameter NPARENTS (np), are varied with the slider on top of each panel

in R. This R package `rsm` has many useful tools for an analysis of the results from the SPOT runs. After evaluating the algorithm in these design points, a seccond order regression model with interactions is fitted to the data. Functions from the `rsm` package were used by the SPOT plugin `spotPredictLmOptim`. Before meta models are build, data are standardized. Data in the original units are mapped to coded data, i.e., data with values in the interval $[-1, 1]$.

6.1.2 Response Surface Models

Based on the number of design points, SPOT automatically determines whether a first-order, two-way interaction, pure quadratic, or second order model can be fitted to the data. The CCD generated by `spotCreateDesignFrF2` allows the fit of an second-order model which can be summarized as follows.

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.93070    0.46212   2.014   0.1375
x1           2.29074    0.36382   6.296   0.0081 **
x2          -1.98286    0.36307  -5.461   0.0121 *
x1:x2       -2.29498    0.40674  -5.642   0.0110 *
x1^2         1.86950    0.87244   2.143   0.1215
x2^2        -0.05832    0.87337  -0.067   0.9510
---
Residual standard error: 0.8135 on 3 degrees of freedom
Multiple R-squared: 0.9734,        Adjusted R-squared: 0.9291
F-statistic: 21.97 on 5 and 3 DF,  p-value: 0.01437
```

6.1.3 Using Gradient Information

The response surface analysis determines the following stationary point on response surface: $(-0.8447783, -0.3781626)$, or, in the original units `temp` = 4.802932 and `tmax` = 16.235016. The eigenanalysis shows that the eigenvalues ($\lambda_1 = 2.4042085$; $\lambda_2 = -0.5930265$) have different signs, so this is a saddle point, as can also be seen in Fig. 12. SPOT automatically determines the path of the steepest descent and selects five points, using the old center point as a starting point, in its direction: (23.7605, 27.215), (22.315, 29.224), (21.4085, 31.6005), (21.139, 34.271), and (21.4085, 37.0395), i.e., decreasing `temp` and increasing `tmax` values are chosen. Rather than at the origin, SPOT can start the search at the saddle point. Set `seq.useCanonicalPath = TRUE` to enable this feature. In this case, SPOT determines the most steeply rising ridge in both directions, see also Lenth (2009) for details:

```
dist     x1     x2 |   TEMP    TMAX
1 -0.2 -0.760 -0.197 | 6.8800 20.6735
2 -0.1 -0.803 -0.288 | 5.8265 18.4440
3  0.0 -0.845 -0.378 | 4.7975 16.2390
4  0.1 -0.887 -0.469 | 3.7685 14.0095
5  0.2 -0.929 -0.559 | 2.7395 11.8045
```

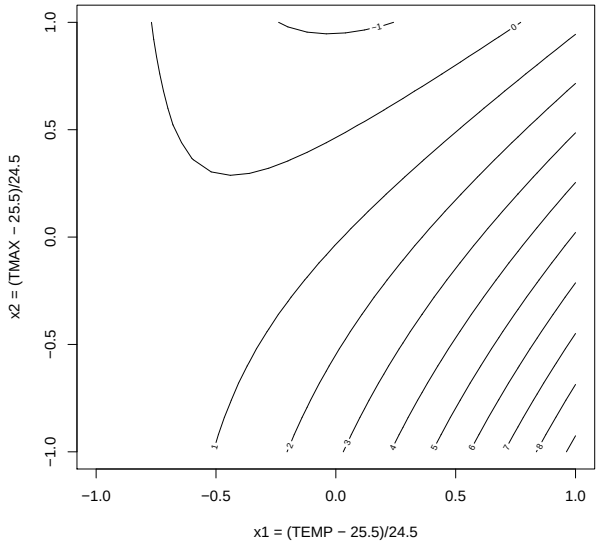

Figure 12: Response surface model based on the initial design. `rsm` was used to generate this plot

In addition to the points from the steepest descent, the best point from the first design (1,50) is evaluated again. Now, these points are evaluated and a new `rsm` model is build.

6.1.4 Automatic Adaptation of the Region of Interest

SPOT modifies the region of interest, if `seq.useAdaptiveRoi = TRUE`. This procedure consists of two phases, which are repeated in an alternating manner.

During the *orientation* phase, the direction of the largest improvement is determined as described in Sect. 6.1.3. Based on an existing design and related function values, the path of the steepest descent is determined. A small number of points is chosen from this path. Optimization runs are performed on these design points. In some situations, where no gradient information is available, the best point from a large number of design points, which were evaluated on the regression model, is chosen as the set of improvement points.

The *recalibration* phase determines the best point $\vec{x}_b$. It can be selected from the complete set of evaluated design points or from the points along the steepest descent only. The best point $\vec{x}_b$ defines the new center point of a central composite design. The minimal distance of $\vec{x}_b$ to the borders of the actual region of interest defines the radius of this design. If $\vec{x}_b$ is located at (or very close to) the borders of the region of interest, a Latin hypercube design which covers the whole region of interest is determined. This can be interpreted as a restart. To prevent premature convergence of this procedure, one additional new design point is generated by a tree based model.

Next, the orientation phase is repeated. The tuning process with adaptive ROI is visualized in Fig. 13. The final output from this tuning process, which is based on regression models and tree based regression reads:

