CIplus
Band 7/2017

In a Nutshell: Sequential Parameter Optimization

Thomas Bartz-Beielstein, Lorenzo Gentile, Martin Zaefferer

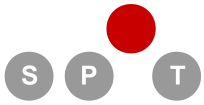
Technology
Arts Sciences
TH Köln

In a Nutshell: Sequential Parameter Optimization

Thomas Bartz-Beielstein
thomas.bartz-beielstein@th-koeln.de

Lorenzo Gentile
lorenzo.gentile@th-koeln.de

Martin Zaefferer
martin.zaefferer@th-koeln.de



SPOTSeven Lab
TH Koeln
Steinmüllerallee 6, 51643 Gummersbach

`www.spotseven.de`

Abstract

The performance of optimization algorithms relies crucially on their parameterizations. Finding good parameter settings is called algorithm tuning. Using a simple simulated annealing algorithm, we will demonstrate how optimization algorithms can be tuned using the Sequential Parameter Optimization Toolbox (SPOT). SPOT provides several tools for automated and interactive tuning. The underling concepts of the SPOT approach are explained. This includes key techniques such as exploratory fitness landscape analysis and response surface methodology. Many examples illustrate how SPOT can be used for understanding the performance of algorithms and gaining insight into algorithm's behavior. Furthermore, we demonstrate how SPOT can be used as optimizer and how a sophisticated ensemble approach is able to combine several meta models via stacking.

1 Introduction

The performance of modern search heuristics such as *evolution strategies* (ES), *differential evolution* (DE), or *simulated annealing* (SANN) relies crucially on their parameterizations—or, statistically speaking, on their factor settings. Finding good parameter settings for an optimization algorithm will be referred to as *tuning*. We will illustrate how an existing search heuristic can be tuned using the Sequential Parameter Optimization Toolbox (SPOT), which is one possible implementation of the *sequential parameter optimization* (SPO) framework introduced in Bartz-Beielstein (2006). The version of SPOT presented in this article is implemented in R. R is "a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc." (R Core Team, 2017). R can be downloaded from `https://cran.r-project.org`. The SPOT package can be installed from within R using the `install.packages()` command.

```
install.packages("SPOT")
```

Note, that the package only has to be installed once (unless an update to a more recent version is needed). Unlike installation, the package has to be loaded to the R work space every time a new R session is started. SPOT can be loaded to the work space with R's `library()` command.

```
library("SPOT")
```

In order to keep the setup as simple as possible, we will use simulated annealing for illustrating the tuning procedure (Kirkpatrick et al., 1983). Simulated annealing is freely available in R. This implementation of the simulated annealing heuristic will be referred to as SANN in the following.

Response surface methodology (RSM) will be used in this article (Box & Draper, 1987). It can be seen as a set of statistical methods for empirical model building. Using design of experiments, a response (dependent variable, output variable, or fitness value, y) that depends on one or several input variables (independent variables or solutions, $\vec{x}$) is optimized. The underlying model can be formulated as

$$y = f(\vec{x}) + \epsilon,$$

where ϵ represents some noise (uncertainty, error observed) in the response y. The term *response surface* refers to the surface represented by $f(\vec{x})$. In order to estimate the quality of a solution, the term *fitness* is used in evolutionary optimization. In physics, the concept of a potential or energy function is used. Since we are dealing with minimization, a low fitness value $f(\vec{x})$ implies that $\vec{x}$ is a good solution.

This report is structured as follows. The algorithm-tuning framework consists of three levels, which are useful for distinguishing several problem domains. These levels are introduced in Sec. 2. The first level is detailed in Sec. 3. Section 4 describes the second level. Here, the well known simulated annealing algorithm is used to exemplify the second level (optimization algorithm). Section 5 describes the third level (tuning). This tuning example can be used as a starting point for beginners. Details of the SPOT configuration are described in Sec. 6. Visual inspection of the results plays an important role in the SPOT approach. Section 7.1 describes the plot functions that are provided in the SPOT toolbox and that enable an exploratory fitness landscape analysis. The response surface methodology is introduced in Sec. 8. Tools for an statistical analysis are described in Sec. 9. Applying SPOT to deterministic problems is briefly explained in Sec. 10. Finally, ensemble models via stacking is described in Sec. 11. This report concludes with a summary in Sec. 12.

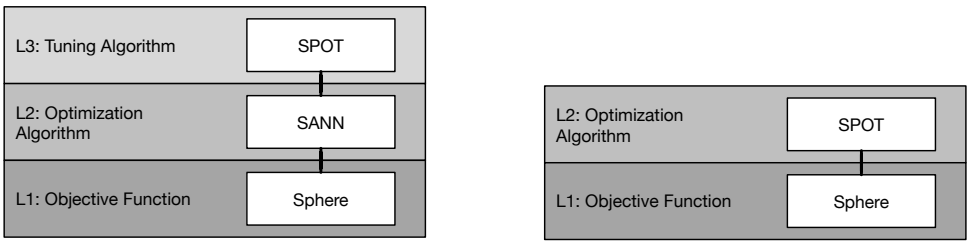


Fig. 1: *Left:* The three levels that occur when tuning an optimization algorithm with SPOT. This setting will be referred to as *algorithm tuning*. *Right:* SPOT can be applied as an optimizer. In this case, SPOT tries to find arguments of the objective function that result in an optimal function value. Following the taxonomy introduced in (Bartz-Beielstein & Zaefferer, 2017), this setting will be referred to as *surrogate model based optimization*.

2 Levels During Tuning and Optimization

We will consider algorithm tuning and surrogate model based optimization in this article. Algorithm tuning is also referred to as off-line parameter tuning in contrast to on-line parameter tuning (Eiben & Smith, 2003). Figure 1 shows the different levels that are used in these scenarios.

2.1 Algorithm Tuning

Tuning an optimization algorithm involves the following three levels can be used to describe the experimental setup.

(L1) The real-world system. This system allows the specification of an objective function, say f. As an example, we will use the sphere function in the following.

(L2) The optimization algorithm, here SANN. It requires the specification of algorithm parameters.

(L3) The tuning algorithm, here SPOT.

An optimization algorithm (L2) requires parameters, e.g., the initial temperature of SANN or the mutation rate of evolution strategies. These parameters determine the performance of the optimization algorithms. Therefore, they should be tuned to get better performance for one algorithm. The algorithm is in turn used to determine optimal values of the objective function f from level (L1).

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. The initial temperature in SANN is one typical factor which belongs to the algorithm design, the search space dimension belongs to the problem design.

2.2 Surrogate Model Based Optimization

Instead of tuning an optimization algorithm, SPOT itself can be used as a surrogate model based optimization algorithm. Then, SPOT has the same role as SANN in the algorithm tuning scenario. In this setting, only two level exist: the objective function (L1) and the optimization algorithm (L2). This situation is seen on the right hand side of Figure 1.

2.3 The SPOT Algorithm

SPOT finds improved solutions in the following way: initially, a population of (random) solutions is created. The initialization step is shown in line 1 in Algorithm 1. Then, the solutions are evaluated on the objective function (level L1). This is line 3 in Algorithm 1. Next, surrogate models are built (line 5). A global search is performed to generate new candidate solutions (line 6). The new solutions

are evaluated on the objective function (level L1) (line 7). These steps are repeated, until a satisfying solution has been found.

Algorithm 1 Sequential parameter optimization

1: $t = 0$. $P(t) = \text{SetInitialPopulation}()$.
2: Select one or several surrogate models $\mathfrak{M}$.
3: Evaluate($P(t)$) on f.
4: **while** not TerminationCriterion() **do**
5: Use $P(t)$ to build a model $M(t)$ using $\mathfrak{M}$.
6: $P'(t+1) = \text{GlobalSearch}(M(t))$.
7: Evaluate($P'(t+1)$) on f.
8: $P(t+1) = P(t) \cup P'(t+1)$.
9: $t = t + 1$.
10: **end while**

3 Level 1: Objective Function

Before SANN can be started, the user has to specify an objective function f. To keep things as simple as possible, the sphere function

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

will be used:

```
sphere <- function(x) {
    sum(x^2)
}
sphere(c(1, 2))

## [1] 5
```

The `sphere()` function uses vector inputs. A matrix-based implementation is defined as `funSphere()` in the SPOT package.

```
funSphere

## function (x)
## matrix(apply(x, 1, function(x) sum(x^2)), , 1)
## <environment: namespace:SPOT>
```

A surface plot of this function in the interval $[0; 1] \times [0; 1]$ is shown in Fig. 2.

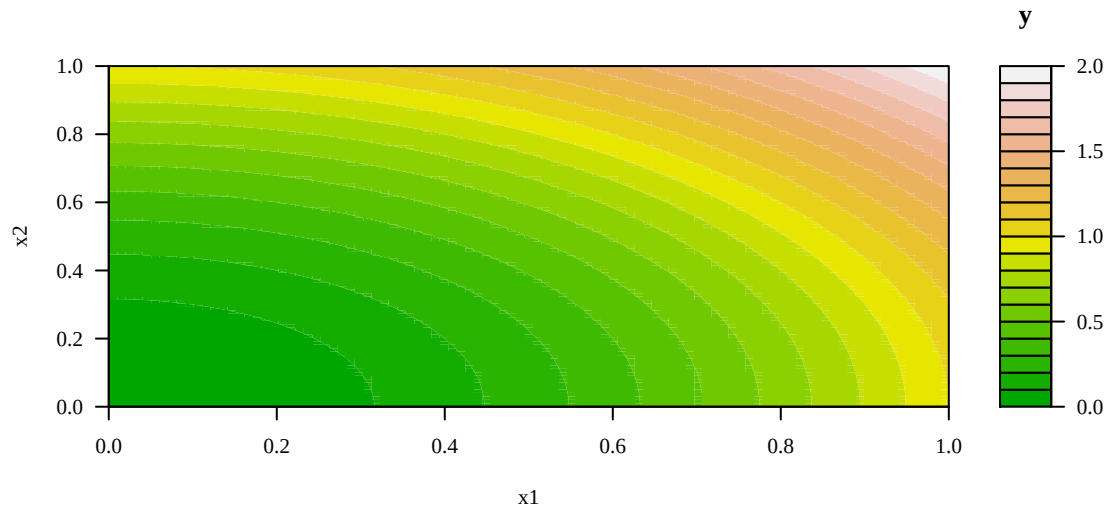


Fig. 2: Surface plot of the sphere function.

4 Level 2: Simulated Annealing SANN

Simulated annealing is a generic probabilistic heuristic for global optimization (Kirkpatrick et al., 1983). The name comes from annealing in metallurgy. Controlled heating and cooling of a material reduces defects. Heating enables atoms to leave their initial positions (which are local minima of their internal energy), and controlled cooling improves the probability to find positions with lower states of internal energy than the initial positions. The SANN algorithm replaces the current solution with a randomly generated new solution. Better solutions are accepted deterministically, where worse solutions are accepted with a probability that depends on the difference between the corresponding function values and on a global parameter, which is commonly referred to as the *temperature*. The algorithm parameter `temp` specifies the initial temperature of the SANN algorithm. The temperature is gradually decreased during the optimization. A second parameter, `tmax`, is used to model this cooling scheme.

We consider the R implementation of SANN, which is available via the general-purpose optimization function `optim()` from the R package `stats`, which is part of every R installation. The function `optim()` is parametrized as follows

```
optim(par, fn, gr = NULL, ..., method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B",
    "SANN", "Brent"), lower = -Inf, upper = Inf, control = list(), hessian = FALSE)
```

Here, `par` denotes *initial values for the parameters to be optimized over*. Note, the problem dimension it specified by the length of this vector, so `par`=c(1,1,1,1) denotes a four-dimensional optimization problem. `fn` is a *function to be minimized* (or maximized), with first argument the vector of parameters over which minimization is to take place. `gr` defines a *function to return the gradient* for the"`BFGS`", "`CG`" and "`L-BFGS-B`" methods. If it is `NULL`, a finite-difference approximation will be used. For the SANN method it specifies a function to generate a new candidate point. If it is `NULL`, a default Gaussian Markov kernel is used. The symbol ... represents *further arguments* (optional) that can be be passed to `fn` and `gr`. The parameter `method` denotes the *optimization method* to be used. Here, we will use the parameter value SANN.

The parameters `lower, upper` specify *bounds* on the variables for the "L-BFGS-B" method, or bounds in which to search for method "`Brent`". So, we will not use these variables in our examples. The argument `control` defines a relatively long *list of control parameters*. We will use the following parameters from this list:

1. `maxit`, i.e., the *maximum number of iterations*, which is for SANN the maximum number of

function evaluations. This is the stopping criterion.

2. `temp` controls the SANN algorithm. It is the *starting temperature* for the cooling schedule with a default value of 10.

3. Finally, we will use `tmax`, which is the number of *function evaluations at each temperature* for the SANN method. Its default value is also 10.

To obtain reproducible results, we will set the random number generator (RNG) seed. Using a two-dimensional objective function (sphere) and the starting point (initial values for the parameters to be optimized over) $(10, 10)$, we can execute the optimization runs as follows:

```
set.seed(123)
resSANN <- optim(c(10, 10), sphere, method = "SANN", control = list(maxit = 100,
    temp = 10, tmax = 10))
resSANN

## $par
## [1] 4.835178 4.664964
##
## $value
## [1] 45.14084
##
## $counts
## function gradient
##      100       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The best, i.e., smallest, function value, which was found by SANN, reads 45.14084. The corresponding point in the search space is approximately (4.835178, 4.664964). No gradient information was used and one hundred function evaluations were performed. The variable `convergence` is an integer code, and its value 0 indicates successful completion of the SANN run. No additional `message` is returned.

Now that we have performed a first run of the SANN algorithm on our simple test function, we are interested in improving SANN's performance. The SANN heuristic requires some parameter settings, namely `temp` and `tmax`. If these values are omitted, a default value of ten is used. The questions is: Are the default algorithm parameter settings, namely `temp` =10 and `tmax` =10, adequate for SANN or can these values be improved? That is, we are trying to tune the SANN optimization algorithm.

A typical beginner in algorithm tuning would try to improve the algorithm's performance by manually increasing or decreasing the algorithm parameter values, e.g., choosing `temp` = 20 and `tmax` = 5.

