Chapter 4
Hyperparameter Tuning Approaches

Thomas Bartz-Beielstein and Martin Zaefferer

Abstract This chapter provides a broad overview over the different hyperparameter tunings. It details the process of HPT, and discusses popular HPT approaches and difficulties. It focuses on surrogate optimization, because this is the most powerful approach. It introduces Sequential Parameter Optimization Toolbox (SPOT) as one typical surrogate method. SPOT is well established and maintained, open source, available on Comprehensive R Archive Network (CRAN), and catches mistakes. Because SPOT is open source and well documented, the human remains in the loop of decision-making. The introduction of SPOT is accompanied by detailed descriptions of the implementation and program code. This chapter particularly provides a deep insight in Kriging (aka Gaussian Process (GP) aka Bayesian Optimization (BO)) as a workhorse of this methodology. Thus it is very hands-on and practical.

4.1 Hyperparameter Tuning: Approaches and Goals

The following HPT approaches are popular:

- manual search (or trial-and-error (Meignan et al. 2015)),
- simple Random Search (RS), i.e., randomly and repeatedly choosing hyperparameters to evaluate,
- grid search (Tatsis and Parsopoulos 2016),
- directed, model free algorithms, i.e., algorithms that do not explicitly make use of a model, e.g., Evolution Strategys (ESs) (Hansen 2006; Bartz-Beielstein et al. 2014) or pattern search (Lewis et al. 2000),

T. Bartz-Beielstein (✉)
Institute for Data Science, Engineering and Analytics, TH Köln, Gummersbach, Germany
e-mail: thomas.bartz-beielstein@th-koeln.de

M. Zaefferer
Bartz & Bartz GmbH and with Institute for Data Science, Engineering, and Analytics, TH Köln, Gummersbach, Germany

Duale Hochschule Baden-Württemberg Ravensburg, Ravensburg, Germany
e-mail: zaefferer@dhbw-ravensburg.de

E. Bartz et al. (eds.), *Hyperparameter Tuning for Machine and Deep Learning with R*,
https://doi.org/10.1007/978-981-19-5170-1_4

- hyperband, i.e., a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive halving to stop poorly performing configurations (Li et al. 2016),
- Surrogate Model Based Optimization (SMBO) such as SPOT, (Bartz-Beielstein et al. 2005, 2021).[1]

Manual search and grid search are probably the most popular algorithms for HPT. Similar to suggestions made by Bartz-Beielstein et al. (2020a), we propose the following recommendations for performing HPT studies:

(R-1) Goals: clearly state the reasons for performing HPT. Improving an existing solution, finding a solution for a new, unknown problem, or benchmarking two methods are only three examples with different goals. Each of these goals requires a different experimental design.
(R-2) Problems: select suitable problems. Decide, how many different problems or problem instances are necessary. In some situations, surrogates (e.g., Computational Fluid Dynamics (CFD) simulations) can accelerate the tuning (Bartz-Beielstein et al. 2018).
(R-3) Algorithms: select a portfolio of ML and DL algorithms to be included in the HPT experimental study. Consider base-line methods such as RS and methods with their default hyperparameter settings.
(R-4) Performance: specify the performance measure(s). See the discussion in Sect. 2.2.
(R-5) Analysis: describe how the results can be evaluated. Decide, whether parametric or non-parametric methods are applicable. See the discussion in Chap. 5.
(R-6) Design: set up the experimental design of the study, e.g., how many runs shall be performed. Tools from Design of Experiments (DOE) and Design and Analysis of Computer Experiments (DACE) are highly recommended. See the discussion in Sect. 5.6.5.
(R-7) Presentation: select an adequate presentation of the results. Consider the audience: a presentation for the management might differ from a publication in a journal.
(R-8) Reproducibility: consider how to guarantee scientifically sound results and how to guarantee a lasting impact, e.g., in terms of comparability. López-Ibáñez et al. (2021a) present important ideas.

In addition to these recommendations, there are some specific issues that are caused by the ML and DL setup.

We consider a HPT approach based on SPOT that focuses on the following topics:

Limited Resources. We focus on situations, where limited computational resources are available. This may be simply due to the availability and

[1] The acronym SMBO originated in the engineering domain (Booker et al. 1999; Mack et al. 2007). It is also popular in the ML community, where it stands for *sequential model-based optimization*. We will use the terms *sequential model-based optimization* and *surrrogate model-based optimization* synonymously.

	cost of hardware, or because confidential data has to be processed strictly locally.
Understanding.	In contrast to standard HPO approaches, SPOT provides statistical tools for *understanding* hyperparameter importance and interactions between several hyperparameters.
Explainability.	Understanding is a key tool for enabling transparency and explainability, e.g., quantifying the contribution of ML and DL components (layers, activation functions, etc.).
Replicablity.	The software code used in this study is available in the open source R software environment for statistical computing and graphics (R) package SPOT via the CRAN. Replicability is discussed in Sect. 2.7.2. SPOT is a well-established open-source software, maintained for more than 15 years (Bartz-Beielstein et al. 2005).

Furthermore, Falkner et al. (2018) claim that practical HPO solutions should fulfill the following requirements:

- strong anytime and final performance,
- effective use of parallel resources,
- scalability, as well as robustness and flexibility.

For sure, we are not seeking the overall best hyperparameter configuration that results in a method which outperforms any other method in every problem domain (Wolpert and Macready 1997). Results are specific for one problem instance—their generalizability to other problem instances or even other problem domains is not self-evident and has to be proven (Haftka 2016).

4.2 Special Case: Monotonous Hyperparameters

A special case is hyperparameters with monotonous effect on the quality and run time (and/or memory requirements) of the tuned model. In our survey (see Table 4.1), two examples are included: `num.trees` (RF) and `thresh` (EN). Due to the monotonicity properties, treating these parameters differently is a likely consideration. In the following, we focus the discussion on `num.trees` as an example, since this parameter is frequently discussed in literature and online communities (Probst et al. 2018).

It is known from the literature that larger values of `num.trees` generally lead to better models. As the size increases, a saturation sets in, leading to progressively lower quality gains. It should be noted that this is not necessarily true for every quality measure. Probst et al. (2018), for example, show that this relation holds for log-loss and Brier score, but not for Area Under the receiver operating characteristic Curve (AUC).

Table 4.1 Global hyperparameter overview. The column "Quality" shows all parameter, where a monotonous relationship between parameter values and model quality is to be expected. (↑↑: quality increases if parameter value increases, ↑↓: quality decreases if parameter value increases). Correspondingly, the column "run time" shows the same information for the relationship of parameter values and run time

Model	Hyperparameter	Quality	Run time
KNN	k		↑↑
	p		
EN	alpha		
	lambda		
	thresh	↑↓	↑↓
DT	minsplit		↑↓
	minbucket		↑↓
	cp		↑↓
	maxdepth		↑↑
RF	num.trees	↑↑	↑↑
	mtry		↑↑
	sample.fraction		↑↑
	replace		
	respect.unordered.factors		
xgBoost	eta		
	nrounds		↑↑
	lambda		
	alpha		
	subsample		↑↑
	colsample_bytree		↑↑
	gamma		↑↓
	max_depth		↑↑
	min_child_weight		↑↓
SVM	kernel		
	degree		
	gamma		
	coef0		
	cost		
	epsilon		

Because of this relationship, Probst et al. (2018) claim that `num.trees` should not be optimized. Instead, it is recommended setting the parameter to a "computationally feasible large number" (Probst et al. 2018). For certain applications, especially for relatively small or medium-sized data sets, we support this assessment. However, at least in perspective, the analysis in this book considers tuning hyperparameters for very large data sets (many observations and/or many features). For this use case,

we do not share this recommendation, because the required run time of the model plays an increasingly important role and is not explicitly considered in the recommendation. In this case, a "computationally feasible large number" is not trivial to determine.

In total, we consider five solutions for handling monotonous hyperparameters:

(M-1) Set manually: The parameter is set to the largest possible value that is still feasible with the available computing resources. This solution involves the following risks:

a. Single evaluations during tuning waste time unnecessarily.
b. Interactions with parameters (e.g., `mtry`) are not considered.
c. The value may be unnecessarily large (from a model quality point of view).
d. The determination of this value can be difficult, it requires detailed knowledge regarding: size of the data set, efficiency of the model implementation, available resources (memory/computer cores / time).