```
Best solution found with 94 evaluations:
          Y TEMP TMAX COUNT CONFIG
0.4006016    1    1     6      2
```

As in Sect. 2.6, ten repeats of the best solution from this tuning process are generated. Results are shown in Fig. 14. Note, this result was found with only half of the number of SANN runs compared to the random forest modeling approach from Sect. 2.6. The example demonstrates how the usage of gradient information can accelerate the tuning procedure.

6.2 Numerical and Categorical Values

SPOT provides mechanisms for handling type information. Categorical values such as “red”, “green”, and “blue” have to be coded as integer values, e.g., “1”, “2”, and “3” in the ROI file. By default, they are treated as numerical values (`FLOAT`). They can be treated as *factors*, if the corresponding type information (`FACTOR`) is provided in the type column of the ROI file. Alternatively, the type `INT` can be specified in the ROI file. These parameters are treated as numerical values, but the `spotCreateDesign` plugins generate integer values which are

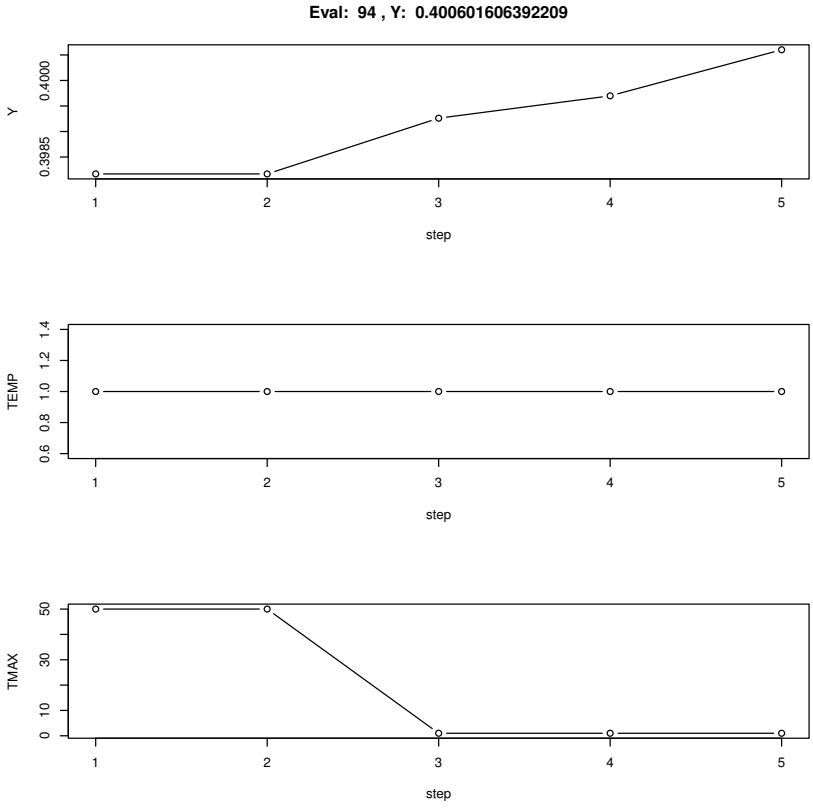

Figure 13: Tuning SANN with SPOT. An rsm and tree based approach are combined

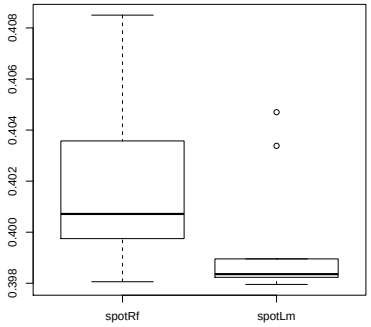

Figure 14: Comparison of SANN's random forest tuned parameter values with parameter settings obtained with `rsm` (*spotLm*)

written to the design files. Bartz-Beielstein et al. (2010) presents an example which illustrates the usage of type information in SPOT.

6.3 Meta Projects

SPOT allows the definition of *meta projects*. Meta project perform tuning over a set of problem instances. One interesting task is to analyze interactions between the search-space dimension, say d, and the best algorithm design $\vec{p}^*$. For example, the experimenter can search for dependencies between population size in ES and d. For a detailed documentation the reader is referred to the package help manuals.

6.4 SPOT as an Optimization Algorithm

SPOT itself can be used as an optimization algorithm. The package includes some demos to illustrate this feature. For example, `spotDemoLm3Branin` uses a linear (meta) model to optimize Branin's function.

7 Summary and Outlook

This article present basic features of the SPOT package which is implemented in R. SPOT provides tools for automatic and interactive tuning of algorithms. Categorical and numerical parameters can be used as input variables, which are specified in the ROI file. A configuration file (CONF) collects data related to the SPOT run (which is considered as a project) such as the prediction model.

The reader is referred to the `SpotGetOptions` help page, which lists SPOT's configuration parameters.

Parameters related to the algorithm or the optimization problem are stored in the APD file. SPOT generates simple text files which are used as interfaces to the algorithm.

The sequential approach comprehends the following steps:

- `init`: generate an initial design
- `run`: evaluate the algorithm
- `seq`: generate new design points (meta model)
- `rep`: statistical analysis and visualization, EDA

Plugins for these steps are subject of on-going research. Plugin development concentrates on combining predictions from several regression models, integrating tools for multi objective optimization, and performing meta SPOT runs.

The SPOT packages contains several demos, which can be used as starting points for setting up your own SPOT project.

8 Acknowledgements

This work was supported by the Bundesministerium für Bildung und Forschung (BMBF) under the grant FIWA (AiF FKZ 17N2309, "Ingenieurnachwuchs") and by the Cologne University of Applied Sciences under the research focus grant COSA. Many thanks go to members of the FIWA and SOMA research group.

9 Appendix

9.1 R Source Code for Starting SANN

Listing 8: spotAlgStartSann.R