```
set.seed(123)
resSANN <- optim(par = c(10, 10), fn = sphere, method = "SANN", control = list(maxit = 100,
    temp = 20, tmax = 5))
resSANN

## $par
## [1] 6.163905 6.657100
##
## $value
## [1] 82.3107
##
## $counts
## function gradient
##      100       NA
##
```

```
## $convergence
## [1] 0
##
## $message
## NULL
```

Obviously, the manual "tuning" step worsened the result. And, this procedure is very time consuming and does not allow efficient statistical conclusions. Therefore, we will present a different approach, which uses the SPOT. Although the setup for the tuning procedure with SPOT is very similar to the setup discussed in this section, it enables deeper insights into the algorithm's performance.

5 Level 3: SPOT

This section presents an example, which demonstrates, how SPOT can be used to tune the SANN algorithm defined at level 2. The goal of this tuning procedure is to determine improved parameter settings for the SANN algorithm. As a level 1 function, which will be optimized by SANN, the sphere function was chosen.

Example 1 (Using Random Forest). *First, the problem setup for Level 1 has to be specified. To keep the situation as simple as possible, we will use the sphere test function, which is defined in Equation (1). The problem design requires the specification of the starting point* `x0` *for the search.*

```
x0 = c(-1, 1, -1)
```

Since `x0` *has three elements, we are facing a three dimensional optimization problem.* SANN *will be used to determine its minimum function value.*

Secondly, the problem setup for level 2 has to be defined. Again, several settings have to be specified for the SANN *algorithm to be tuned. The budget, i.e., the maximum number of function evaluations that can be used by* SANN *is specified via*

```
maxit = 100
```

As above, the R *implementation of* SANN *will be used via the* `optim()` *function. We will consider two parameters:*

1. *the initial temperature (*`temp`*) and*
2. *the number of function evaluations at each temperature (*`tmax`*).*

Both are integer values. All parameters and settings of SANN *which were used for this simple example are summarized in Table 1.*

Thirdly, the tuning procedure at level L3 has to be specified. To interface SANN *with* SPOT*, a wrapper function* `sann2spot()` *is used. Note,* SPOT *uses matrices as the basic data structure. The matrix format was chosen as a compromise between speed and flexibility. The* `matrix` *command can be used as follows:*

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

The interface function receives a matrix where each row is proposed parameter setting (`temp`*,* `tmax`*), and each column specifies the parameters. It generates a* $(n, 1)$*-matrix as output, where* n *is the number of (*`temp`*,* `tmax`*) parameter settings.*

```
sann2spot <- function(algpar) {
    performance <- NULL
    for (i in 1:nrow(algpar)) {
        resultList <- optim(par = c(10, 10), fn = sphere, method = "SANN", control = list(maxit = 100,
            temp = algpar[i, 1], tmax = algpar[i, 2]))
        performance <- c(performance, resultList$value)
    }
    return(matrix(performance, , 1))
}
```

Table 1: SANN parameters. The first two parameters belong to the algorithm design, whereas the remaining parameters are from the problem design. Note, the starting point defines the problem dimension, i.e., by specifying a three dimensional starting point the problem dimension is set to three. The initial seed is the value that the RNG is initialized with.

Name	Symbol	Factor name
Initial temperature	t	`temp`
Number of function evaluations at each temperature	$t_{\max}$	`tmax`
Starting point	$\vec{x_0} = (-1, 1, -1)$	`x0`
Problem dimension	$n = 3$	
Objective function	sphere	`sphere()`
Quality measure	Expected performance, e.g., $E(y)$	`y`
Initial seed	s	`1`
Budget	maxit = 100	`maxit`

Now we can test the interface. First, we run SANN *with* `temp` *= 10 and* `tmax` *= 10. A second* SANN *run is performed using* `temp` *= 20 and* `tmax` *= 5.*

```
set.seed(1)
sann2spot(algpar = matrix(c(10, 10), 1))

##           [,1]
## [1,] 8.224742

set.seed(1)
sann2spot(algpar = matrix(c(20, 5), 1))

##           [,1]
## [1,] 22.53343
```

SPOT itself has various parameters that need to be configured so that it can solve the tuning problem efficiently. A configuration or `control` *list can be defined for SPOT. If no configuration list is specified, the default configuration that works fine for many problems, is used.*

In the following, some elements of the configuration list that are important for our example (tuning SANN *+ sphere) will be explained.*

- ***types:*** *Since* SANN*'s parameters* `temp` *and* `tmax` *are integers, we provide this type information via* `types = c("integer", "integer")`*.*
- ***funEvals:*** *The number of algorithm runs, i.e., runs of the* SANN *algorithm, is specified via* `funEvals`*.*
- ***noise:*** SANN *is a stochastic optimizer, so we specify* `noise = TRUE`
- ***seedFun:*** *Also due to the stochasticity of the optimizer, a random number generator seed has to be specified,* `seedFun = 1`*. Every time an algorithm configuration is tested, the RNG seed will be set. For the first evaluation, the seed will be* `seedFun`*, subsequent evaluations will increment this value.*
- ***replicates:*** *Since our evaluations are subject to noise, we can make replicates to mitigate it. In our example, each algorithm configuration will be evaluated twice, so the setting* `replicates = 2` *is used.*
- ***seedSPOT:*** *An additional seed for* SPOT *can be specified using* `seedSPOT = 1`*. This second seed is only set once in the beginning. This ensures that the* SPOT *run is reproducible, since* SPOT *itself may also be of a stochastic nature (depending on the configuration).*

- ***design:*** *The* `design` *parameter defines the method to be used to generate an initial design (a set of initial algorithm settings (here: a number of pairs of* `temp` *and* `tmax`*). A Latin hyper cube design (LHD) is specified via* `design = designLHD`*.*
- ***model:*** *Based on the initial design and subsequent evaluations, a* `model` *can be trained to learn the relation between algorithm parameters (*`temp`*,* `tmax`*) and algorithm performance. To generate the meta* `model`*, we use a random forest implementation Breiman (2001). This can be specified via* `model = buildRandomForest`*. Random forest was chosen, because it is a robust method which can handle categorical and numerical variables.*
- ***optimizer:*** *Once a* `model` *is trained, we need an* `optimizer` *to find the best potential algorithm configuration, based on the* `model` *Here, we choose a very simple optimizer, which creates a large LHD and evaluates it on the* `model`*,* `optimizer = optimLHD`*.*
- ***optimizerControl:*** *The specified* `optimizer` *may have options that need to be set. Here, we only specify the number of* `model` *evaluations to be performed by the* `optimizer` *with* `optimizerControl = list(funEvals=1000)`*.*

Overall, we obtain the following configuration:

```
spotConfig <- list(
  types = c("integer", "integer"), #data type of tuned parameters
  funEvals = 50, #maximum number of SANN runs
  noise = TRUE, #problem is noisy (SANN is non-deterministic)
  seedFun = 1, #RNG start seed for algorithm calls (iterated)
  replicates = 2, #2 replicates for each SANN parameterization
  seedSPOT = 1, #main RNG
  design = designLHD, #initial design: Latin Hypercube
  model = buildRandomForest, # model = buildKriging Kriging surrogate model
  optimizer = optimLHD, #Use LHD to optimize on model
  optimizerControl = list(funEvals=100) #100 model evals in each iteration
)
```

The region of interest *(ROI) specifies* SPOT*'s search intervals for the* SANN *parameters, i.e., for* `tmax` *and* `temp`*. Here, both parameters* `temp` *and* `tmax` *will be tuned in the region between one and 100.*

```
tempLo = 1
tempHi = 100
tmaxLo = 1
tmaxHi = 100
lower = c(tempLo, tmaxLo)
upper = c(tempHi, tmaxHi)
```

The order (first `temp` *then* `tmax`*) has to be the same as in the* `sann2spot` *interface. Now we are ready to perform our first experiment. We will start* SPOT *via* `spot()`*. The result is stored in a list, e.g.,* `resRf`*.*

```
resRf <- spot(x = NULL, fun = sann2spot, lower = lower, upper = upper, control = spotConfig)
```

Now, we are able to take a look at the results from the tuning procedure. Output from the SPOT *run, which is stored in the list* `resRf`*, has the following structure.*

```
str(resRf)

## List of 7
##  $ xbest   : num [1, 1:2] 3 69
##  $ ybest   : num [1, 1] 0.00198
##  $ x       : num [1:50, 1:2] 43 61 37 29 98 18 85 5 79 57 ...
##  $ y       : num [1:50, 1] 4.03 103.08 64.03 10.14 27.5 ...
##  $ count   : int 50
##  $ msg     : chr "budget exhausted"
##  $ modelFit:List of 3
```

```
##   ..$ rfFit:List of 17
##   .. ..$ call           : language randomForest(x = x, y = y)
##   .. ..$ type           : chr "regression"
##   .. ..$ predicted      : num [1:48] 28.5 45.7 30.7 10.1 43.9 ...
##   .. ..$ mse            : num [1:500] 297 538 475 367 340 ...
##   .. ..$ rsq            : num [1:500] 0.4163 -0.0588 0.0663 0.2783 0.3311 ...
##   .. ..$ oob.times      : int [1:48] 183 160 167 188 192 187 168 177 170 182 ...
##   .. ..$ importance     : num [1:2, 1] 12109 10797
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "1" "2"
##   .. .. .. ..$ : chr "IncNodePurity"
##   .. ..$ importanceSD   : NULL
##   .. ..$ localImportance: NULL
##   .. ..$ proximity      : NULL
##   .. ..$ ntree          : num 500
##   .. ..$ mtry           : num 1
##   .. ..$ forest         :List of 11
##   .. .. ..$ ndbigtree    : int [1:500] 25 29 27 19 23 29 29 23 27 19 ...
##   .. .. ..$ nodestatus   : int [1:35, 1:500] -3 -3 -3 -1 -3 -3 -1 -1 -1 -3 ...
##   .. .. ..$ leftDaughter : int [1:35, 1:500] 2 4 6 0 8 10 0 0 0 12 ...
##   .. .. ..$ rightDaughter: int [1:35, 1:500] 3 5 7 0 9 11 0 0 0 13 ...
##   .. .. ..$ nodepred     : num [1:35, 1:500] 10.14 52.32 2.94 86.03 46.71 ...
##   .. .. ..$ bestvar      : int [1:35, 1:500] 2 2 2 0 2 1 0 0 0 1 ...
##   .. .. ..$ xbestsplit   : num [1:35, 1:500] 59.5 9.5 73 0 17.5 21 0 0 0 8.5 ...
##   .. .. ..$ ncat         : num [1:2] 1 1
##   .. .. ..$ nrnodes      : int 35
##   .. .. ..$ ntree        : num 500
##   .. .. ..$ xlevels      :List of 2
##   .. .. .. ..$ : num 0
##   .. .. .. ..$ : num 0
##   .. ..$ coefs          : NULL
##   .. ..$ y              : num [1:48, 1] 4.03 103.08 64.03 10.14 27.5 ...
##   .. ..$ test           : NULL
##   .. ..$ inbag          : NULL
##   .. ..- attr(*, "class")= chr "randomForest"
##   ..$ x    : num [1:48, 1:2] 43 61 37 29 98 18 85 5 79 57 ...
##   ..$ y    : num [1:48, 1] 4.03 103.08 64.03 10.14 27.5 ...
##   ..- attr(*, "class")= chr "spotRandomForest"
```

SPOT *generates many information which can be used for a statistical analysis. For example, the best configuration found can be displayed as follows.*

```
cbind(resRf$xbest, resRf$ybest)
```

```
##      [,1] [,2]        [,3]
## [1,]    3   69 0.001980789
```

SPOT *recommends using* `temp = 3` *and* `tmax = 69`*. These parameter settings differ significantly from the default values. These values also make sense: The starting temperature* `temp` *should be low and the number of evaluations at each temperature* `tmax` *should be high. Hence, worse solutions will rarely be accepted by* SANN*, which leads to a very localized search. This is a good configuration for this example, since the sphere function is unimodal.* □

6 Details

6.1 Details 1: The spot() Interface

SPOT uses the same interface as R's standard `optim()` function, which uses the arguments reported

Table 2: SPOT arguments

name	description
`x`	Optional start point (or set of start points), specified as a matrix. One row for each point, and one column for each optimized parameter.
`fun`	Objective function. It should receive a matrix x and return a matrix y. In case the function uses external code and is noisy, an additional seed parameter may be used, see the `control$seedFun` argument in the function documentation for details.
`lower`	Vector that defines the lower boundary of search space.
`upper`	Vector that defines the upper boundary of search space.
`control`	List of additional settings

Table 3: SPOT return values

name	description	type
`xbest`	Parameters of the best found solution	`matrix`
`ybest`	Objective function value of the best found solution	`matrix`
`x`	Archive of all evaluation parameters	`matrix`
`y`	Archive of the respective objective function values	`matrix`
`count`	Number of performed objective function evaluations	`integer`
`msg`	Message specifying the reason of termination	`character`
`modelFit`	The fit of the model from the last SPOT iteration, i.e., an object returned by the last call to the function specified by `control$model`	`list`

in Table 2.

NOTE: take care of consistency of `upper`, `lower` and, if specified, `x`. In cases of inconsistency, the dimension of the variable `lower` will be taken into account to establish the dimension of the problem. The function `spot()` returns a list with the values shown in Table 3.

6.2 Details 2: SPOT Configuration

SPOT configuration settings are listed in Table 4. The configuration list stores information about SPOT specific settings. All settings not specified in the list will be set to their default values.

6.3 Details 3: Initial Designs

Problem design: For example, the starting point `x0 = (-1,1,-1)` belongs to the problem design. A *region of interest* (ROI) specifies algorithm parameters and associated lower and upper bounds for the algorithm parameters.

- Values for `temp` are chosen from the interval $[1; 100]$.
- Values for `tmax` are chosen from the interval $[1; 100]$.

SPOT implements a sequential approach, i.e., the available budget is not used in one step. Rather, sequential steps are made, comprised of model training, optimization, and evaluation. To initialize this procedure, some first data set is required to train the first, coarse-grained meta model.

Example 2 (Using ten function evaluations). *For example, the total number of* SANN *algorithm runs, i.e., the available budget, can be set to 10 using* `funEvals = 10`*, the size of the initial design can be modified via* `designControl$size`*, the number of repeated evaluations of the initial design can be modified via* `designControl$replicates`*, and the number of replicates used for each new algorithm design point suggested by* SPOT *can be modified via* `replicates`*:*