(M-2) Manual adjustment of the tuning: After a preliminary examination (as represented, e.g., by the initial design step of SPOT) a user intervention takes place. Based on the preliminary investigation, a value that seems reasonable is chosen by the user and is not changed in the further course of the tuning. This solution involves the following risks:

a. The preliminary investigation itself takes too much time.
b. The decision after the preliminary investigation requires intervention by the user (problematic for automation). While this is feasible for individual cases, it is not practical for numerous experiments with different data (as in the experiments of the study in Chap. 12). Moreover, this reduces the reproducibility of the results.
c. Depending on the scope and approach of the preliminary study, interactions with other parameters may not be adequately accounted for.

(M-3) No distinction: parameters like `num.trees` are optimized by the tuning procedure just like all other hyperparameters. This solution involves the following risks:

a. The upper bound for the parameter is set too low, so potentially good models are not explored by the tuning procedure. (Note: bounds set too tight for the search space are a general risk that can affect all other hyperparameters as well).
b. The upper bound is set too high, causing individual evaluations to use unnecessary amounts of time during tuning.
c. The best found value may become unnecessarily large (from a model quality point of view).

(M-4) Multi-objective: run time and model quality can be optimized simultaneously in the context of multi-objective optimization. This solution involves the following risks:

a. Again, manual evaluation is necessary (selection of a sector of the Pareto front) to avoid that from a practical point of view irrelevant (but possibly Pareto-optimal) solutions are investigated.
b. This manual evaluation also reduces reproducibility.

(M-5) Regularization via weighted sum: The number of trees (or similar parameters) can be incorporated into the objective function. In this case, the objective function becomes a weighted sum of model quality and number of trees (or run time), with a weighting factor θ.

a. The new parameter of the tuning procedure, θ, has to be determined.
b. Moreover, the optimization of a weighted sum cannot find certain Pareto-optimal solutions if the Pareto front is non-convex.

In the experimental investigation in Chap. 12, we use solution (M-3). That is, the corresponding parameters are tuned but do not undergo any special treatment during tuning. Due to the large number of experiments, user interventions would not be possible and would also complicate the reproducibility of the results. In principle, we recommend this solution for use in practice.

In individual cases, or if a good understanding of algorithms and data is available, solution (M-2) can also be used. For this, SPOT can be interrupted after the first evaluation step, in order to set the corresponding parameters to a certain value or to adjust the bounds if necessary (e.g., if `num.trees` was examined with too low an upper bound).

4.3 Model-Free Search

4.3.1 Manual Search

A frequently applied approach is that ML and DL methods are configured manually (Bergstra and Bengio 2012). Users apply their own experience and trial-and-error to find reasonable hyperparameter values.

In individual cases, this approach may indeed yield good results: when expert knowledge about data, methods, and parameters is available. At the same time, this approach has major weaknesses, e.g., it may require significant amount of work time by the users, bias may be introduced due to wrong assumptions, limited options for parallel computation, and extremely limited reproducibility. Hence, an automated approach is of interest.

4.3.2 Undirected Search

Undirected search algorithms determine new hyperparameter values independently of any results of their evaluation. Two important examples are Grid Search and RS.

Grid Search covers the search space with a regular grid. Each grid point is evaluated. RS selects new values at random (usually independently, uniform distributed) in the search space.

Grid Search is a frequently used approach, as it is easy to understand and implement (including parallelization). As discussed by Bergstra and Bengio (2012), RS shares the advantages of Grid Search. However, they show that RS may be preferable to Grid Search, especially in high-dimensional spaces or when the importance of individual parameters is fairly heterogeneous. They hence suggest to use RS instead Grid Search if such simple procedures are required. Probst et al. (2019a) also use a RS variant to determine the tunability of models and hyperparameters. For these reasons, we employ RS as a baseline for the comparison in our experimental investigation in Chap. 12.

Next to Grid Search and RS, there are other undirected search methods. Hyperband is an extension of RS, which controls the use of certain resources (e.g., iterations, training time) (Li et al. 2018). Another relevant set of methods is the Design of Experiments methods, such as Latin Hypercube Designs (Leary et al. 2003).

! Attention: Random Search Versus Grid Search

Interestingly, Bergstra and Bengio (2012) demonstrate empirically and show theoretically that randomly chosen trials are more efficient for HPT than trials on a grid. Because their results are of practical relevance, they are briefly summarized here: In grid search the set of trials is formed by using every possible combination of values, grid search suffers from the curse of dimensionality because the number of joint values grows exponentially with the number of hyperparameters.

> A Gaussian process analysis of the function from hyper-parameters to validation set performance reveals that for most data sets only a few of the hyper-parameters really matter, but that different hyper-parameters are important on different data sets. This phenomenon makes grid search a poor choice for configuring algorithms for new data sets (Bergstra and Bengio 2012).

Let Ψ denote the space of hyperparameter response functions. Bergstra and Bengio (2012) claim that RS is more efficient in ML than grid search because a hyperparameter response function $\psi \in \Psi$ usually has a low effective dimensionality (see Definition 2.25), i.e., ψ is more sensitive to changes in some dimensions than others (Caflisch et al. 1997).

The observation that only a few of the parameters matter can also be made in the engineering domain, where parameters such as pressure or temperature play a dominant role. In contrast to DL, this set of important parameters does not change fundamentally in different situations. We assume that the high variance in the set of important DL hyperparameters is caused by confounding.

Due to its simplicity, it turns out in many situations that RS is the best solution, especially in high-dimensional spaces. Hyperband should also be mentioned in this context, although it can result in a worse final performance than model-based approaches, because it only samples configurations randomly and does not learn from previously sampled configurations (Li et al. 2016). Bergstra and Bengio (2012) note that RS can probably be improved by automating what manual search does, i.e., using SMBO approaches such as SPOT.

HPT is a powerful technique that is an absolute requirement to get to state-of-the-art models on any real-world learning task, e.g., classification and regression. However, there are important issues to keep in mind when doing HPT: for example, validation-set overfitting can occur, because hyperparameters are usually optimized based on information derived from the validation data.

4.3.3 Directed Search

One obvious disadvantage of undirected search is that a large amount of the computational effort may be spent on evaluating solutions that cover the whole search space. Hence, only a comparatively small amount of the computational budget will be spent on potentially optimal or at least promising regions of the search space.

Directed search on the other hand may provide a more purposeful approach. Basically any gradient-free, global optimization algorithm could be employed. Prominent examples are Iterative Local Search (ILS) (Hutter et al. 2010b) and Iterative Racing (IRACE) (López-Ibáñez et al. 2016). Metaheuristics like Evolutionary Algorithms (EAs) or Swarm Optimization are also applicable (Yang and Shami 2020). In comparison to undirected search procedures, directed search has two frequent drawbacks: an increased complexity that makes implementation a larger issue, and being more complicated to parallelize.

We employ a *model-based* directed search procedure in this book, which is described in the following Sect. 4.4.

4.4 Model-Based Search

A disadvantage of model-free, directed search procedures is that they may require a relatively large number of evaluations (i.e., long run times) to approximate the values of optimal hyperparameters.

Tuning ML and DL algorithms can become problematic if complex methods are tuned on large data sets, because the run time for evaluating a single hyperparameter configuration may go up into the range of hours or even days. Model-based search is one approach to resolve this issue. These search procedures use information gathered during the search to learn the relationship between hyperparameter values and

performance measures (e.g., misclassification error). The model that encodes this learned relationship is called the surrogate model or *surrogate*.

Definition 4.1 (*Surrogate Optimization*) The surrogate optimization uses two phases.

Construct Surrogate Generate (random) solutions. Evaluate the (expensive) objective function at these points. Construct a surrogate, $\mathcal{S}$, of the objective function, e.g., by building a GP aka Kriging model (surrogate).

Search for Minimum Search for a minimum of the objective function on the (cheap) surrogate. Choose the best point as a candidate. Evaluate the objective function at the best candidate point. This point is called an infill point. Update the surrogate using this value and search again.