```
spotAlgStartSann <- function(io.apdFileName, io.desFileName, io.
    resFileName){
  writeLines(paste("Loading design file data from:", io.
      desFileName), con=stderr());
  source( io.apdFileName,local=TRUE)
  des <- read.table( io.desFileName, sep=" ", header = TRUE);
  pNames <- names(des);
  config<-nrow(des);
  for (k in 1:config){
    for (i in 1:des$REPEATS[k]){
      if (is.element("TEMP", pNames)){
        temp <- des$TEMP[k]
      }
      if (is.element("TMAX", pNames)){
        tmax <- round(des$TMAX[k])
      }
```

```
      conf <- k
      if (is.element("CONFIG", pNames)){
        conf <- des$CONFIG[k]
      }
      spotStep<-NA
      if (is.element("STEP", pNames)){
        spotStep <- des$STEP[k]
      }
      seed <- des$SEED[k]+i-1
      set.seed(seed)
      y <- optim(x0, spotFuncStartBraninSann, method="SANN",
          control=list(maxit=maxit, temp=temp, tmax=tmax, parscale=
              parscale))
      res <- NULL
      res <- list(Y=y$value, TEMP=temp, TMAX=tmax, FUNCTION=f, DIM=
          n, SEED=seed, CONFIG=conf)
      if (is.element("STEP", pNames)){
        res=c(res,STEP=spotStep)
      }
      res <-data.frame(res)
      colNames = TRUE
      if (file.exists(io.resFileName)){
        colNames = FALSE
      }
      write.table(res, file = io.resFileName, row.names = FALSE,
          col.names = colNames, sep = " ", append = !colNames,
              quote = FALSE);
      colNames = FALSE
    }
  }
}
```

9.2 R Source Code for Combining Meta Models

Listing 9: spotPredictRandomForestMlegp.R