```
spotConfig10 <- list(funEvals = 10, designControl = list(size = 6, replicates = 1),
    noise = TRUE, seedFun = 1, seedSPOT = 1, replicates = 2, model = buildRandomForest)
```

Using this configuration, the budget will be spent as follows: Six initial algorithm design points (parameter sets) will be created, and each will be evaluated just once. Afterwards, SPOT *will have a remaining*

Table 4: SPOT configuration

name	description	default
`funEvals`	Budget of function evaluations (spot uses no more than funEvals evaluations of fun).	`20`
`types`	Vector of data type of each variable as a string.	`"numeric"`
`design`	A function that creates an initial design of experiment. Functions that accept the same parameters, and return a matrix like `designLHD` or `designUniformRandom` can be used.	`designLHD`
`designControl`	List of controls passed to the `control` list of the `design` function.	empty `list`
`model`	Function that builds a model of the observed data. Functions that accept the same parameters, and return a matrix like `buildKriging` or `buildRandomForest` can be used.	`buildKriging`
`modelControl`	List of controls passed to the `control` list of the `model` function.	empty `list`
`optimizer`	Function that is used to optimize the `model`, finding the most promising candidate solutions. Functions that accept the same parameters, and return a matrix like `optimLHD` or `optimLBFGSB` can be used.	`optimLHD`
`optimizerControl`	List of controls passed to the `control` list of the `optimizer` function.	empty `list`
`noise`	Boolean, whether the objective function has noise.	`FALSE`
`OCBA`	Boolean, indicating whether Optimal Computing Budget Allocation (OCBA) should be used in case of a noisy objective function. OCBA controls the number of replications for each candidate solution. Note, that `replicates` should be larger than one in that case, and that the initial experimental design (see `design`) should also have replicates larger than one.	`FALSE`
`OCBAbudget`	Number of objective function evaluations that OCBA can distribute in each iteration.	`3`
`replicates`	Number of times a candidate solution is initially evaluated, that is, in the initial design, or when created by the optimizer.	`1`
`seedFun`	Initial seed for the objective function in case of noise. The default means that no seed is set. The user should be very careful with this setting. It is intended to generate reproducible experiments for each objective function evaluation, e.g., when tuning non-deterministic algorithms. If the objective function uses a constant number of random number generations, this may be undesirable. Note, that this seed is by default set prior to each evaluation. A replicated evaluation will receive an incremented value of the seed. Sometimes, the user may want to call external code using random numbers. To allow for that case, the user can specify an objective function (`fun`), which has a second parameter `seed`, in addition to first parameter (matrix `x`). This seed can then be passed to the external code, for random number generator initialization. See end of examples section in the documentation of `spot()` for a demonstration.	`NA`
`seedSPOT`	Value used to initialize the random number generator. It ensures that experiments are reproducible.	`1`
`duplicate`	In case of a deterministic (non-noisy) objective function, this handles duplicated candidate solutions. By default (`duplicate = "EXPLORE"`), duplicates are replaced by new candidate solutions, generated by random sampling with uniform distribution. If desired, the user can set this to `"STOP"`, which means that the optimization stops and results are returned to the user (with a warning). This may be desirable, as duplicates can be a indicator of convergence, or problems with the configuration. In case of noise, duplicates are allowed regardless of this parameter.	`"EXPLORE"`
`plots`	Logical. Should the progress be tracked by a line plot?	`FALSE`

budget of four evaluations. These can be spent on sequentially testing two additional design points. Each of those will be evaluated twice.

```
res10 <- spot(, fun = sann2spot, lower = lower, upper = upper, control = spotConfig10)
```

Results from these runs can be displayed as follows. The first two columns show the settings of the algorithm parameters `temp` *and* `tmax`*, respectively. The third row shows the corresponding function values.*

```
cbind(res10$x, res10$y)
```

```
##             [,1]     [,2]         [,3]
##  [1,] 25.454322 79.91839 13.429821426
##  [2,] 93.392836 10.12510 88.943521433
##  [3,]  9.143432 42.74037  0.011436182
##  [4,] 70.072590 30.52438 74.191697318
##  [5,] 47.651660 50.88496 57.968961980
##  [6,] 61.529701 91.37430 11.290122825
##  [7,] 11.598664 91.19592  0.008496108
##  [8,] 11.598664 91.19592  0.050744463
##  [9,] 12.436667 90.58432  0.008496108
## [10,] 12.436667 90.58432  0.426068154
```

The results show the desired outcome: the first six configurations are unique, the following four contain replications (two identical settings, but with different stochastic result). □

`designLHD()` is the default setting to generate designs. A simple one-dimensional design with values from the interval $[-1, 1]$ can be generated as follows:

```
designLHD(, -1, 1)
```

```
##              [,1]
##  [1,]  0.26479322
##  [2,]  0.91469238
##  [3,] -0.74340436
##  [4,]  0.41414078
##  [5,]  0.11284640
##  [6,] -0.41965779
##  [7,] -0.92411596
##  [8,]  0.76315614
##  [9,] -0.25867274
## [10,] -0.03293109
```

A more complicated design, which consists of numeric values for the first variable in the range from -1 to +1, of integer values for the second variable in the range from -2 to 4, of integer values for the third variable in the range from 1 to 9, and with two factors 0 and 1 for the fourth variable, can be generated as follows:

```
designLHD(, c(-1, -2, 1, 0), c(1, 4, 9, 1), control = list(size = 5, retries = 100,
    types = c("numeric", "integer", "factor", "factor")))
```

```
##            [,1] [,2] [,3] [,4]
## [1,] -0.7755956    1    3    0
## [2,]  0.1777632    2    8    1
## [3,]  0.8543608    0    1    1
## [4,]  0.4930152    4    5    0
## [5,] -0.4011451   -1    7    0
```

Designs can also be combined as illustrated in the following example. The corresponding plot (Fig. 3) shows the resulting design.

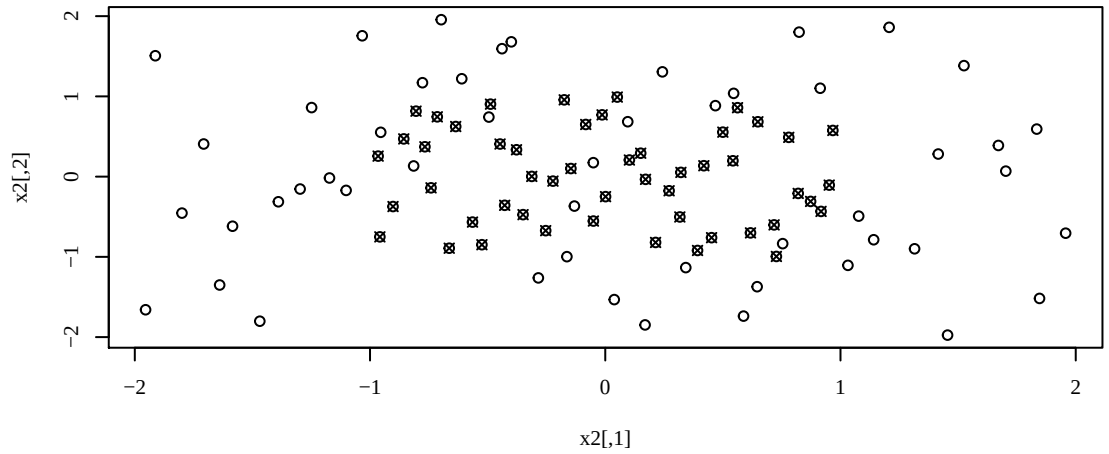


Fig. 3: Combination of two designs based on Latin Hypercube Sampling.

```
set.seed(123)
x1 <- designLHD(, c(-1, -1), c(1, 1), control = list(size = 50, retries = 100))
x2 <- designLHD(x1, c(-2, -2), c(2, 2), control = list(size = 50, retries = 100))
plot(x2, pch = 1)
points(x1, pch = 4)
```

Designs based on uniform random sampling can be generated with the function `designUniformRandom()` as follows:

```
designUniformRandom(, c(-1, 0), c(1, 10), control = list(size = 5))

##              [,1]      [,2]
## [1,] -0.59701202 3.6807802
## [2,]  0.95638068 0.4767697
## [3,] -0.17457507 4.5883452
## [4,] -0.04937504 1.3611036
## [5,] -0.96457920 4.8153870
```

6.4 Details 4: Models

In SPOT, a meta model or surrogate model is used to determine promising algorithm design points. To that end, it aims to learn the relation between algorithm parameters and the corresponding algorithm performance. The different models and their options are described next.

6.4.1 Kriging

As default, the `buildKriging` function is used for modeling. This function builds a Kriging model (also known as Gaussian process regression) loosely based on code by Forrester et al. (2008). Kriging models are based on measures of similarity, or kernels. Here, a Gaussian kernel is used:

$$k(x, x') = \exp\left(-\sum_{i=1}^{n} \theta_i |x_i - x_i'|^{p_i}\right)$$

By default exponents are fixed at a value of two, $p_i = 2 \forall i$, and the nugget effect (or regularization constant) is used. To correct the uncertainty estimates in case of using the nugget effect, re-interpolation is also by default turned on (see Forrester et al. (2008) for more details on these features of the model).

The following code exemplifies how a Kriging model is built with an artificial data set. Note, this example exemplifies how SPOT can be used as a surrogate model based optimization algorithm, i.e., no algorithm is tuned.

```
## Test-function:
braninFunction <- function(x) {
    (x[2] - 5.1/(4 * pi^2) * (x[1]^2) + 5/pi * x[1] - 6)^2 + 10 * (1 - 1/(8 *
        pi)) * cos(x[1]) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(20) * 15 - 5, runif(20) * 15)
## Compute observations at design points (for Branin function)
y <- as.matrix(apply(x, 1, braninFunction))
## Create model with default settings
fit <- buildKriging(x, y, control = list(algTheta = optimLHD))
## Print model parameters
print(fit)

## ------------------------
## Forrester Kriging model.
## ------------------------
## Estimated activity parameters (theta) sorted
## from most to least important variable
## x1  x2
## 6.90722 0.4533334
##
## exponent(s) p:
## 2
##
## Estimated regularization constant (or nugget) lambda:
## 1.098503e-05
##
## Number of Likelihood evaluations during MLE:
## 600
## ------------------------

## Define a new location
newloc <- matrix(c(1, 2), nrow = 1)
## Predict at new location
predict(fit, newloc)

## $y
## [1] 22.29809

## True value at location
braninFunction(newloc)

## [1] 21.62764

##
```

6.4.2 Handling Factor Variables in the Kriging Model

Sometimes, parameters optimized or modeled by SPOT functions will not be numerical, but rather categorical. This may, e.g., occur if an evolutionary algorithm is tuned: while some parameters like mutation rates may be real valued, the selection between different mutation operators may be a categorical parameter. Hence, if dimension x_i of a parameter configuration x is a factor variable (see parameter `types`), Hamming distance, that determines the number of positions at which the corresponding values

are different, will be used instead of $|x_i - z_i|$. To illustrate how factor variables can be handled, we create a test function that uses a factor variable. Here, the third dimension x_3 is categorical.

```
braninFunctionFactor <- function(x) {
    y <- (x[2] - 5.1/(4 * pi^2) * (x[1]^2) + 5/pi * x[1] - 6)^2 + 10 * (1 -
        1/(8 * pi)) * cos(x[1]) + 10
    if (x[3] == 1)
        y <- y + 1 else if (x[3] == 2)
        y <- y - 1
    y
}
```

To test how this affects our model, we first generate some training data and fit the model with default settings, which ignores factor information and uses the standard kernel.

```
set.seed(1)
## Replace x with new data
x <- cbind(runif(50) * 15 - 5, runif(50) * 15, sample(1:3, 50, replace = TRUE))
##
y <- as.matrix(apply(x, 1, braninFunctionFactor))
fitDefault <- buildKriging(x, y, control = list(algTheta = optimLBFGSB))
```

Afterwards we fit the model, which includes information about the factor variable:

```
fitFactor <- buildKriging(x, y, control = list(algTheta = optimLBFGSB, types = c("numeric",
    "numeric", "factor")))
```

We generate some new, unseen data for testing and perform some predictions with both models.

```
## Replace xtest with new data
xtest <- cbind(runif(200) * 15 - 5, runif(200) * 15, sample(1:3, 200, replace = TRUE))
##
ytest <- as.matrix(apply(xtest, 1, braninFunctionFactor))
## Predict test data with both models, and compute error
ypredDef <- predict(fitDefault, xtest)$y
ypredFact <- predict(fitFactor, xtest)$y
mean((ypredDef - ytest)^2)

## [1] 4.530052

mean((ypredFact - ytest)^2)

## [1] 1.514662
```

The error of the factor-aware model is lower. This demonstrates that users should make sure to declare the nature of the modeled variables correctly, via the `types` variable.

6.5 Details 5: Optimization on the Meta Model

Minimization by *Latin hyper cube sampling* (LHS) is the default optimizer used for finding the next algorithm design parameters on the meta model. The LHS procedure generates a set of design points which is called a *Latin hyper cube design* (LHD). The function `optimLHD()` uses LHS to optimize a specified target function as follows: A Latin hyper cube Design (LHD) is created with `designLHD()` , then evaluated by the objective function. All results are reported, including the best (minimal) objective value, and corresponding design point. A standalone optimization run using `optimLHD` can be implemented as follows. It uses 100 design points as a default value.