The advantage of this surrogate optimization is that a considerable part of the evaluation burden (i.e., the computational effort) can be shifted from *real* evaluations to evaluations of the surrogate, which should be faster to evaluate.

In HPT, mixed optimization problems are common, i.e., the variables are continuous or discrete (Cuesta Ramirez et al. 2022). Bartz-Beielstein and Zaefferer (2017) provide an overview of metamodels that have or can be used in optimization. They show how it was made possible by the realization that GP kernels (covariance functions) in mixed variables can be created by composing continuous and discrete kernels. In this case, the infill criterion (acquisition function) is defined over the same space as the objective function. Therefore maximizing the acquisition function is also a mixed variables problem.

One variant of model-based search is SPOT (Bartz-Beielstein 2005), which be will described in Sect. 4.5.

4.5 Sequential Parameter Optimization Toolbox

SMBO methods are common approaches in simulation and optimization. SPOT 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 Classification and Regression Trees (CART) and random forest; BO (Gaussian process models, aka Kriging), and combinations of different meta-modeling approaches.

Basic elements of the Kriging-based surrogate optimization such as interpolation, expected improvement, and regression are presented in the Appendix, see Sect. 4.6. The Sequential Parameter Optimization (SPO) toolbox implements a modified version of this method and will be described in this section.

SPOT implements key techniques such as exploratory fitness landscape analysis and sensitivity analysis. SPOT can be used for understanding the performance of algorithms and gaining insight into algorithm's behavior. Furthermore, SPOT can be used as an optimizer and for automatic and interactive tuning.

Details of SPOT and its application in practice are given by Bartz-Beielstein et al. (2021). SPOT was originally developed for the tuning of optimization algorithms. The requirements and challenges of algorithm tuning in optimization broadly reflect those of tuning machine learning models. SPOT uses the following approach (outer loop).

Setup: In a first step, several candidate solutions (here: different combinations of hyperparameter values) are created. These are steps (S-1) and (S-2) in the function `spot`, see Fig. 4.1.

Evaluate: All new candidate solutions are evaluated (here: training the respective ML or DL model with the specified hyperparameter values and measuring the quality / performance). This is step (S-3).

Termination: Check whether a termination criterion has been reached (e.g., number of iterations, evaluations, run time, or a satisfying solution has been found). These are steps (S-4) to (S-9).

Select: Samples for building the surrogate $\mathcal{S}$are selected. This is step (S-10).

Training: The surrogate $\mathcal{S}$will be trained with all data derived from the evaluated candidate solutions, thus learning how hyperparameters affect model quality. This is step (S-11).

Surrogate search: The trained model is used to perform a search for new, promising candidate solutions. These are steps (S-12) and (S-16).

Budget: Optimal Computing Budget Allocation (OCBA) is used to determine the number of repeated evaluations. This is step (S-17).

Evaluation The new solutions are evaluated on the objective function, e.g., the loss is determined. These are steps (S-18) to (S-22).

Exploit: An optimizer is used to perform a local search on $\mathcal{S}$to refine the best solution found. These are steps (S-23) and (S-24). Whereas optimization on the surrogate in the main loop is a weighted combination of exploration and exploita-

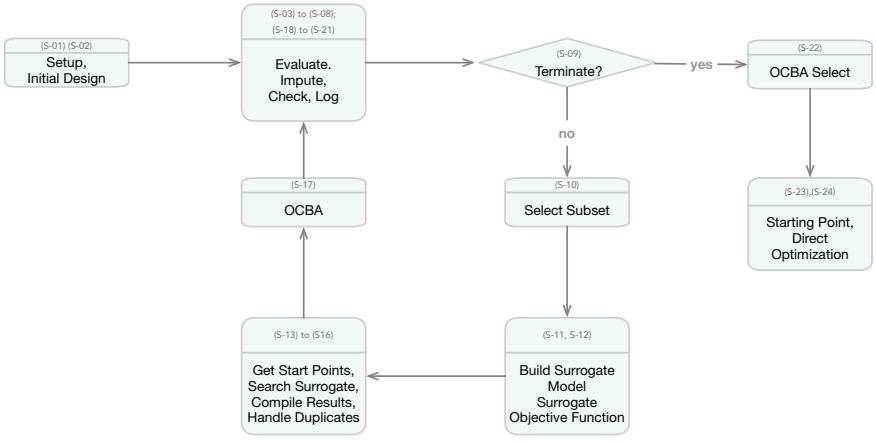

Fig. 4.1 Visual representation of model-based search with SPOT

tion, using Expected Improvenment (EI) as a default weighting mechanism, this final optimization step is purely exploitative.

Note, that it can be useful to allow for user interaction with the tuner after the evaluation step. Thus, the user may affect changes of the search space (stretch or shrink bounds on parameters, eliminate parameters). However, we will consider an automatic search in our experiments.

We use the R implementation of SPOT, as provided by the R package `SPOT` (Bartz-Beielstein et al. 2021, 2021c). The SPOT workflow will be described in the following sections.

In the remainder of this book, SPOT will refer to the general method, whereas `spot` denotes the function from the R package `SPOT`.

Steps, subroutines and data of the `spot` process are shown in Fig. 4.2.

4.5.1 spot as an Optimizer

`spot` uses the same syntax as `optim`, R's general-purpose optimization based on Nelder-Mead, quasi-Newton, and conjugate-gradient algorithms (R Core Team 2022). `spot` can be called as shown in the following example.

Example: spot

`SPOT` comes with many pre-defined functions from optimization, e.g., Sphere, Rosenbrock, or Branin. These implementations use the prefix "`fun`", e.g., `funSphere` is the name of the sphere function. The package `SPOTMisc` provides `funBBOBCall`, an interface to the real-parameter Black-Box Optimization Benchmarking (BBOB) function suite (Mersmann et al. 2010a). Furthermore, users can also specify their own objective functions.

Searching for the optimum of the (two-dimensional) sphere function `funSphere`, i.e., $f(x) = \sum_{i=1}^{2} x_i^2$, on the interval between $(-1, -1)$ and $(1, 1)$ can be done as follows:

```
library("SPOT")
spot(x = NULL, fun = funSphere, lower = c(-1, -1), upper = c(1, 1))
```

Four arguments are passed to `spot`: no explicit starting point for the optimization is used, because the parameter `x` was set to `NULL`, the function `funSphere`, and the `lower` and `upper` bounds. The length of the lower bound argument defines the problem dimension n.