```
spotInstAndLoadPackages("mlegp")
xNames <- setdiff(names(rawB),"y")
x <- rawB[,xNames]
y <- rawB$y
rf.fit <- randomForest(x, y)
rf.res <- predict(rf.fit,largeDesign)
rf.largeDesign <- largeDesign[order(rf.res,decreasing=FALSE),]
rf.s <- round(spotConfig$seq.design.new.size/2)
mlegp.s <- spotConfig$seq.design.new.size - rf.s
rf.largeDesign <- rf.largeDesign[1:rf.s,]
if (mlegp.s> 0){
  mlegp.fit <- mlegp(x, y)
  mlegp.res <- predict(mlegp.fit,largeDesign)
  mlegp.largeDesign <- largeDesign[order(rf.res,decreasing=FALSE)
      ,]
  mlegp.largeDesign <- largeDesign[1:mlegp.s,]
  return(rbind(rf.largeDesign,mlegp.largeDesign))
}
else{ return(rf.largeDesign)}
```

```
19    writeLines("spotPredictRandomForestMlegp finished");
      return(largeDesign)
21  }
```

9.3 R Source Code for the Comparison From Sect. 2.7

First, we will set the seed to obtain reproducible results.

```
> set.seed(1)
```

Next, we will define the objective function.

```
> spotFunctionBranin <- function(x) {
+     x1 <- x[1]
+     x2 <- x[2]
+     (x2 - 5.1/(4 * pi^2) * (x1^2) + 5/pi * x1 - 6)^2 + 10 * (1 -
+         1/(8 * pi)) * cos(x1) + 10
+ }
```

Then, the starting point for the optimization x_0 and the number of function evaluations `maxit` are defined:

```
> x0 <- c(10, 10)
> maxit <- 250
```

The parameters specified so far belong to the problem design. Now we have to consider parameters from the algorithm design, i.e., parameters that control the behavior of the SANN algorithm, namely `tmax` and `temp`:

```
> tmax <- 10
> temp <- 10
```

Finally, we can start the optimization algorithm (SANN):

```
> y1 <- NULL
> for (i in 1:10) {
+     set.seed(i)
+     y1 <- c(y1, optim(x0, spotFunctionBranin, method = "SANN",
+         control = list(maxit = maxit, temp = temp,
+         tmax = tmax))$value)
+ }
> summary(y1)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3995  0.4037  0.4174  0.9716  0.6577  4.0670

> temp <- 1.283295
> tmax <- 41
> y2 <- NULL
```

```
> for (i in 1:10) {
+     set.seed(i)
+     y2 <- c(y2, optim(x0, spotFunctionBranin, method = "SANN",
+         control = list(maxit = maxit, temp = temp,
+         tmax = tmax))$value)
+ }
> summary(y2)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3981  0.3999  0.4007  0.4018  0.4035  0.4085