```
resOptimumLHD <- optimLHD(, fun = funSphere, lower = c(-10, -20), upper = c(20,
    8))
str(resOptimumLHD)

## List of 6
##  $ x    : num [1:100, 1:2] 2.55 9.62 10.73 10.49 1.43 ...
```

```
##  $ y    : num [1:100, 1] 8.13 92.91 193.1 112.67 234.13 ...
##  $ xbest: num [1, 1:2] 1.145 0.698
##  $ ybest: num [1, 1] 1.8
##  $ count: num 100
##  $ msg  : chr "success"

resOptimumLHD$ybest

##          [,1]
## [1,] 1.799124
```

Using more sophisticated algorithms, as the variable metric algorithm (L-BFGS-B), might lead to better results. However, they are not as robust as the simple `optimLHD` search. Also, L-BFGS-B is a pure local search, which may not be ideal to solve potentially multi-modal tuning problems.

```
resOptimBFGS <- optimLBFGSB(, fun = funSphere, lower = c(-10, -20), upper = c(20,
    8))
resOptimBFGS$ybest

## [1] 1.98805e-30
```

Hence, SPOT also includes interfaces to more sophisticated algorithms, such as differential evolution from `DEoptim` package or various methods included in the `nloptr` package. For further details on these, see e.g., the help of the corresponding functions

```
`?`(optimDE)
`?`(optimNLOPTR)
```

6.6 Details 6: SPOT Loop: Continued Evaluation

Sometimes, users may desire to continue a previously finished SPOT run. We will demonstrate, how SPOT can be restarted, reusing the collected data.

Example 3 (SPOT with continued evaluation). *The surrogate model based optimization setting will be used to exemplify the continued evaluation. The two dimensional sphere function is used as an objective function (level L1) and* SPOT *will be used at level L2.* SPOT *uses 5 function evaluations.*

```
control01 <- list(designControl = list(size = 5, replicates = 1), funEvals = 5)
res1 <- spot(, funSphere, lower = c(-2, -3), upper = c(1, 2), control01)
cbind(res1$x, res1$y)

##            [,1]       [,2]      [,3]
## [1,]  0.6316685 -0.3315333 0.5089193
## [2,] -1.9919658  1.7942399 7.1872244
## [3,] -1.1705672 -1.8920564 4.9501050
## [4,]  0.3218145 -2.2762891 5.2850564
## [5,] -0.5957906  0.4112744 0.5241131
```

Now, we continue with a larger budget. If we would like to add 3 function evaluations, the total number of function evaluations is $5 + 3 = 8$. *To continue a* SPOT *run, the command* `spotLoop()` *can be used as follows:* `spotLoop(x, y, fun, lower, upper, control, ...)`. *The arguments are*

- *x: the known candidate solutions that the SPOT loop is started with, specified as a matrix. One row for each point, and one column for each optimized parameter.*
- *y: the corresponding observations for each solution in x, specified as a matrix. One row for each point.*
- *fun: is the objective function. It should receive a matrix x and return a matrix y.*

- *lower: a vector that defines the lower boundary of search space. This determines also the dimensionality of the problem.*
- *upper: a vector that defines the upper boundary of search space.*
- *control: a list with control settings for spot.*
- *...: additional parameters passed to fun.*

```
control01$funEvals <- 8
res2 <- spotLoop(res1$x, res1$y, funSphere, lower = c(-2, -3), upper = c(1,
    2), control01)
cbind(res2$x, res2$y)

##             [,1]         [,2]       [,3]
## [1,]  0.6316685 -0.331533262 0.50891934
## [2,] -1.9919658  1.794239861 7.18722443
## [3,] -1.1705672 -1.892056374 4.95010495
## [4,]  0.3218145 -2.276289054 5.28505643
## [5,] -0.5957906  0.411274430 0.52411310
## [6,] -1.8502780  0.025438423 3.42417578
## [7,]  0.3335189  0.311271847 0.20812503
## [8,]  0.1119788  0.009135746 0.01262271
```

□

7 Exploratory Fitness Landscape Analysis

7.1 Introduction to SPOT's Plot Functions

The SPOT package offers three plot functions that can be used to visualize data or evaluate a model's performance creating 2D and 3D surface plots.

- `plotFunction()` plots function objects
- `plotData()` plots data
- `plotModel()` plots model objects, created by `build*` functions from the SPOT package.

7.1.1 plotFunction

The function `plotFunction()` visualizes the fitness landscape. It generates a (filled) contour plot or perspective / surface plot of a function. It has several options for changing plotting styles, colors etc., but default values should be fine in many cases. The basic arguments are:

```
plotFunction(f, lower, upper, type)
```

Example 4 (Plot of the sphere function). *The following code generates a 2D filled contour plot of the sphere function (see Fig. 4). Note that `plotFunction()` requires a function that handles matrix objects, so `funSphere()` is used.*

```
plotFunction(funSphere, rep(-1, 2), rep(1, 2))
```

□

Example 5 (Plotting a user defined function). *Figure 5 shows the plot of the user defined function*

$$f(\vec{x}) = \sum_{i=1}^{n} (x_i^3 - 1). \tag{2}$$

It also illustrates how the color scheme can be modified.

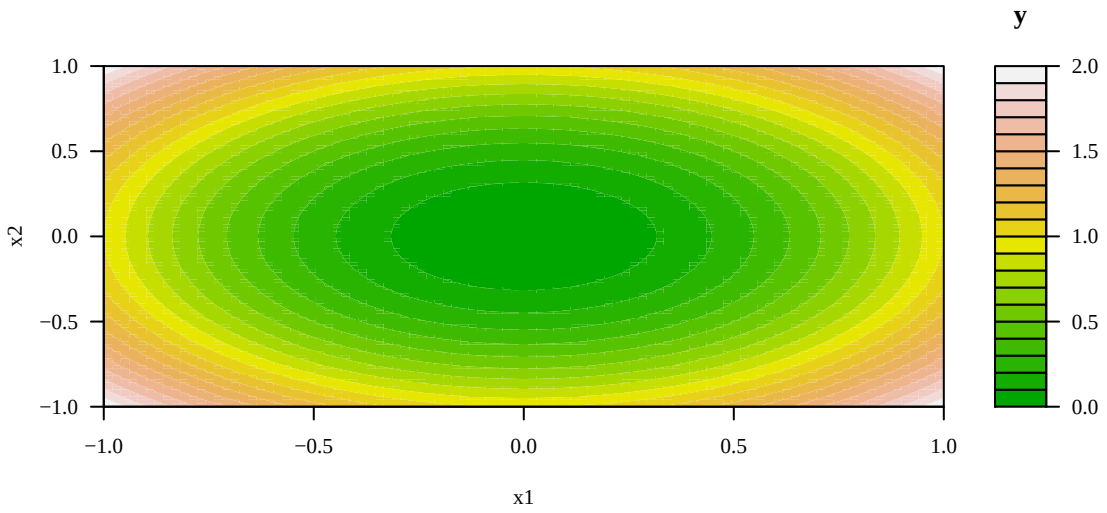


Fig. 4: Example 4: 2D plot of the sphere function generated with `plotFunction()`.

```
myFunction <- function (x){
  matrix(apply(x, # matrix
               1, # margin (apply over rows)
               function(x) sum(x^3-1) # objective function
               ),
         , 1) # number of columns
  }
plotFunction(myFunction,
             rep(-1,2),
             rep(1,2),
             color.palette = rainbow)
```

We can also generate a perspective plot of the user defined function (see Fig. 6):

```
plotFunction(myFunction, rep(-1, 2), rep(1, 2), type = "persp", theta = 10,
    phi = 25, border = NA)
```

□

7.1.2 plotModel

Furthermore, `plotModel()` offers the possibility to visualize models that have already been trained, e.g., during a SPOT run. Some simple examples are given below.

Example 6 (Plotting a trained model). *First, we generate some training data. To demonstrate how plots from data with more than two input dimensions can be generated, we generate a three-dimensional input design, i.e., we consider a functional relationship of the type*

$$f : \mathbb{R}^3 \to \mathbb{R}, \qquad y = f(x_1, x_2, x_3) \tag{3}$$

The output is one dimensional.

```
set.seed(123)
k <- 30
x.test <- designLHD(, rep(-1, 3), rep(1, 3), control = list(size = k))
y.test <- funSphere(x.test)
head(cbind(x.test, y.test))
```

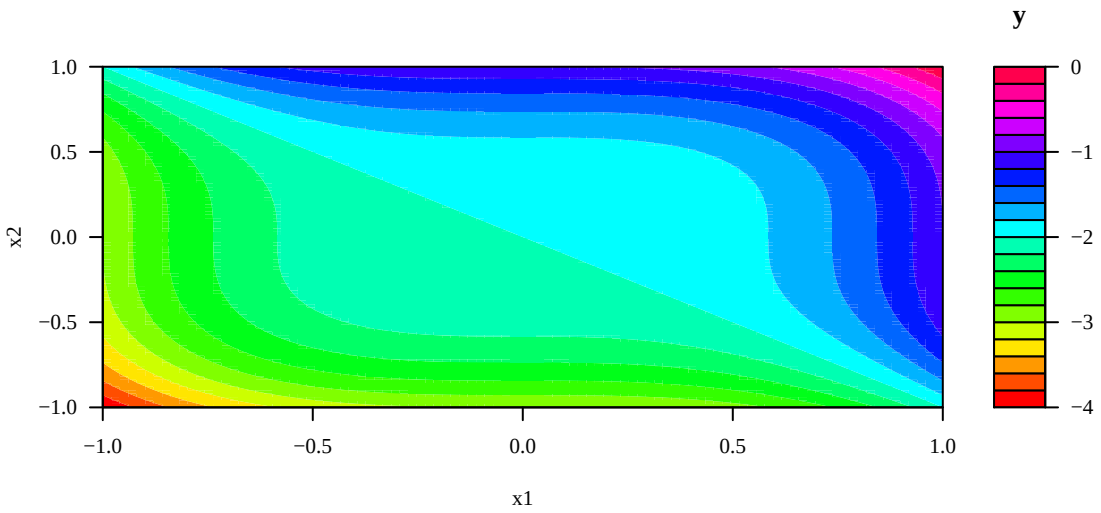


Fig. 5: Example 5. Plotting the user defined function from Equation 2 with plotFunction() using the color palette `rainbow`

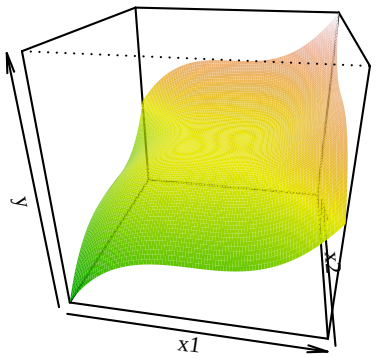


Fig. 6: Example 5: Perspective plot of the user defined function from Equation 2 using plotFunction()

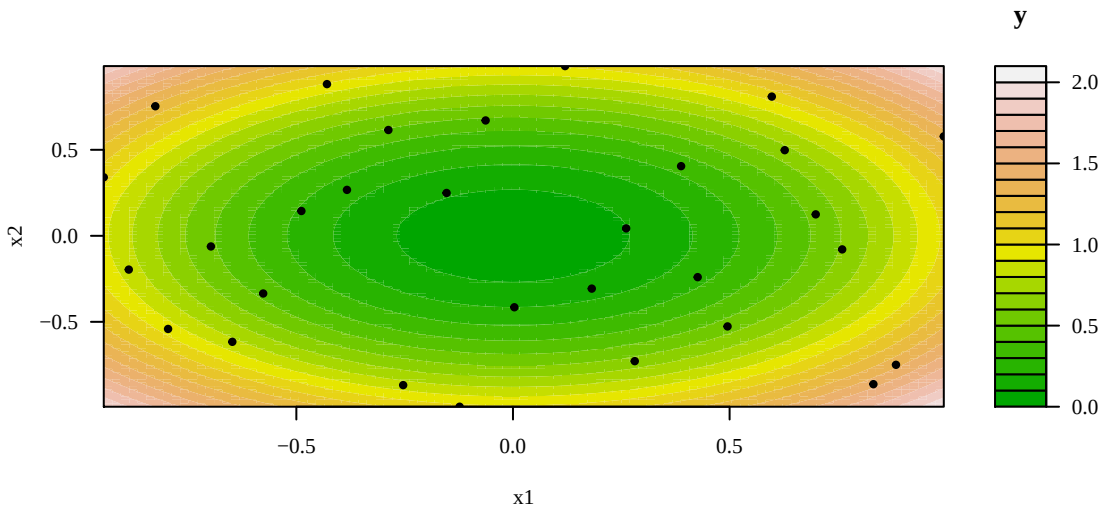


Fig. 7: Example 6: 2D representation of a model using plotModel()

```
##              [,1]       [,2]        [,3]      [,4]
## [1,]  0.698802840  0.1247362  0.57448235 0.8339145
## [2,] -0.383135671  0.2670867 -0.98091431 1.1803211
## [3,]  0.281083653 -0.7285295  0.44918649 0.8115318
## [4,] -0.488374907  0.1442807  0.32227029 0.3631851
## [5,]  0.003177575 -0.4153111  0.08761805 0.1801703
## [6,] -0.153276461  0.2490123 -0.16716223 0.1134440
```

Then, we train a standard response surface using SPOT*'s* `buildRSM()` *function. We generate the default contour plot using* `plotModel()`*.*

```
fit.test <- buildRSM(x.test, y.test)
plotModel(fit.test)
```

Passing the argument `type="contour"` *to the* `plotModel()` *function, a 2D contour plot can be generated as shown in Fig. 8. By default, the dependent variable y is plotted against the first two x_i variables from Equation 3. Note, that the argument* `which` *specifies the independent variables x_i that are plotted. To plot y against x_1 and x_3, the argument* `which=c(1,3)` *can be used.*

```
plotModel(fit.test, which = c(1, 3), type = "contour", pch1 = 24, col1 = "blue")
```

Perspective plots of the same model can be generated as follows. The arguments `theta` *and* `phi` *can be used to modify the view point. The result is shown in Fig. 9.*