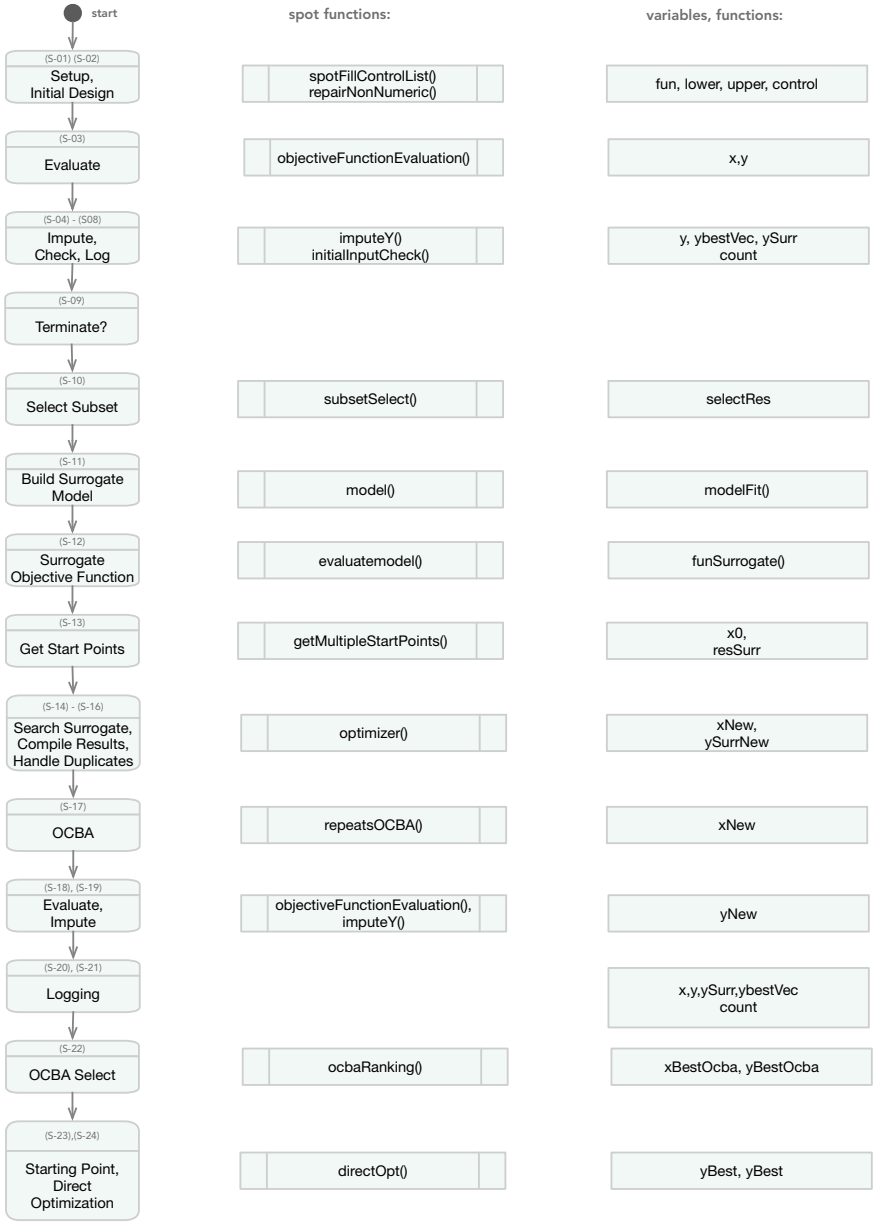

Fig. 4.2 Steps, functions and variables of the `spot` function

Table 4.2 SPOT parameters. This table shows the mandatory parameters. The list `control` can be used to pass additional parameters to `spot`. Additional arguments to the objective function `fun` can be passed via "...", similar to the varargs method in other programming languages

Parameter	Default value	Description
`x`	`NULL`	Starting point
`fun`		Objective function, e.g., `funSphere`, or as described in Sect. 8.44
`lower`		Lower bound, defines the problem dimension n
`upper`		Upper bound
`control`	List	See description in Table 4.3
`...`		Used to pass those additional arguments on to the objective function `fun`

> Mandatory Parameters

The arguments `x`, `fun`, `lower`, and `upper` are mandatory for `spot`, they are shown in Table 4.2.

Additional arguments can be passed to `spot`. They allow a very flexible handling, e.g., for passing extra arguments to the objective function `fun`. To improve the overview, parameters are organized as lists. The "main" list is called `control`, see Table 4.3. It collects `spot`'s parameters, some of them are organized as lists. They are shown in Table 4.4.

The `control` list is used for managing `SPOT`'s parametrization, e.g., for defining hyperparameter types and ranges.

4.5.2 spot's Initial Phase

The initial phase consists of five steps (S-1) to (S-5). The corresponding R code is shown in Sect. 4.7.

(S-1) *Setup.* After performing an initial check on the control list, the `control` list is completed.
The `control` list contains the parameters from Table 4.3.

(S-2) *Initial design.* The parameter `seedSPOT` is used to set the seed for `spot` before the initial design X is generated. The design type is specified via `control$design`. The recommended design function is `designLHD`, i.e., a Latin Hypercube Design (LHD), which is also the default configuration.

Table 4.3 SPOT: parameters of the control list

Parameter	Default value, type	Description
design	`designLHD`, function	The design function is used to generate the initial design (see `spot`) and to generate multiple start points (see `getMultiStartPoints`)
directOpt	`optimNLOPTR`, function	Optimizer used for direct optimization after SMBO is done
funEvals	20	Number of objective function (`fun`) evaluations
infillCriterion	`NULL`, function	A function defining an infill criterion to be used while optimizing a model
model	`buildKriging`, function	A function that builds a statistical model of the observed data
multiStart	`1` (no multi starts), integer	Number of restarts of the `optimizer` on the surrogate `model`
noise	`FALSE`, logical	
OCBA	paramFALSE, logical	Use OCBA
OCBABudget	`3`, integer	Budget for OCBA
optimizer	function	Optimizer on surrogate `model`
parNames	character, paste0 (“x”, 1:dimension)	Hyperparameter names
plots	paramFALSE, logical	Show progress plots
progress	paramFALSE, logical	Show numerical information about the progress
replicateResults	paramFALSE, logical	Evaluate configuration(s), do not perform SMBO
replicates	integer	Number of replicates
returnFullControlList	logical	Return the full control list
seedFun	seed function for objective function	
seedSPOT	seed used for spot	
subsetSelect	selectAll	Subset used for fitting the surrogate model
tolerance	numerical	Sqrt(.Machine$double.eps)
transformFun	vector	Variable transformation
types	rep (“numeric”, dimension)	Hyperparameter types
verbosity	integer	Verbosity
xNewActualSize	integer	Number of new design points proposed by the surrogate model
designControl	list	Parameters used by the `design` function
directOptControl	list	Parameters used by the `direct` function
modelControl	list	Parameters used by the surrogate `model`
optimizerControl	list	Parameters used by the `optimizer`
subsetControl	list	Parameters used by the `subsetSelect` function
time	list	Time related parameters
yImputation	list	List of functions to determine imputations, handleNAsMethod

Table 4.4 SPOT: parameters of the other lists

List	Parameter	Default value, type	Description
`designControl`	replicates	Rinit	
	size	initSizeFactor * length(cfg$lower)	
`directOptControl`	list	parameters used by the `direct` function. Direct optimization is performed, if `control$directOptControl$funEvals` > 0	
`modelControl`	target	krigingTarget	
	useLambda	krigingUseLambda	
	reinterpolate	krigingReinterpolate	re-interpolation is supposed to be used with stochastic experiments, which do need a non-interpolating model that avoids predicting nonzero error at sample locations. Can be useful when the model is deterministic, i.e., repeated evaluations of one parameter vector do not yield different values but does have a "noisy" structure (e.g., due to computational inaccuracies, systematical errors)
	infillCriterion	`NULL`	
`optimizerControl`	funEvals	multFun * length(cfg$lower))	
`time`	maxTime	`Inf`	time bugdet in minutes
`yImputation`	handleNAsMethod	handleNAsMethod	
	imputeCriteriaFuns	imputeCriteriaFuns	
	penaltyImputation	3	

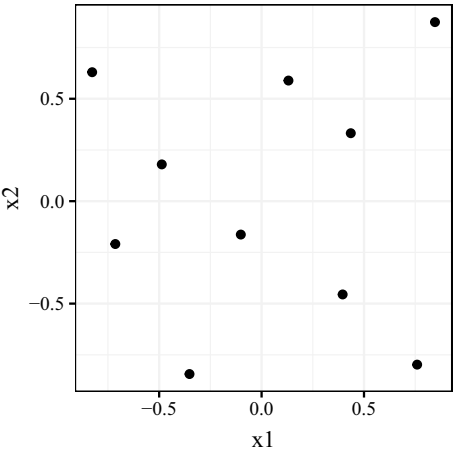

Fig. 4.3 Initial design. The first ten points created by designLHD

Ten initial design points are available now, because the default value of the parameter `designControl$size`, which specifies the initial design size, is set to 10 if the function `designLHD` is used (Fig. 4.3).

Program Code: Steps (S-1) and (S-2)

Steps (S-1) and (S-2) are implemented as follows:

```
## (S-1) Setup:
fun <- funNoise
lower <- c(-1, -1)
upper <- c(1, 1)
control <- list(
  OCBA = TRUE,
  OCBABudget = 3,
  replicates = 2,
  noise = TRUE,
  multiStart = 2,
  designControl = list(replicates = 2)
)
control <- spotFillControlList(control, lower, upper)
## (S-2) Initial design:
set.seed(control$seedSPOT)
x <- control$design(
  x = NULL,
  lower = lower,
  upper = upper,
  control = control$designControl
)
x <- repairNonNumeric(x, control$types)
```

Example: Modifying the initial design size

Arbitrary initial design sizes can be generated by modifying the `size` argument of the `designControl` list:

```
control$designControl$size <- 5
```

Here is the full code for starting `spot` with an initial design of size five:

```
spot(
  x = NULL,
  fun = funSphere,
  lower = c(-1, -1), upper = c(1, 1),
  control = list(designControl = list(size = 5))
)
```

Because the `lower` bound was set to `(-1, -1)`, a two-dimensional problem is defined, i.e., $f(x_1, x_2) = x_1^2 + x_2^2$. The result from this `spot` run is stored as a list in the variable `return`.