> temp <- 0.1
> tmax <- 1
> y3 <- NULL
> for (i in 1:10) {
+     set.seed(i)
+     y3 <- c(y3, optim(x0, spotFunctionBranin, method = "SANN",
+         control = list(maxit = maxit, temp = temp,
+         tmax = tmax))$value)
+ }
> summary(y3)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3980  0.3982  0.3984  0.3995  0.3989  0.4047
```

References

Bartz-Beielstein, T. (2005). The new experimentalism—an approach to analyze evolutionary algorithms. Institute of Applied Informatics and Formal Description Methods (AIFB), University of Karlsruhe (TH), Karlsruhe. `http://ls11-www.cs.uni-dortmund.de/people/tom`. Cited 30 May 2005.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation—The New Experimentalism*. Natural Computing Series. Springer, Berlin, Heidelberg, New York.

Bartz-Beielstein, T. (2008). How experimental algorithmics can benefit from Mayo's extensions to Neyman-Pearson theory of testing. *Synthese*, 163(3):385–396. DOI 10.1007/s11229-007-9297-z.

Bartz-Beielstein, T. (2010). Sequential parameter optimization—an annotated bibliography. Technical Report 04/2010, Institute of Computer Science, Faculty of Computer Science and Engineering Science, Cologne University of Applied Sciences, Germany.

Bartz-Beielstein, T., Lasarczyk, C., and Preuß, M. (2005). Sequential parameter optimization. In McKay, B. et al., editors, *Proceedings 2005 Congress on*

Evolutionary Computation (CEC'05), Edinburgh, Scotland, volume 1, pages 773–780, Piscataway NJ. IEEE Press.

Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2010). The sequential parameter optimization toolbox. In Bartz-Beielstein, T., Chiarandini, M., Paquete, L., and Preuss, M., editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 337–360. Springer, Berlin, Heidelberg, New York.

Bartz-Beielstein, T., Naujoks, B., Wagner, T., and Wessing, S. (2009). In Locarek-Junge, H. and Weihs, C., editors, *Proceedings 11th IFCS Internat. Conference 2009. Classification as a tool for research*, Dresden.

Bartz-Beielstein, T., Parsopoulos, K. E., and Vrahatis, M. N. (2004). Design and analysis of optimization algorithms using computational statistics. *Applied Numerical Analysis and Computational Mathematics (ANACM)*, 1(2):413–433.

Bartz-Beielstein, T. and Preuß, M. (2004). Experimental research in evolutionary computation (tutorial). Congress on Evolutionary Computation (CEC 2004), Portland OR. `http://ls11-www.cs.uni-dortmund.de/people/tom`. Cited 30 June 2004.

Bartz-Beielstein, T. and Preuß, M. (2005a). Experimental research in evolutionary computation (tutorial). Congress on Evolutionary Computation (CEC 2005), Edinburgh UK. `http://ls11-www.cs.uni-dortmund.de/people/tom`. Cited 10 October 2004.

Bartz-Beielstein, T. and Preuß, M. (2005b). Experimental research in evolutionary computation (tutorial). Genetic and Evolutionary Computation Conf. (GECCO 2005), Washington DC.

Bartz-Beielstein, T. and Preuss, M. (2010). The future of experimental resaerch. In Bartz-Beielstein, T., Chiarandini, M., Paquete, L., and Preuss, M., editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 17–46. Berlin, Heidelberg, New York.

Beielstein, T. (2002). Threshold selection, hypothesis tests, and DOE methods and their applicability to elevator group control problems (seminar). Centrum voor Wiskunde en Informatica, Amsterdam.

Belisle, C. J. P. (1992). Convergence theorems for a class of simulated annealing algorithms. *Journal Applied Probability*, 29:885–895.

Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies—A comprehensive introduction. *Natural Computing*, 1:3–52.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5 –32.

Chen, J., Chen, C., and Kelton, D. (2003). Optimal computing budget allocation of indifference-zone-selection procedures. Working paper, taken from `http://www.cba.uc.edu/faculty/keltonwd`. Cited 6 January 2005.

Flasch, O., Bartz-Beielstein, T., Davtyan, A., Koch, P., Konen, W., Oyetoyan, T. D., and Tamutan, M. (2010). Comparing ci methods for prediction models in environmental engineering. Technical Report 02/2010, Institute of Computer Science, Faculty of Computer Science and Engineering Science, Cologne University of Applied Sciences, Germany.

Fober, T. (2006). Experimentelle Analyse Evolutionärer Algorithmen auf dem CEC 2005 Testfunktionensatz. Master's thesis, Universität Dortmund, Germany.

Fober, T., Mernberger, M., Klebe, G., and Hüllermeier, E. (2009). Evolutionary construction of multiple graph alignments for the structural analysis of biomolecules. *Bioinformatics*, 25(16):2110–2117.

Fu, M. C. (2002). Optimization for simulation. *INFORMS Journal on Computing*, 14(3):192–215.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In Lozano, J., Larranaga, P., Inza, I., and Bengoetxea, E., editors, *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pages 75–102. Springer.