```
par(mfrow = c(1, 2))
plotModel(fit.test, type = "persp", border = "NA", theta = 255, phi = 20)
plotModel(fit.test, which = 1:2, type = "persp", border = "NA")
par(mfrow = c(1, 1))
```

□

7.1.3 plotData

Finally, using `plotData()`, different models built on provided data can be compared. The `plotData()` function generates a (filled) contour or perspective plot of a data set with two independent and one dependent variable. The plot is generated by some interpolation or regression model. By default, the loess function is used. Some simple examples are given below.

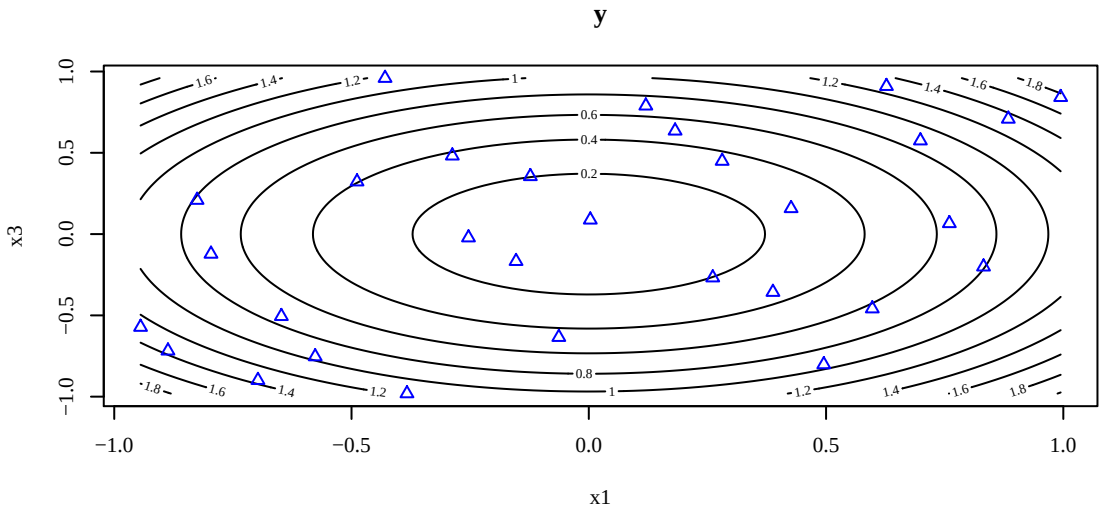


Fig. 8: Example 6: 2D representation of a model using `plotModel()` with the argument `contour`

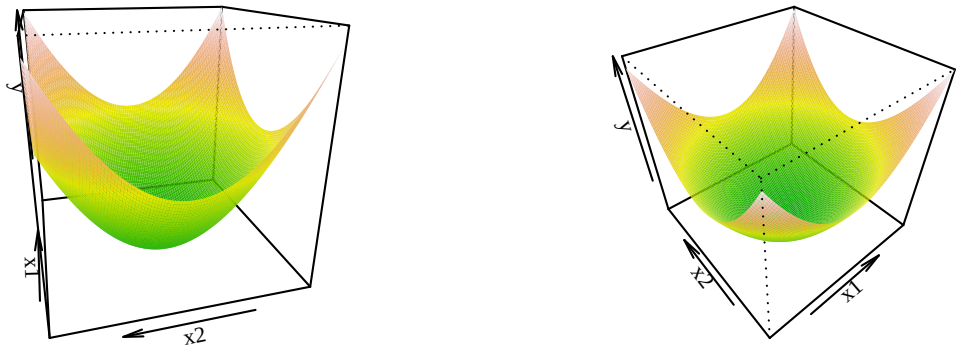


Fig. 9: Example 6: 3D representations of a model using plotModel() with different view points.

Example 7 (Plotting data using loess and random forest). *Figure 10 shows a comparison of two different models fitting the same data.*

```
plotData(x.test, y.test)
plotData(x.test, y.test, type = "filled.contour", cex1 = 1, col1 = "red", pch1 = 21,
    model = buildRandomForest)
```

□

Example 8 (Perspective plot). *Figure 11) shows a perspective plot, which is based on the same data that were used in Example 7.*

```
## Plot 3D figure
plotData(x.test, y.test, type = "persp", border = NA, model = buildLOESS)
```

□

7.2 Exploratory Fitness Landscape Analysis Using SPOT

This section demonstrates how functions from the SPOT package, which were introduced in Sec. 7.1, can be used to perform a visual inspection of the fitness landscape during an interactive SPOT run. These results can also be used to illustrate the algorithm's performance. In this section, reference will be made to the application shown in section 5, in which SPOT is used for the tuning procedure of two SANN design parameters: starting temperature `temp` and the number of function evaluations at each temperature `tmax`.

The fitness landscape can be visualized in two different ways:

1. Because SPOT builds a surrogate model during the sequential optimization, this model can be used to visualize the fitness landscape. In this case, the `plotModel()` function will be used.
2. using standard interpolation or local regression functions. In this case, the `plotData()` function will be used. Note, that the `plotData()` function allows the specification of several interpolation functions (loess is default).

Using `plotFunction()` is usually not applicable, because the underlying (true) analytical function is not known. We consider the SPOT model based approach first.

7.2.1 Plotting the Final Model with plotModel

Plotting the final model from the SPOT run might be the most generic way of visualizing the results, because during the optimization, the optimizer trusted this model. So, why should it be considered unreliable after the optimization is finished? Based on the tuning results from Example 1, we will demonstrate how the final model, which was built during the SPOT run, can be plotted. Since the model is stored in the result list from the SPOT run, i.e., in `resRf`, the parameter `resRf$modelFit` can be passed as an argument to the `plotModel()` function. The result is shown in Fig. 12.

```
plotModel(resRf$modelFit)
```

7.2.2 Plotting the Data with plotData

Results from Example 1 were obtained with the random forest model. But, the data can be fitted to a different model, e.g., a *locally (weighted) scatter plot smoothing* (LOESS) or Kriging model as follows.

Example 9 (Plotting data using loess). *Figure 13 illustrates the LOESS model fit, which uses the data generated with a random forest model.*

```
plotData(resRf$x, resRf$y, model = buildLOESS)
```

□

Example 10 (Plotting data using Kriging). *Plotting the same data as in Fig. 13 with a Kriging model can easily be done. The result is shown in Fig. 14.*

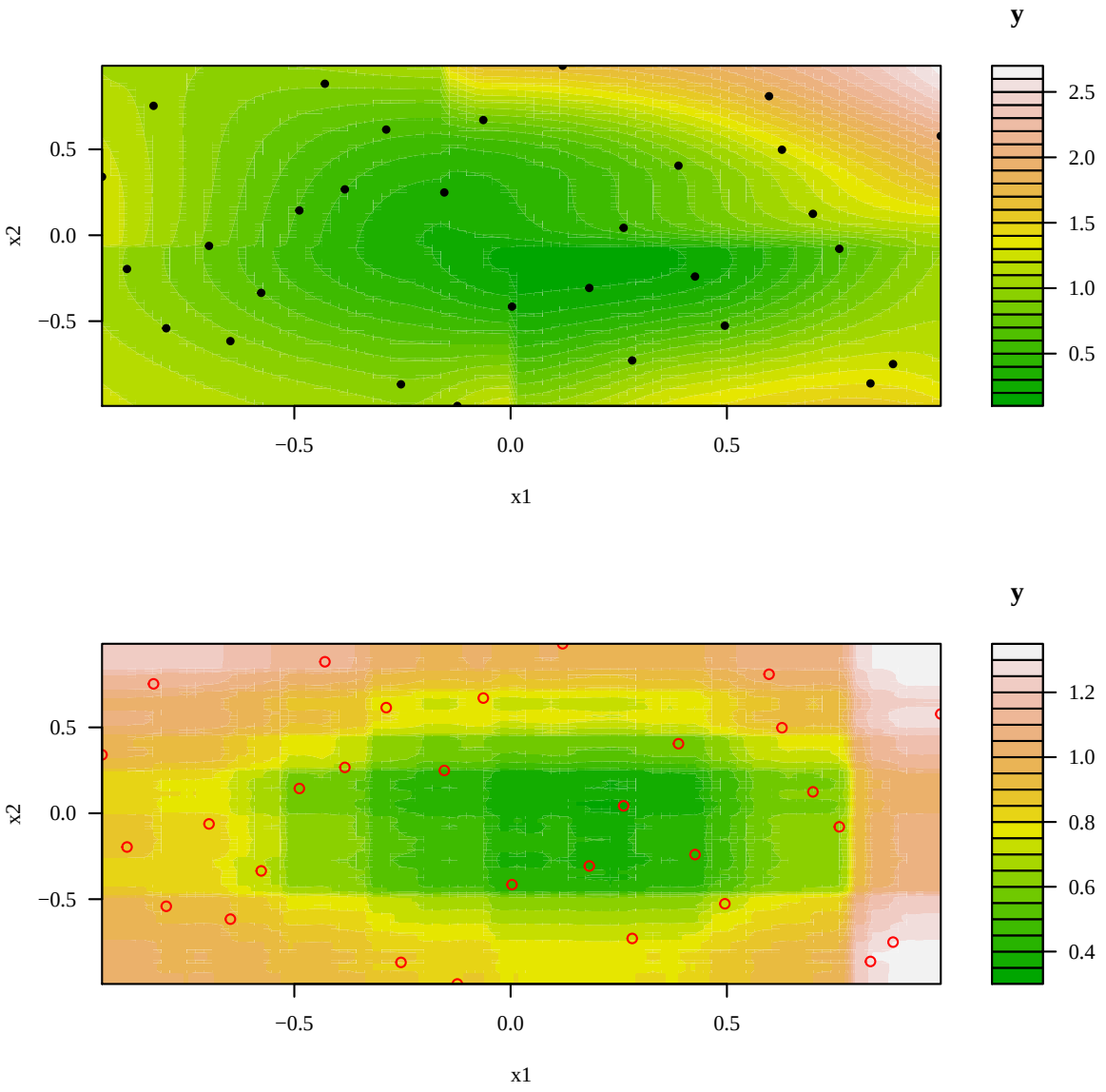


Fig. 10: Example 7: 2D representations using plotData(). Top: The default loess model is used for interpolation. Bottom: random forest was used for interpolation.

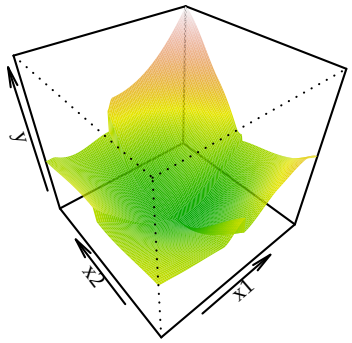


Fig. 11: Example 8: 3D representation of a data set using plotData()

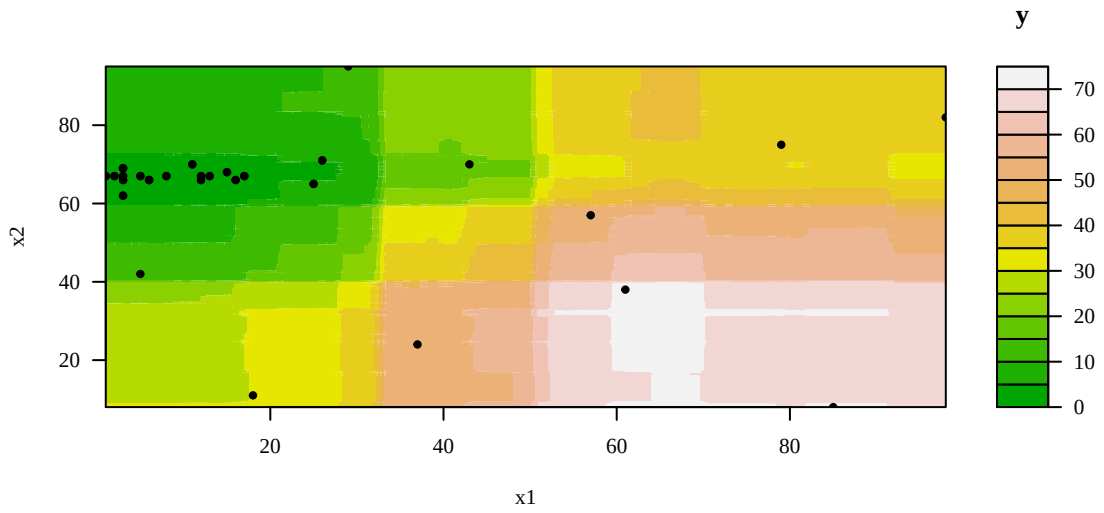


Fig. 12: Example 1: Surface plot from the SPOT run with random forest. The function `plotModel()` was used to generate this plot, based on the random forest model.

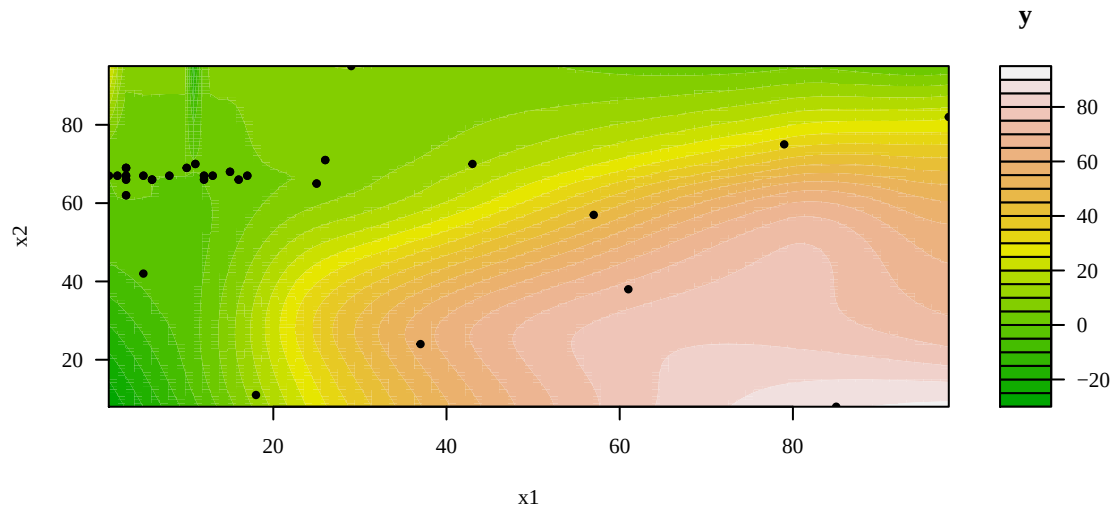


Fig. 13: Example 9: Results from the SPOT run with random forest, visualized with a LOESS model using `plotData()`