Variable types are assumed to be `numeric`, which is the default type if no other type is specified. Type information, which is available from `config$types`, is used to transform the variables. The function `spot` can handle the data types `numeric`, `integer`, and `factor`. The function `repairNonNumeric` maps non-numerical values to `integers`.

(S-3) *Evaluation of the Initial Design.* Using `objectiveFunction Evaluation`, the objective function `fun` is evaluated on the initial design matrix `x`.
In addition to `xnew`, a matrix of already known solutions, to determine whether Random Number Generator (RNG) seeds for new solutions need to be incremented, can be passed to the function `objectiveFunction Evaluation`.

! Transformation of Variables

If variable transformation functions are defined, the function `transformX` is applied to the parameters during the execution of the function `objectiveFunctionEvaluation`.

The function `objectiveFunctionEvaluation` returns the matrix `y`.

(S-4) *Imputation: Handling Missing Values.* The feasibility of the y-matrix is checked. Methods to handle `NA` and infinite y-values are applied, which are available via the function `imputeY`.
The spot loop starts after the initial phase. The function `spotLoop` is called.

Program Code: Steps (S-3) and (S-4)

Steps (S-3) and (S-4) are implemented as follows:

```
## (S-3) Eval initial design:
y <- objectiveFunctionEvaluation(
  x = NULL,
  xnew = x,
  fun = fun,
  control = control
)
## (S-4) Imputation:
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}
```

4.5.3 The Function `spotLoop`

(S-5) *Calling the `spotLoop` function.* After the initial phase is finished, the function `spotLoop` is called, which manages the main loop. It is implemented as a stand-alone function, because it can be called separately, e.g., to continue interrupted experiments. With this mechanism, `spot` provides a convenient way for continuing experiments on different computers or extending existing experiments, e.g., if the results are inconclusive or a pre-experimental study should be performed first.

Example: Continue existing experiments

The studies in Sects. 5.8.1, 5.8.2, and 5.8.3 start with a relatively small pre-experimental design. Results from the pre-experimental tests are combined with results from the full experiment.

(S-6) *Consistency Check and Initialization.* Because the `spotLoop` can be used to continue an interrupted `spot` run, it performs a consistency check before the main loop is started.

(S-7) *Imputation.* The function defined by the argument `control$yImputation$handleNAsMethod` is called to handle `NA` s, `Inf` s, etc. This is necessary here, because `spotLoop` can be used as an

entry point to continue an interrupted `spot` optimization run. How to continue existing `spot` runs is explained in the `spotLoop` documentation.

(S-8) *Counter and Log Data.* Furthermore, counters and logging variables are initialized. The matrix `yBestVec` stores the best function value found so far. It is initialized with the minimum value of the objective function on the initial design. Note, `ySurr`, which keeps track of the objective function values on the surrogate $\mathcal{S}$, has `NA` s, because no surrogate was built so far:

Program Code: Steps (S-5) to (S-8)

```
## (S-5) Enter spotLoop:

## (S-6) Initial check:
initialInputCheck(x, fun, lower, upper, control, inSpotLoop = TRUE)
dimension <- length(lower)
con <- spotControl(dimension)
con[names(control)] <- control
control <- con
rm(con)
control <- spotFillControlList(control, lower, upper)

## (S-7) Imputation:
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}

## (S-8) Counter and logs:
count <- nrow(y)
modelFit <- NA
ybestVec <- rep(min(y[, 1]), count)
ySurr <- matrix(NA, nrow = 1, ncol = count)
```

4.5.4 Entering the Main Loop

(S-9) *Termination Criteria, Conditions.* The main loop is entered as follows:

```
while ((count < control$funEvals) &
         (difftime(Sys.time(), control$time$startTime, units = 'mins')
          < control$time$maxTime))
```

Two termination criteria are implemented:

a. the number of objective function evaluations must be smaller than `funEvals` and
b. the time must be smaller than `maxTime`.

(S-10) *Subset Selection for the Surrogate.* Surrogates can be built with the full or a reduced set of available `x`- and `y`-values. A subset selection method, which is defined via `control$subsetSelect`, can be used before the surrogate $\mathcal{S}$is built. If `subsetSelect` is set to `selectAll`, which is the default, all points are used. Fitting the surrogate $\mathcal{S}$with a subset of the available points only appears to be counterintuitively, but can be reasonable, e.g., if the sample points are too close to each other or if the problem changes dynamically.

(S-11) *Fitting the Surrogate.* SPOT can use arbitrary regression models as surrogates, e.g., RF or GP models (Kriging).

The arguments `x` and `y` are mandatory for the function `model`. The `model` function must return a fit object that provides a `predict` method. A Gaussian process model, which performs well in many situations and can work well with discrete and continuous hyperparameters, is `SPOT`'s default model. Random forest is less suited as a surrogate for continuous parameters, as it has to approximate said parameters in a step-wise constant manner. The function `control$model` is applied to the `x`- and `y`-matrices. A default `model` is fitted to the data with the function `buildKriging`.

Program Code: Steps (S-9) to (S-11)

Steps (S-9) to (S-11) are implemented as follows:

```
## (S-9) Termination (while loop):

## (S-10) Subsect select:
selectRes <- control$subsetSelect(
  x = x,
  y = y[, 1, drop = FALSE],
  control = control$subsetControl
)

## (S-11) Surrogate fit:
modelFit <- control$model(
  x = selectRes$x,
  y = selectRes$y,
  control = control$modelControl
)
```

Table 4.5 Surrogates in `spot` require two arguments, `x` and `y`. The return values of the `build*` functions are shown below

Return Value	Type	Description
`x`	matrix	x values
`y`	matrix	y values
`fit`	object	Fitted model
`pNames`	character	Names of the independent variables
`yName`	character	Name of the dependent variable
`class`	character	Name of the model class

Background: Surrogates

There is a naming convention for surrogates in `spot`: functions names should start with the prefix "`build`". Surrogates in `spot` use the same interface. They accept the arguments `x`, `y`, which must be matrices, and the list `control`. They fit a model, e.g., `buildLM` uses the `lm`, which provides a method `predict`. Each model returns an object of the corresponding model class, here: `"spotLinearModel"`, with a `predict` method. The return value is implemented as a list with the entries from Table 4.5.

Note, `buildLM` is a very simple model. `SPOT`'s workhorse is a Kriging model, that is fitted via Maximum Likelihood Estimation (MLE). `buildKriging` is explained in Sect. 4.6.5.

(S-12) *Objective Function on the Surrogate (Predict).* After building the surrogate, the `modelFit` (surrogate model) is available. It is used to define the function `funSurrogate`, which works as an objective function on the surrogate $\mathcal{S}$: `funSurrogate` does not evaluate solutions on the original function f, but on the surrogate $\mathcal{S}$. Thus, `spot` searches for the hyperparameter configuration that is predicted to result in the best possible model quality. Therefore, an objective function is generated based on the `modelFit` via `predict`.

Program Code: Step (S-12)

Step (S-12) is implemented as follows:

```
## (S-12) Surrogate optimization function:
funSurrogate <- evaluateModel(
  modelFit,
  control$infillCriterion,
  control$verbosity
)
```

Background: Surrogate and Infill Criteria

The function `evaluateModel` generates an objective function that predicts function values on the surrogate. Some surrogate optimization procedures do not use the function values from the surrogate $\mathcal{S}$—they use an infill criterion instead.

Definition 4.2 (*Infill Criterion, Acquisition Function*) Infill criteria are methods that guide the exploration of the surrogate. They combine information from the predicted mean and the predicted variance generated by the GP model. In BO, the term "acquisition function" is used for functions that implement infill criteria.