Henrich, F., Bouvy, C., Kausch, C., Lucas, K., Preuß, M., Rudolph, G., and Roosen, P. (2008). Economic optimization of non-sharp separation sequences by means of evolutionary algorithms. *Computers and Chemical Engineering*, 32(7):1411–1432.

Hutter, F., Bartz-Beielstein, T., Hoos, H., Leyton-Brown, K., and Murphy, K. P. (2010a). Sequential model-based parameter optimisation: an experimental investigation of automated and interactive approaches empirical methods for the analysis of optimization algorithms. In Bartz-Beielstein, T., Chiarandini, M., Paquete, L., and Preuss, M., editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 361–414. Springer, Berlin, Heidelberg, New York.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Murphy, K. P. (2010b). Time-bounded sequential parameter optimization. In *Proc. of LION-10*. To appear.

Kleijnen, J. P. C. (1987). *Statistical Tools for Simulation Practitioners*. Marcel Dekker, New York NY.

Kleijnen, J. P. C. (2008). *Design and analysis of simulation experiments*. Springer, New York NY.

Klein, G. (2002). The fiction of optimization. In Gigerenzer, G. and Selten, R., editors, *Bounded Rationality: The Adaptive Toolbox*, pages 103–121. MIT Press, Cambridge MA.

Konen, W., Zimmer, T., and Bartz-Beielstein, T. (2009). Optimierte Modellierung von Füllständen in Regenüberlaufbecken mittels CI-basierter Parameterselektion Optimized Modelling of Fill Levels in Stormwater Tanks Using CI-based Parameter Selection Schemes. *at-Automatisierungstechnik*, 57(3):155–166.

Kramer, O., Gloger, B., and Goebels, A. (2007). An experimental analysis of evolution strategies and particle swarm optimisers using design of experiments. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 674–681, New York, NY, USA. ACM.

Lasarczyk, C. W. G. (2007). *Genetische Programmierung einer algorithmischen Chemie*. PhD thesis, Technische Universität Dortmund.

Lasarczyk, C. W. G. and Banzhaf, W. (2005). Total synthesis of algorithmic chemistries. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1635–1640, New York, NY, USA. ACM.

Law, A. M. (2007). *Simulation Modeling and Analysis*. McGraw-Hill, Boston MA, 4 edition.

Lenth, R. V. (2009). Response-surface methods in `r`, using `rsm`. *Journal of Statistical Software*, 32(7):1–17.

Mehnen, J., Michelitsch, T., Lasarczyk, C., and Bartz-Beielstein, T. (2007). Multi-objective evolutionary design of mold temperature control using DACE for parameter optimization. *International Journal of Applied Electromagnetics and Mechanics*, 25(1–4):661–667.

Naujoks, B., Quagliarella, D., and Bartz-Beielstein, T. (2006). Sequential parameter optimisation of evolutionary algorithms for airfoil design. In Winter, G. e. a., editor, *Proc. Design and Optimization: Methods and Applications, (ERCOFTAC'06)*, pages 231–235. University of Las Palmas de Gran Canaria.

Pothmann, N. H. (2007). Kreuzungsminimierung für k-seitige Buchzeichnungen von Graphen mit Ameisenalgorithmen. Master's thesis, Universität Dortmund, Germany.

Preuss, M., Rudolph, G., and Tumakaka, F. (2007). Solving multimodal problems via multiobjective techniques with Application to phase equilibrium detection. In *Proceedings of the International Congress on Evolutionary Computation (CEC2007). Piscataway (NJ): IEEE Press. Im Druck*.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rudolph, G., Preuss, M., and Quadflieg, J. (2009). Two-layered surrogate modeling for tuning optimization metaheuristics. Algorithm Engineering Report TR09-2-005, Faculty of Computer Science, Algorithm Engineering (Ls11), Technische Universität Dortmund, Germany.

Smit, S. K. and Eiben, A. E. (2009). Comparing Parameter Tuning Methods for Evolutionary Algorithms. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 399–406.

Tosic, M. (2006). Evolutionäre Kreuzungsminimierung. Diploma thesis, University of Dortmund, Germany.

Trautmann, H. and Mehnen, J. (2009). Statistical methods for improving multiobjective evolutionary optimisation. *International Journal of Computational Intelligence Research (IJCIR)*, 5(2):72–78.

Volkert, L. (2006). Investigating ea based training of hmm using a sequential parameter optimization approach. In Yen, G. G. et al., editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 2742–2749, Vancouver, BC, Canada. IEEE Press.

Wald, A. (1947). *Sequential Analysis*. Wiley, New York NY.

Yi, Y. (2008). *Fuzzy Operator Trees for Modeling Utility Functions*. PhD thesis, Philipps-Universität Marburg.

Ziegenhirt, J., Bartz-Beielstein, T., Flasch, O., Konen, W., and Zaefferer, M. (2010). Optimization of biogas production with computational intelligence—a comparative study. In Fogel, G. e. a., editor, *Proc. 2010 Congress on Evolutionary Computation (CEC'10) within IEEE World Congress on Computational Intelligence (WCCI'10), Barcelona, Spain*, Piscataway NJ. IEEE Press.