```
plotData(resRf$x, resRf$y, model = buildKriging)
```

□

This result can be compared to where a case where Kriging models were used during the SPOT run and for the final illustration.

Example 11. *The same setting as in the random forest based optimization will be used. Only the meta model is changed. The* SPOT *configuration parameters can be changed as follows:*

```
spotConfig$model = buildKriging
spotConfig$optimizer = optimLBFGSB
spotConfig$modelControl = list(algTheta = optimLBFGSB)
## Run SPOT
resK <- spot(x = NULL, fun = sann2spot, lower = lower, upper = upper, control = spotConfig)
```

Now, the Kriging model used during the SPOT *run will be used to visualize the fitness landscape. The result is shown in Fig. 15.*

```
plotModel(resK$modelFit)
```

□

The landscape generated with the random forest based SPOT run shown in Fig. 14 and the landscape from the Kriging-based SPOT run differ. We can check if longer runs result increase the similarities.

7.2.3 Continued Runs

The optimization procedure from Example 1, which is described in Sec. 5, uses 50 runs of the SANN algorithm. Now this number is increased to 100 and the procedure is continued. This means, an additional 50 evaluations are necessary since the results from the first runs are reused.

Example 12 (Example 1 continued with 50 additional runs). *The fitness landscape of the random forest meta model with 100 function evaluations is shown at the top in Fig. 16.*

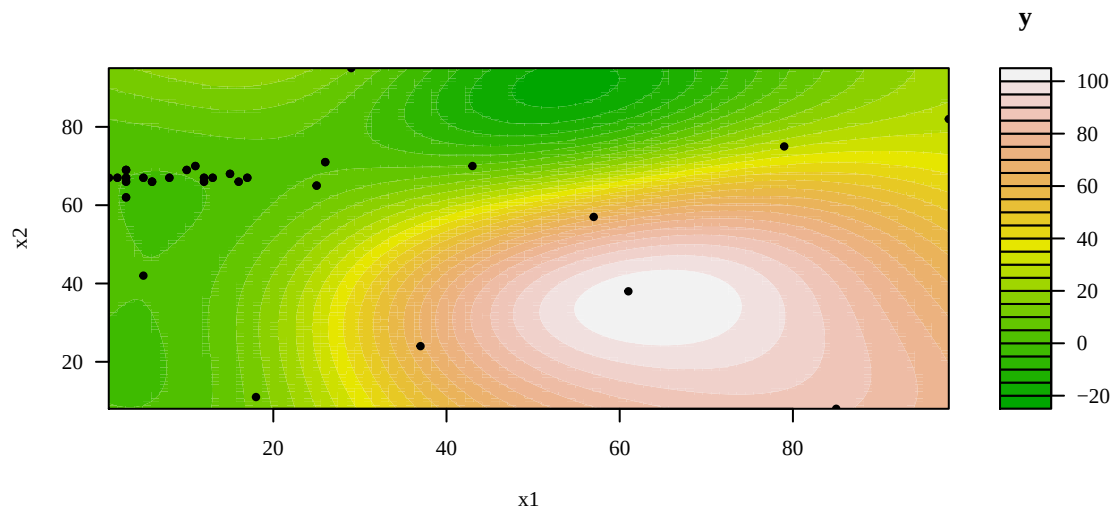


Fig. 14: Example 10: Results from the SPOT run with random forest, visualized with a Kriging model using `plotData()`

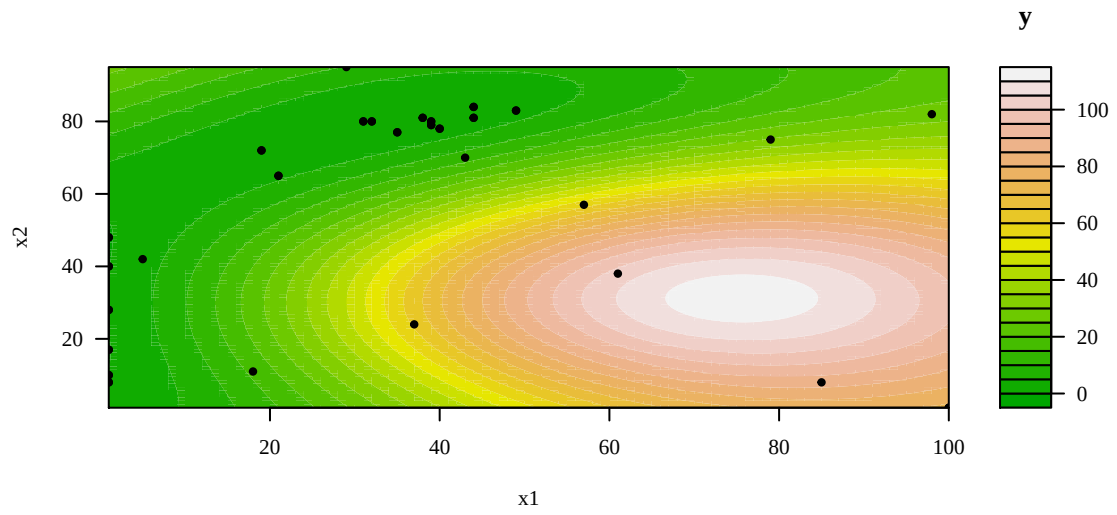


Fig. 15: Example 11: This shows the result of a SPOT run with Kriging, the visualization is based on the final Kriging model.

```
spotConfig$funEvals <- 100
spotConfig$model <- buildRandomForest
res100Rf <- spotLoop(resRf$x, resRf$y, fun = sann2spot, lower = lower, upper = upper,
    control = spotConfig)
```

□

Example 13 (Example 11 continued with 50 additional runs). *In a similar manner, new results can be added to the Kriging based optimization runs from Example 11.*

```
spotConfig$model = buildKriging
spotConfig$optimizer = optimLBFGSB
spotConfig$modelControl = list(algTheta = optimLBFGSB)
res100K <- spotLoop(resK$x, resK$y, fun = sann2spot, lower = lower, upper = upper,
    control = spotConfig)
```

□

A comparison of the models resulting from the two long runs (Example 12 and 13) is shown in Fig. 16. A visual inspection indicates that the landscapes are qualitatively similar, e.g., poor algorithm settings can be found in the lower right corner.

```
plotModel(res100Rf$modelFit)
plotModel(res100K$modelFit)
```

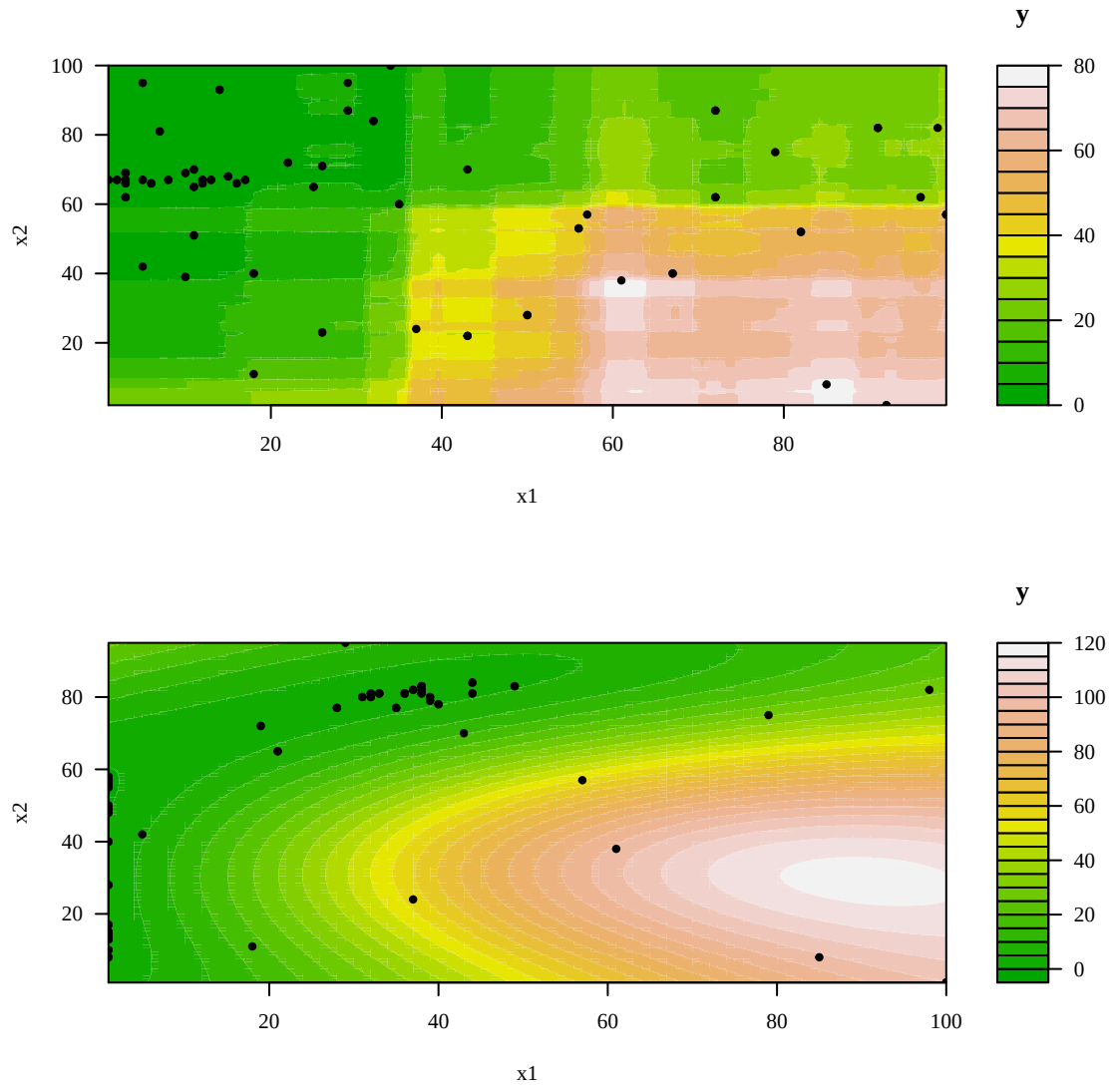


Fig. 16: Comparison of the long runs with 100 function evaluations and two different surrogate models. Top: Example 12. Long run using Random forest model. Bottom: Example 13. Long run with 100 function evaluations using a Kriging model.

8 Response Surface Methodology (RSM)

Using the `rsm` package, which is maintained by Lenth (2012), the `buildRSM()` function builds a linear response surface model. The arguments of `buildRSM(x, y, control = list())` are as follows:

- x: design matrix (sample locations), rows for each sample, columns for each variable.
- y: vector of observations at x
- control: (list), with the options for the model building procedure:
  - mainEffectsOnly: Logical, defaults to FALSE. Set to TRUE if a model with main effects only is desired (no interactions, second order effects).
  - canonical Logical, defaults to FALSE. If this is TRUE, use the canonical path to descent from saddle points. Else, simply use steepest descent

Example 14 (Path of the steepest descent). *First, we create some design points and compute observations at design points. Then, using `buildRSM()`, the response surface model is build. The function `descentSpotRSM()` returns the path of the steepest descent.*

```
x <- designUniformRandom(lower = rep(-5, 2), upper = rep(15, 2), control = list(size = 20))
y <- funSphere(x)
## Create model with default settings
fit <- buildRSM(x, y)
## Predict new point
predict(fit, cbind(1, 2))
```

```
## $y
##      [,1]
## [1,]    5
```

```
## True value at location
sphere(c(1, 2))
```

```
## [1] 5
```

```
descentSpotRSM(fit)
```

```
## Path of steepest descent from ridge analysis:
## $x
##           V1       V2
## 1   4.594420  5.21888
## 2   3.955628  4.60130
## 3   3.335624  3.96680
## 4   2.725014  3.32384
## 5   2.133192  2.66396
## 6   1.560158  1.99562
## 7   0.996518  1.31036
## 8   0.461060  0.61664
## 9  -0.065004 -0.09400
## 10 -0.572280 -0.80464
##
## $y
##               [,1]
##  [1,] 48.34540359
##  [2,] 36.81895456
##  [3,] 26.86188971
##  [4,] 18.47361365
##  [5,] 11.64719099
##  [6,]  6.41659217
##  [7,]  2.71009145
##  [8,]  0.59282121
##  [9,]  0.01306152
## [10,]  0.97494993
```

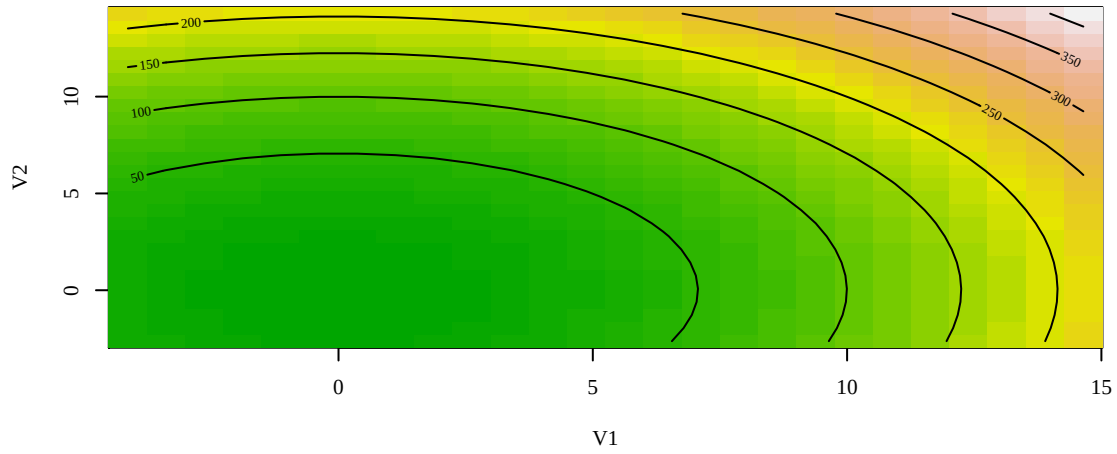


Fig. 17: Example 14. Illustrating the response surface, which fits the sphere function.

This situation is illustrated in Fig. 17.