For example, the function `buildKriging` provides three return values that can be used to generate elementary infill criteria. These return values are specified via the argument `target`, which is a vector of strings. Each string specifies a value to be predicted, e.g., `"y"` for mean, `"s"` for standard deviation, and `"ei"` for expected improvement. In addition to these elementary values, `spot` provides the function `infillCriterion` to specify user-defined criteria. The function `evaluateModel` that manages the infill criteria in `spot` is shown below.

```
evaluateModel <-
  function(object,
           infillCriterion = NULL) {
    evalModelFun <- function(x) {
      res <- predict(object = object, newdata = x)[object$target]
      return(res)
    }
    if (is.null(infillCriterion)) {
      return(function(x) {
        res <- evalModelFun(x)
        return(res)
      })
    } else {
      return(function(x) {
        return(infillCriterion(evalModelFun(x), object))
      })
    }
  }
```

Example: Expected Improvement

EI is a popular infill criterion, which was defined in Eq. (4.10). It is calculated as shown in Eq. (4.11) and can be called from `evaluateModel` via `modelControl = list(target = c("ei"))`. The following code shows an EI implementation that returns a vector with the negative logarithm of the expected improvement values, $-\log_{10}(\text{EI})$. The function `expectedImprovement`

is called, if the argument "ei" is selected as a target, e.g., spot(,fun,l,u,control=list(modelControl=list(target="ei"))).

```
expectedImprovement <- function(mean, sd, min) {
  EITermOne = (min - mean) * pnorm((min - mean) / sd)
  EITermTwo = sd * (1 / sqrt(2 * pi))
                 * exp(-(1 / 2) * ((min - mean) ^ 2 / (sd ^ 2)))
  - log10(EITermOne + EITermTwo + (.Machine$double.xmin))
}
```

(S-13) *Multiple Starting Points.* If the current best point is feasible, it is used as a starting point for the search on the surrogate $\mathcal{S}$. Because the surrogate can be multi-modal, multiple starting points are recommended. The function getMultiStartPoints implements a multi-start mechanism. spot provides the function getMultiStartPoints.
In addition to the current best point further starting points can be used. Their amount can be specified by the value of multiStart. If multiStart > 1, then additional starting points will be used. The design function, which was used for generating the initial design in Sect. 4.5.2, will be used here to generate additional points.

(S-14) *Optimization on the Surrogate.* The search on the surrogate $\mathcal{S}$can be performed next. The simplest objective function is optimLHD, which selects the point with the smallest function value from a relatively large set of LHD points. Other objective functions are available, e.g., optimLBFGS or optimDE. To find the next candidate solution, the predicted value of the surrogate is optimized via Differential Evolution (Storn and Price 1997). Other global optimization algorithms can be used as well. Even RS would be a feasible strategy.

> Mandatory Parameters

Optimization functions must use the same interface as spot, i.e., function(x, fun,lower,upper,control=list(),...). The arguments fun, lower, and upper are mandatory for optimization functions. This is similar to the interface of R's general-purpose optimization function optim.

As described in Sect. 4.5.4, the optimization on the surrogate $\mathcal{S}$can be performed with or without pre-defined starting points. We describe a search without starting points first.

(S-14a) *Search Without Starting Points.* If no starting points for the search are provided, the optimizer, which is specified via `control$optimizer`, is called.
The result from this optimization is stored in the list `optimResSurr`. The optimal value from the search on the surrogate is `optimResSurr$xbest`, the corresponding y-value is `optimResSurr$ybest`. Alternatively, the search on the surrogate can be performed with starting points.

(S-14b) *Search With Starting Points.* If starting points are used for the optimization on the surrogate, these are passed via `x = x0` to the optimizer. Several starting points result in several `optimResSurr$xbest` and `optimResSurr$ybest` values from which the best, i.e., the point with the smallest y-value, is selected.
For example, if `multiStart = 2` is selected, the current best and one random point will be used.
The optimization on the surrogate $\mathcal{S}$ is performed separately for each starting point and the matrix `xnew` is computed.
`xnew` is determined based on the multi-start results.

(S-15) *Compile Results from the Search on the Surrogate.* The function value of `xnew` (from (S-14a) or (S-14b)) is saved as `ySurrNew`. Note, this function values can be modified using `control$modelControl$target`, e.g., `"y"`, `"s2"`, or `"ei"`, i.e., the optimization on the surrogate can be based on the predicted new value `"y"`, a combination of `"y"` and the variance or the EI `"ei"`.

(S-16) *Noise, Repeats, and Consistency Checks for New Points.* After the new solution candidate `xnew` and its associated function value on the surrogate `ySurrNew` have been determined, `spot` checks for duplicates and determines the number of replicates. This step treats noisy and deterministic objective functions in a different way.
If `control$noise == TRUE`, then replicates are allowed, i.e., a single solution x can be evaluated several times. If `control$noise == FALSE`, then every solution is evaluated only once.

Program Code: Steps (S-13) to (S-16)

Steps (S-13) to (S-16) are implemented as follows:

```
## (S-13) Random starting points for optimization on the surrogate
x0 <- getMultiStartPoints(x, y, control)
resSurr <- matrix(NA, nrow = nrow(x0), ncol = ncol(x0) + 1)

## (S-14b) Search on the surrogate with starting point/s x0:
for (i in 1:nrow(x0)) {
  optimResSurr <- control$optimizer(
    x = x0[i, , drop = FALSE],
    funSurrogate,
    lower,
```

```
    upper,
    control$optimizerControl
  )
  resSurr[i, ] <- c(optimResSurr$xbest, optimResSurr$ybest)
}

## (S-15) Compile surrogate results:
m <- which.min(resSurr[, ncol(x) + 1])
## Determine xnew based on multi start results
xnew <- resSurr[m, 1:ncol(x), drop = FALSE]
## value on the surrogate (can be "y", "s2, "ei", "negLog10ei" etc.)
ySurrNew <- resSurr[m, ncol(x) + 1]


## (S-16) Duplicate handling:
xnew <- duplicateAndReplicateHandling(xnew, x, lower, upper, control)
# Repair non-numeric results
xnew <- repairNonNumeric(xnew, control$types)
```

Background: Duplicates and Replicates

The function `duplicateAndReplicateHandling` checks whether the new solution `xnew` has been evaluated before. In this case, it is taken as it is and no additional evaluations are performed. If `xnew` was not evaluated before, it will be evaluated. The number of evaluations is defined via `control$replicates`. Duplicate and replicate handling in `spot` depends on the setting of the parameter `noise`. If the value is `TRUE` then a test whether `xnew` is new or has been evaluated before is performed. If `xnew` is new (was not evaluated before), then it should be evaluated `replicates` times. Assume, `control$replicates` < -3, i.e., three initial replicates are required and `xnew` was not evaluated before. Then two additional evaluations should be done, i.e., `xtmp` contains two entries which are combined with one already existing entry in `xnew`.

```
control$replicates <- 3
xtmp <- NULL
for (i in 1:nrow(xnew)) {
  if (!any(apply(x, 1, identical, xnew[i, ]))) {
    xtmp <- rbind(xtmp, xnew[rep(i, control$replicates - 1), ])
  }
}
xnew <- rbind(xnew, xtmp)
xnew
  ##             [,1]        [,2]
  ## [1,] -0.01292055 -0.02666901
  ## [2,] -0.01292055 -0.02666901
  ## [3,] -0.01292055 -0.02666901
```

If the parameter noise has the value FALSE, two cases have to be distinguished. First, if xnew was not evaluated before, then it should be evaluated once (and not replicates times), because additional evaluation is useless. They would deterministically generate the same result.

```
for (i in 1:nrow(xnew)) {
  if (any(apply(x, 1, identical, xnew[i, ]))) {
    warning("Duplicate is replaced by random solution.")
    control$designControl$replicates <- 1
    control$designControl$size <- 1
    xnew[i, ] <-
      designUniformRandom(, lower, upper, control$designControl)
  }
}
xnew
  ##             [,1]        [,2]
  ## [1,] -0.01292055 -0.02666901
  ## [2,] -0.01292055 -0.02666901
  ## [3,] -0.01292055 -0.02666901
```

Second, if xnew was evaluated before, then a warning is issued and a randomly generated solution for each entry in xnew will be used.

```
# xnew has two already known solutions:
xnew <- x[1:2, ]
for (i in 1:nrow(xnew)) {
  if (any(apply(x, 1, identical, xnew[i, ]))) {
    warning("Duplicate is replaced by random solution.")
    control$designControl$replicates <- 1
    control$designControl$size <- 1
    xnew[i, ] <-
      designUniformRandom(, lower, upper, control$designControl)
  }
}
xnew
  ##           [,1]      [,2]
  ## [1,] 0.6704420 0.1762307
  ## [2,] 0.5094811 0.3023359
```

A type check is performed, i.e., all non-numeric values produced by the optimizer are rounded.

(S-17) *OCBA for Known Points.* OCBA is called next if OCBA and noise are both set to TRUE: the function repeatsOCBA returns a vector that specifies how often each known solution should be re-evaluated (or replicated). This function can spend a budget of control$OCBABudget additional evaluations. The solutions proposed by repeatsOCBA are added to the set of new x candidates xnew. Because OCBA calculates an estimate of the variance, it is

based on evaluated solutions and their function values, i.e., `x` and `y` values respectively.

Program Code: Step (S-17)

Step (S-17) is implemented as follows:

```
## (S-17) OCBA:
if (control$noise &
  control$OCBA) {
  xnew <- rbind(xnew, repeatsOCBA(x, y[, 1, drop = FALSE], control$OCBABudget))
}
```

Background: Optimal Computational Budget Allocation

OCBA is a very efficient solution to solve the "general ranking and selection problem" if the objective function is noisy (Chen 2010; Bartz-Beielstein et al. 2011). It allocates function evaluations in an uneven manner to identify the best solutions and to reduce the total optimization costs.

Theorem 4.1 *Given a total number of optimization samples N to be allocated to k competing solutions whose performance is depicted by random variables with means $\bar{y}_i$ $(i = 1, 2, \ldots, k)$, and finite variances σ_i^2, respectively, as $N \to \infty$, the Approximate Probability of Correct Selection (APCS) can be asymptotically maximized when*

$$\frac{N_i}{N_j} = \left(\frac{\sigma_i/\delta_{b,i}}{\sigma_j/\delta_{b,j}}\right)^2, i, j \in \{1, 2, \ldots, k\}, \text{and } i \neq j \neq b, \tag{4.1}$$

$$N_b = \sigma_b \sqrt{\sum_{i=1, i\neq b}^{k} \frac{N_i^2}{\sigma_i^2}}, \tag{4.2}$$

*where N_i is the number of replications allocated to solution i, $\delta_{b,i} = \bar{y}_b - \bar{y}_i$, and $\bar{y}_b \leq \min_{i\neq b} \bar{y}_i$ (*Chen 2010*).*

(S-18) *Evaluating New Solutions.* To avoid exceeding the available budget of objective function evaluations, which is specified via `control$funEvals`, a check is performed. Solution candidates are passed to the function `objectiveFunctionEvaluation`, which calculates the associated objective function values `ynew` on the function `fun`.

(S-19) *Imputation.* Because the evaluation of solution candidates might result in infinite `Inf` or Not-a-Number `NaN` `ynew` values, the function `imputeY`, which handled non-numeric values, is called.

(S-20) *Update Counter and Log Data.* Next, counters `count` and `ySurr`, information about the function values on the surrogate $\mathcal{S}$, are updated.
Calculation of the progress and preparation of progress plots conclude the main loop. The last step of the main loop compiles the list `return`, which is returned to the `spot` function.

(S-21) *Reporting after the While-Loop.* After the while loop is finished, results are compiled. Some objective functions return several values (Multi Objective Optimization (MOO)). The corresponding values are stored as `logInfo`, because the default `spot` function uses only one objective function value. This mechanism enables `spot` handling MOO problems. The values of the transformed parameters are stored as `xt`. Important for noisy optimization is the following feature: OCBA can be used for the selection of the best value. The function `ocbaRanking` computes the best `x` and `y` values, `xBestOcba` and `yBestOcba`, respectively. `yBestOcba` is the mean value of the corresponding x-parameter setting `xBestOcba`.

Program Code: Steps (S-18) to (S-22)

Steps (S-18) to (S-22) are implemented as follows:

```
## (S-18) Evaluate xnew:
ynew <- tryCatch(
  expr = {
    objectiveFunctionEvaluation(
      x = x,
      xnew = xnew,
      fun = fun,
      control = control
    )
  },
  error = function(e) {
    if (!is.null(control$yImputation$handleNAsMethod)) {
      n <- nrow(xnew)
      m <- ncol(y)
      return(matrix(rep(NA, m * n), nrow = n))
    }
  }
)
## (S-19) Impute:
colnames(xnew) <- colnames(x)
x <- rbind(x, xnew)
y <- rbind(y, ynew)
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}
## (S-20) Update counter, logs, etc.:
```

```
ySurr <- c(ySurr, ySurrNew)
count <- count + nrow(ynew)
indexBest <- which.min(y[, 1, drop = FALSE])
ybestVec <- c(ybestVec, y[indexBest, 1, drop = FALSE])
## END while loop
## (S-21) Reporting after while loop in spotLoop
if (ncol(y) > 1) {
  logInfo <- y[, -1, drop = FALSE]
} else {
  logInfo <- NA
}
if (length(control$transformFun) > 0) {
  xt <- transformX(xNat = x, fn = control$transformFun)
} else {
  xt <- NA
}
# (S-22) OCBA-best selection:
if (control$noise & control$OCBA) {
  ocbaRes <- ocbaRanking(
    x = x,
    y = y,
    fun = fun,
    control = control
  )
  control$xBestOcba <- ocbaRes[1, 1:(ncol(ocbaRes) - 1)]
  control$yBestOcba <- ocbaRes[1, ncol(ocbaRes)]
}
# Compile results in spotLoop
result <- list(
  xbest = x[indexBest, , drop = FALSE],
  ybest = y[indexBest, 1, drop = FALSE],
  xBestOcba = matrix(control$xBestOcba, ncol = length(lower)),
  yBestOcba = matrix(control$yBestOcba, ncol = length(lower)),
  x = x,
  xt = xt,
  y = y[, 1, drop = FALSE],
  logInfo = logInfo,
  count = count,
  msg = "budget exhausted",
  modelFit = modelFit,
  ybestVec = ybestVec,
  ySurr = ySurr
)
## END spotLoop()
```

The function `spotLoop` ends here and the final steps of the main function `spot`, which are summarized in the following section, are executed.

Table 4.6 `spot`: return parameters

Parameter	Value, type	Description
`xbest`	matrix	Best `x` values
`ybest`	matrix	Best `y` values
`xBestOcba`	matrix	Best `x` values
`yBestOcba`	matrix	Best `y` values
`x`	matrix	`x` values
`xt`	matrix	Transformed `x` values
`y`	matrix	`y` values
`logInfo`	matrix	Additional `y` information, also multi-objective values
`count`	integer	Number of function evaluations
`msg`	character	Information about the optimization
`modelFit`		
`yBestVec`	matrix	History of best `y` values
`ySurr`	matrix	`y` values on the surrogate
`control`	list	Control parameters

4.5.5 Final Steps

To exploit the region of the best solution from the surrogate, $\mathcal{S}$, which was determined during the SMBO in the main loop with `spotLoop`, `SPOT` allows a local optimization step. If `control$directOptControl$funEvals` is larger than zero, this optimization is started. If the best solution from the surrogate, `xbest`, satisfies the inequality constraints, it is used as a starting point for the local optimization with the local optimizer `control$directOpt`. For example, `directOpt = optimNLOPTR` or `directOpt = optimLBFGSB`, can be used.

Results from the direct optimization will be appended to the matrices of the `x` and `y` values based on SMBO. SPOT returns the gathered information in a list (Table 4.6). Because SPOT focuses on reliability and reproducibility, it is not the speediest algorithm.

4.6 Kriging

Basic elements of the Kriging-based surrogate optimization such as interpolation, expected improvement, and regression are presented. The presentation follows the approach described in Forrester et al. (2008a).