```
plot(fit)
```

□

8.1 RSM and SPOT

Example 15 (RSM with SPOT). *We can use RSM for an interactive tuning approach. The starting point in this example are the 100 design points that were generated during the Kriging-based* SPOT *run in Example 13. These 100 data points are used to build a response surface with* `buildRSM()`.

```
rsm100K <- buildRSM(x = res100K$x, y = res100K$y)
summary(rsm100K$rsmfit)

##
## Call:
## rsm(formula = y ~ FO(x1, x2) + TWI(x1, x2) + PQ(x1, x2), data = codedData)
##
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  50.5759     5.4677  9.2499 7.234e-15 ***
## x1           37.4445     3.9033  9.5931 1.347e-15 ***
## x2          -36.7993     4.0066 -9.1847 9.957e-15 ***
## x1:x2       -18.5259     4.1972 -4.4138 2.708e-05 ***
## x1^2        -11.9005     5.6435 -2.1087 0.0376291 *
## x2^2        -25.4375     6.5577 -3.8790 0.0001944 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Multiple R-squared:  0.7836,Adjusted R-squared:  0.7721
## F-statistic: 68.08 on 5 and 94 DF,  p-value: < 2.2e-16
##
## Analysis of Variance Table
##
## Response: y
```

```
##             Df Sum Sq Mean Sq  F value    Pr(>F)
## FO(x1, x2)   2  42038 21019.0 147.7727 < 2.2e-16
## TWI(x1, x2)  1   4110  4109.9  28.8940 5.546e-07
## PQ(x1, x2)   2   2269  1134.7   7.9771  0.000631
## Residuals   94  13370   142.2
## Lack of fit 39   4944   126.8   0.8275  0.730891
## Pure error  55   8426   153.2
##
## Stationary point of response surface:
##        x1        x2
##  2.981237 -1.808933
##
## Stationary point in original units:
##        V1        V2
## 198.07121 -37.01983
##
## Eigenanalysis:
## eigen() decomposition
## $values
## [1]  -7.196669 -30.141326
##
## $vectors
##          [,1]      [,2]
## x1 -0.8916227 0.4527792
## x2  0.4527792 0.8916227
```

Following the path of the steepest descent on the RSM meta model, we obtain a new design point, which can be evaluated.

```
(xSteep <- descentSpotRSM(rsm100K))

## Path of steepest descent from ridge analysis:
## $x
##         V1     V2
## 1  47.0845 51.384
## 2  43.8670 54.956
## 3  40.9465 58.810
## 4  38.3725 62.852
## 5  36.2935 67.223
## 6  34.7095 71.876
## 7  33.7195 76.764
## 8  33.2740 81.840
## 9  33.3235 87.057
## 10 33.8680 92.274
##
## $y
##              [,1]
##  [1,]  45.246195
##  [2,]  39.708586
##  [3,]  33.918724
##  [4,]  27.953296
##  [5,]  21.717621
##  [6,]  15.163739
##  [7,]   8.309532
##  [8,]   1.063617
##  [9,]  -6.654501
## [10,] -14.722493
```

We have chosen the eighth point, i.e.,

```
xNew <- xSteep$x[8, ]
```

Then we determine its function value.

```
(yNew <- sann2spot(xNew))

##             [,1]
## [1,] 0.09871212
```

Next, we can refine the rsm meta model by including this point to the set of design points.

```
x101 <- rbind(res100K$x, xNew)
y101 <- rbind(res100K$y, yNew)
rsm101K <- buildRSM(x = x101, y = y101)
summary(rsm101K$rsmfit)

##
## Call:
## rsm(formula = y ~ FO(x1, x2) + TWI(x1, x2) + PQ(x1, x2), data = codedData)
##
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  50.5892     5.4365  9.3054 5.045e-15 ***
## x1           37.4612     3.8772  9.6618 8.727e-16 ***
## x2          -36.8123     3.9823 -9.2440 6.825e-15 ***
## x1:x2       -18.5126     4.1719 -4.4374 2.451e-05 ***
## x1^2        -11.8961     5.6137 -2.1191 0.0366875 *
## x2^2        -25.4750     6.5066 -3.9153 0.0001699 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Multiple R-squared:  0.7839,Adjusted R-squared:  0.7725
## F-statistic: 68.91 on 5 and 95 DF,  p-value: < 2.2e-16
##
## Analysis of Variance Table
##
## Response: y
##             Df Sum Sq Mean Sq  F value    Pr(>F)
## FO(x1, x2)   2  42109 21054.5 149.5869 < 2.2e-16
## TWI(x1, x2)  1   4103  4102.8  29.1492 4.927e-07
## PQ(x1, x2)   2   2284  1141.8   8.1120  0.000559
## Residuals   95  13371   140.8
## Lack of fit 40   4945   123.6   0.8069  0.759838
## Pure error  55   8426   153.2
##
## Stationary point of response surface:
##        x1        x2
##  2.978901 -1.804897
##
## Stationary point in original units:
##        V1        V2
## 197.95560 -36.83017
##
## Eigenanalysis:
## eigen() decomposition
## $values
## [1]  -7.206176 -30.164858
##
## $vectors
##          [,1]      [,2]
```

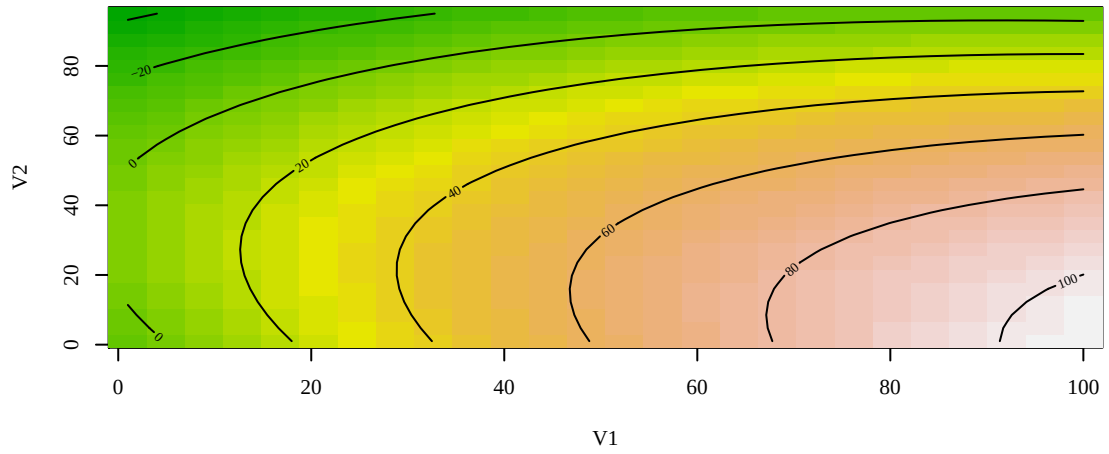


Fig. 18: Example 15. `buildRSM()` was used to build a response surface. Data from the Kriging-based SPOT run with 100 function evaluations and one additional point, which was calculated using the steepest descent function `descentSpotRSM()`, were used to generate this model.

```
## x1 -0.8920338 0.4519686
## x2  0.4519686 0.8920338
```

```
plot(rsm101K)
```

Now, we have two options for continuing the optimization. Either, we continue following the path of the steepest descent, i.e., we use SPOT*'s* `descentSpotRSM()` *function:*

```
descentSpotRSM(rsm101K)

## Path of steepest descent from ridge analysis:
## $x
##         V1     V2
## 1  47.0845 51.384
## 2  43.8670 54.956
## 3  40.9465 58.810
## 4  38.3725 62.852
## 5  36.2935 67.223
## 6  34.7095 71.876
## 7  33.7195 76.764
## 8  33.2740 81.840
## 9  33.3730 87.057
## 10 33.8680 92.274
##
## $y
##             [,1]
##  [1,]  45.257160
##  [2,]  39.716711
##  [3,]  33.923393
##  [4,]  27.953878
##  [5,]  21.713329
##  [6,]  15.153711
```

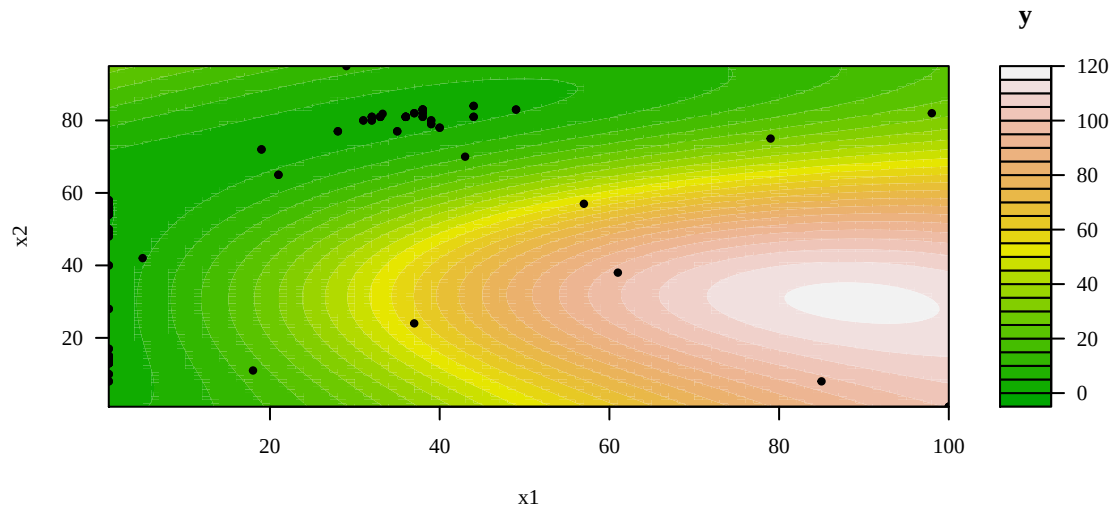


Fig. 19: Example 15. `buildRSM()` was used to build a response surface. Data from the Kriging-based SPOT run with 100 function evaluations, one additional point, which was calculated using the steepest descent function `descentSpotRSM()`, and one additional SPOT run with nine additional design points, were used to generate this model. Altogether, 110 design points were used to generate this model.

```
##  [7,]   8.292901
##  [8,]   1.039489
##  [9,]  -6.656699
## [10,] -14.764062
```

Or, we can continue with SPOT. *Following this second option, we have built an updated Kriging model using nine additional function evaluations:*

```
spotConfig$model = buildKriging
spotConfig$optimizer = optimLBFGSB
spotConfig$modelControl = list(algTheta = optimLBFGSB)
spotConfig$funEvals <- 110
res110K <- spotLoop(x = x101, y = y101, fun = sann2spot, lower = lower, upper = upper,
    control = spotConfig)
```

Finally, we can plot the updated model.

```
plotModel(res110K$modelFit)
```

□

9 Statistical Analysis

This section describes basic approaches for the analysis of the results from the previous example. We will continue using the data from Example 12.

9.1 Tree-based Analysis

SPOT's `buildRandomForest()` function is a wrapper function for the `randomForest` function from the `randomForest` package. Since the `randomForest` package has no default plot function, we switch to the `party` package. This package provides the `ctree()` function, which can be applied as follows:

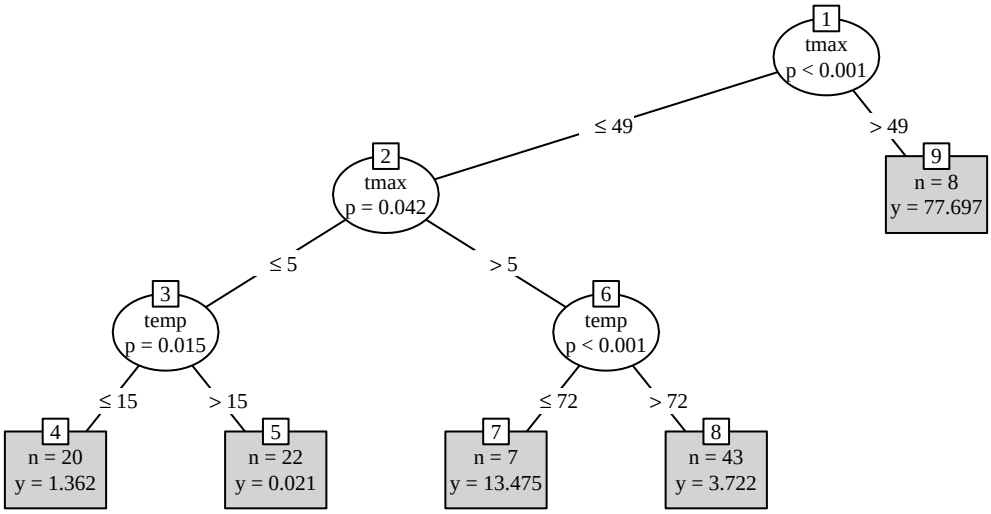


Fig. 20: Tree based analysis with Kriging data. tmax and temp

```
tmaxtempz.df <- data.frame(res100K$x[, 1], res100K$x[, 2], res100K$y)
names(tmaxtempz.df) <- c("tmax", "temp", "y")
tmaxtempz.tree <- ctree(y ~ ., data = tmaxtempz.df)
plot(tmaxtempz.tree, type = "simple")
```

9.2 Regression Analysis

9.2.1 Building the Linear Model

Data from the SPOT run can be used for building linear models. First, we extract the data from the result file. Then we can use the standard `lm()` function for building the linear model.

```
xyz100K.df <- data.frame(res100K$x[, 1], res100K$x[, 2], res100K$y)
names(xyz100K.df) <- c("x", "y", "z")
lm100K <- lm(z ~ x * y, data = xyz100K.df)
summary(lm100K)

##
## Call:
## lm(formula = z ~ x * y, data = xyz100K.df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -62.017  -4.323  -2.869   2.339  58.572
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.017644   3.328637   1.207   0.2304
## x            1.100721   0.076897  14.314  < 2e-16 ***
## y           -0.142629   0.079602  -1.792   0.0763 .
## x:y         -0.009159   0.001824  -5.023 2.35e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.76 on 96 degrees of freedom
```

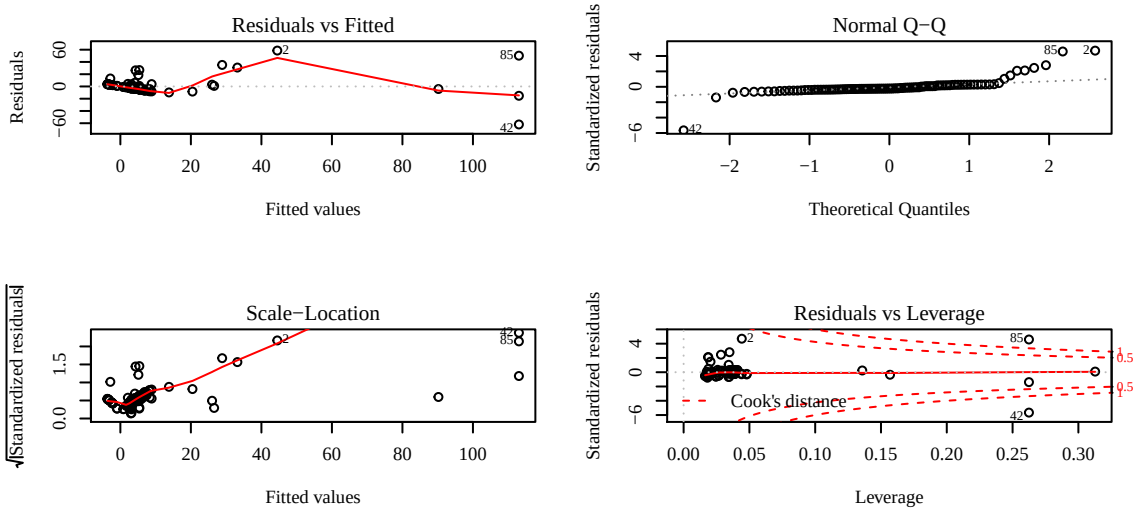


Fig. 21: Diagnostic plots of the linear regression model.

```
## Multiple R-squared:  0.7469,Adjusted R-squared:  0.739
## F-statistic: 94.42 on 3 and 96 DF,  p-value: < 2.2e-16
```

Diagnostic plots can be generated as follows:

```
par(mfrow = c(2, 2))
plot(lm100K)
par(mfrow = c(1, 1))
```

9.2.2 Estimating Variable Effects

R's `termplot()` function can be used to plot regression terms against their predictors, optionally with standard errors and partial residuals added.

```
par(mfrow = c(1, 2))
termplot(lm100K, partial = TRUE, smooth = panel.smooth, ask = FALSE)
par(mfrom = c(1, 1))
```

The `car` package provides the function `avPlots()`, which can be used for visualization as follows.

```
par(mfrow = c(1, 3))
avPlots(lm100K, ask = F)
par(mfrow = c(1, 1))
```

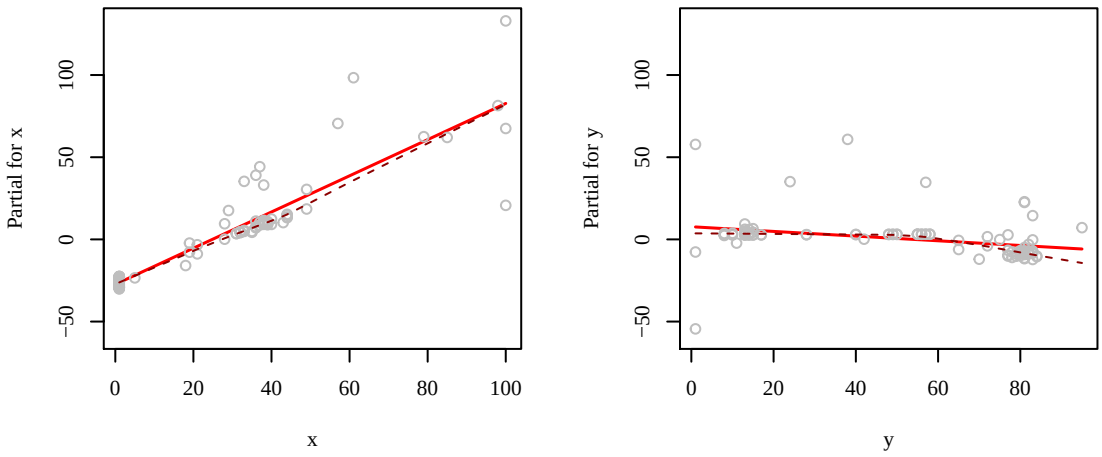


Fig. 22: Example 12: Termplots.

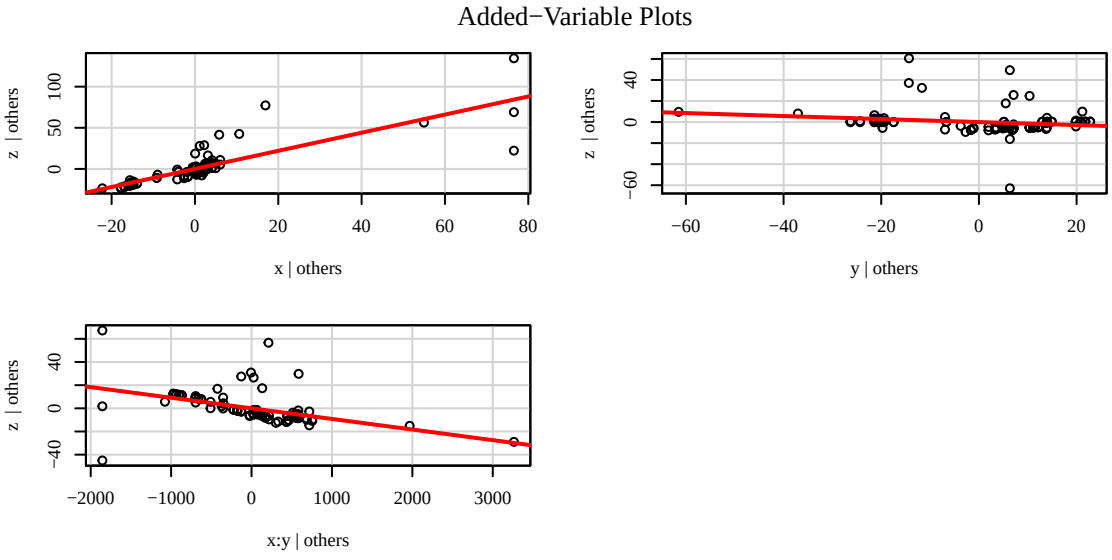


Fig. 23: Example 12: Added variable plots.

10 Deterministic Problems

Previous sections discussed the tuning of non-deterministic, i.e., noisy, algorithms, which is the usual case when tuning evolutionary algorithms. This section presents an application of SPOT in a simple setting: We will describe how SPOT can be used for tuning deterministic problems directly. To present a very simple example, SPOT will be used for minimizing the sphere function. Level (L2) from Fig. 1 is omitted, and the tuning algorithm for level (L3) is applied directly to the real-world system on level (L1). So instead of tuning SANN, which in turn optimizes the sphere function, SPOT tries to find the minimum of the sphere function directly. This example illustrates necessary modifications of the SPOT configuration in deterministic settings. Since no randomness occurs, repeats or other mechanism to cope with noise are not necessary anymore.

The deterministic sphere function (1) is used as an example. The interval $[-5; 5] \times [-5; 5]$ was chosen as the region of interest, i.e., we are considering a two-dimensional optimization problem.

```
res <- spot(, funSphere, c(-5, -5), c(5, 5), control = list(optimizer = optimLBFGSB))
```

We can extract the best solution with the following command.

```
res$xbest

##              [,1]         [,2]
## [1,] 0.002330654 0.0006385206

res$ybest

##               [,1]
## [1,] 5.839654e-06
```

11 Model Ensembles: Stacking

SPOT provides several models that can be used as surrogates. Sometimes it is not obvious, which surrogate should be chosen. Ensemble-based models provide a well-established solution to this model selection problem (Bartz-Beielstein & Zaefferer, 2017). Therefore, SPOT provides a stacking approach, that combines several models in a sophisticated manner. The stacking procedure is described in detail in Bartz-Beielstein (2016).

Example 16 (Stacking). *We will use the data from Example 6 to illustrate the stacking approach.*

```
fit.stack <- buildEnsembleStack(x.test, y.test)
plotModel(fit.stack)
```

We can compare predicted and true values as follows.

```
xNew <- cbind(1, 1, 1)
predict(fit.stack, xNew)

## $y
##        1
## 2.984905

funSphere(xNew)

##      [,1]
## [1,]    3
```

□

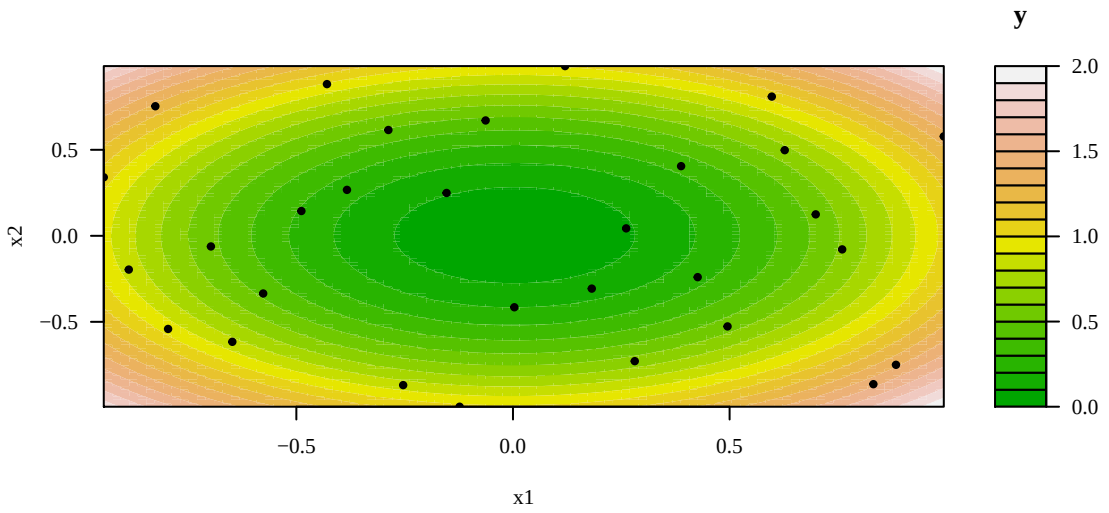


Fig. 24: Example 16: Fitness landscape based on stacking.

12 Summary

This report describes experimental methods for tuning algorithms. Using a simple simulated annealing algorithm, it was demonstrated how optimization algorithms can be tuned using the SPOT. Several tools from the SPOT for automated and interactive tuning were illustrated and the underling concepts of the SPOT approach were explained. Central in the SPOT approach are techniques such as exploratory fitness landscape analysis and response surface methodology. Furthermore, we demonstrated how SPOT can be used as optimizer and how a sophisticated ensemble approach is able to combine several meta models via stacking.

References

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

Bartz-Beielstein, T. (2016). *Stacked Generalization of Surrogate Models - A Practical Approach*. Technical Report 5/2016, TH Köln, Köln. `https://cos.bibl.th-koeln.de/frontdoor/index/index/docId/375`.

Bartz-Beielstein, T. & Zaefferer, M. (2017). Model-based methods for continuous and discrete global optimization. *Applied Soft Computing*, 55, 154 – 167.

Box, G. E. P. & Draper, N. R. (1987). *Empirical Model Building and Response Surfaces*. New York NY: Wiley.

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

Eiben, A. E. & Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Berlin, Heidelberg, New York: Springer.

Forrester, A., Sóbester, A., & Keane, A. (2008). *Engineering Design via Surrogate Modelling*. Wiley.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598), 671–680.

Lenth, R. V. (2012). *Response-Surface Methods in R Using rsm (Updated to version 2.00)*. Technical report.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Kontakt/Impressum

Diese Veröffentlichungen erscheinen im Rahmen der Schriftenreihe "CIplus". Alle Veröffentlichungen dieser Reihe können unter
https://cos.bibl.th-koeln.de/home
abgerufen werden.

Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.
Datum der Veröffentlichung: 18.12.2017

Herausgeber / Editorship

Prof. Dr. Thomas Bartz-Beielstein,
Prof. Dr. Wolfgang Konen,
Prof. Dr. Boris Naujoks,
Prof. Dr. Horst Stenzel
Institute of Computer Science,
Faculty of Computer Science and Engineering Science,
TH Köln,
Steinmüllerallee 1,
51643 Gummersbach
url: www.ciplus-research.de

Schriftleitung und Ansprechpartner/ Contact editor's office

Prof. Dr. Thomas Bartz-Beielstein,
Institute of Computer Science,
Faculty of Computer Science and Engineering Science,
TH Köln,
Steinmüllerallee 1, 51643 Gummersbach
phone: +49 2261 8196 6391
url: http://www.spotseven.de
eMail: thomas.bartz-beielstein@th-koeln.de

ISSN (online) 2194-2870

Technology
Arts Sciences
TH Köln

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722734.

Technology
Arts Sciences
TH Köln