4.6.1 The Kriging Model

Consider sample data $\mathbf{X}$ and $\mathbf{y}$ from n locations that are available in matrix form: $\mathbf{X}$ is a $(n \times k)$ matrix, where k denotes the problem dimension and $\mathbf{y}$ is a $(n \times 1)$ vector. The observed responses $\mathbf{y}$ are considered as if they are from a stochastic process, which will be denoted as

$$\begin{pmatrix} \mathbf{Y}(\mathbf{x}^{(1)}) \\ \vdots \\ \mathbf{Y}(\mathbf{x}^{(n)}) \end{pmatrix}.$$

The set of random vectors (also referred to as a "random field") has a mean of $\mathbf{1}\mu$, which is a $(n \times 1)$ vector. The random vectors are correlated with each other using the basis function expression

$$\text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(i)}), \mathbf{Y}(\mathbf{x}^{(l)})\right) = \exp\left\{-\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j^{(l)}|^{p_j}\right\}.$$

The $(n \times n)$ correlation matrix of the observed sample data is

$$\mathbf{\Psi} = \begin{pmatrix} \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(i)}), \mathbf{Y}(\mathbf{x}^{(l)})\right) & \dots & \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(i)}), \mathbf{Y}(\mathbf{x}^{(l)})\right) \\ \vdots & \vdots & \vdots \\ \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(i)}), \mathbf{Y}(\mathbf{x}^{(l)})\right) & \dots & \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(i)}), \mathbf{Y}(\mathbf{x}^{(l)})\right) \end{pmatrix}. \tag{4.3}$$

Note: correlations depend on the absolute distances between sample points $|x_j^{(n)} - x_j^{(n)}|$ and the parameters p_j and θ_j.

To estimate the values of $\boldsymbol{\theta}$ and $\mathbf{p}$, they are chosen to maximize the likelihood of $\mathbf{y}$, which can be expressed as

$$L\left(\mathbf{Y}(\mathbf{x}^{(1)}), \dots, \mathbf{Y}(\mathbf{x}^{(n)})|\mu, \sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}} \exp\left\{\frac{-\sum_{j=1}^{n}\left(\mathbf{Y}^{(j)} - \boldsymbol{\mu}\right)^2}{2\sigma^2}\right\},$$

which can be expressed in terms of the sample data

$$L\left(\mathbf{Y}(\mathbf{x}^{(1)}), \dots, \mathbf{Y}(\mathbf{x}^{(n)})|\mu, \sigma\right) = \frac{1}{(2\pi\sigma)^{n/2}|\mathbf{\Psi}|^{1/2}} \exp\left\{\frac{-(\mathbf{y} - \mathbf{1}\mu)^T \mathbf{\Psi}^{-1}(\mathbf{y} - \mathbf{1}\mu)}{2\sigma^2}\right\},$$

and formulated as the log-likelihood:

$$\ln(L) = -\frac{n}{2}\ln(2\pi\sigma) - \frac{1}{2}\ln|\mathbf{\Psi}| \frac{-(\mathbf{y} - \mathbf{1}\mu)^T \mathbf{\Psi}^{-1}(\mathbf{y} - \mathbf{1}\mu)}{2\sigma^2}. \tag{4.4}$$

Optimization of the log-likelihood by taking derivatives with respect to μ and σ results in

$$\hat{\mu} = \frac{\mathbf{1}^T \boldsymbol{\Psi}^{-1} \mathbf{y}^T}{\mathbf{1}^T \boldsymbol{\Psi}^{-1} \mathbf{1}^T} \tag{4.5}$$

and

$$\hat{\sigma} = \frac{(\mathbf{y} - \mathbf{1}\mu)^T \boldsymbol{\Psi}^{-1} (\mathbf{y} - \mathbf{1}\mu)}{n}. \tag{4.6}$$

Substituting (4.5) and (4.6) into (4.4) leads to the concentrated log-likelihood:

$$\ln(L) = -\frac{n}{2} \ln(\hat{\sigma}) - \frac{1}{2} \ln |\boldsymbol{\Psi}|. \tag{4.7}$$

Note: To maximize $\ln(L)$, optimal values of $\boldsymbol{\theta}$ and $\mathbf{p}$ are determined numerically, because (4.7) is not differentiable.

4.6.2 Kriging Prediction

For a new prediction $\hat{y}$ at $\mathbf{x}$, the value of $\hat{y}$ is chosen so that it maximizes the likelihood of the sample data $\mathbf{X}$ and the prediction, given the correlation parameter $\boldsymbol{\theta}$ and $\mathbf{p}$. The observed data $\mathbf{y}$ is augmented with the new prediction $\hat{y}$ which results in the augmented vector $\tilde{\mathbf{y}} = (\mathbf{y}^T, \hat{y})^T$. A vector of correlations between the observed data and the new prediction is defined as

$$\boldsymbol{\psi} = \begin{pmatrix} \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(1)}), \mathbf{Y}(\mathbf{x})\right) \\ \vdots \\ \text{Cor}\left(\mathbf{Y}(\mathbf{x}^{(n)}), \mathbf{Y}(\mathbf{x})\right) \end{pmatrix} = \begin{pmatrix} \boldsymbol{\psi}^{(1)} \\ \vdots \\ \boldsymbol{\psi}^{(n)} \end{pmatrix}.$$

The augmented correlation matrix is constructed as

$$\tilde{\boldsymbol{\Psi}} = \begin{pmatrix} \boldsymbol{\Psi} & \boldsymbol{\psi} \\ \boldsymbol{\psi}^T & 1 \end{pmatrix}.$$

Similar to (4.4), the log-likelihood of the augmented data is

$$\ln(L) = -\frac{n}{2} \ln(2\pi) - \frac{n}{2} \ln(\hat{\sigma}^2) - \frac{1}{2} \ln |\hat{\tilde{\boldsymbol{\Psi}}}| - \frac{(\tilde{\mathbf{y}} - \mathbf{1}\hat{\mu})^T \tilde{\boldsymbol{\Psi}}^{-1} (\tilde{\mathbf{y}} - \mathbf{1}\hat{\mu})}{2\hat{\sigma}^2}. \tag{4.8}$$

The MLE for $\hat{y}$ can be calculated as

$$\hat{y}(\mathbf{x}) = \hat{\mu} + \boldsymbol{\psi}^T \tilde{\boldsymbol{\Psi}}^{-1} (\mathbf{y} - \mathbf{1}\hat{\mu}). \tag{4.9}$$

Equation 4.9 reveals two important properties of the Kriging predictor.

- The basis function impacts the vector $\boldsymbol{\psi}$, which contains the n correlations between the new point $\mathbf{x}$ and the observed locations. Values from the n basis functions are added to a mean base term μ with weightings $\mathbf{w} = \tilde{\boldsymbol{\Psi}}^{(-1)}(\mathbf{y} - \mathbf{1}\hat{\mu})$.
- The predictions interpolate the sample data. When calculating the prediction at the ith sample point, $\mathbf{x}^{(i)}$, the ith column of $\boldsymbol{\Psi}^{-1}$ is $\boldsymbol{\psi}$, and $\boldsymbol{\psi}\boldsymbol{\Psi}^{-1}$ is the ith unit vector. Hence, $\hat{y}(\mathbf{x}^{(i)}) = y^{(i)}$.

4.6.3 Expected Improvement

The EI is a criterion for error-based exploration, which uses the MSE of the Kriging prediction. The MSE is calculated as

$$s^2(\mathbf{x}) = \sigma^2 \left(1 - \boldsymbol{\psi}^T \boldsymbol{\Psi}^{-1} \boldsymbol{\psi} + \frac{(1 - \mathbf{1}^T \boldsymbol{\Psi}^{-1} \boldsymbol{\Psi})^2}{\mathbf{1}^T \boldsymbol{\Psi}^{-1} \mathbf{1}}\right).$$

Here, $s^2(\mathbf{x}) = 0$ at sample points, and the last term is omitted in Bayesian settings.

Since the EI extends the Probability of Improvement (PI), it will be described first. Let $y_{\min}$ denote the best-observed value so far and consider $\hat{y}(\mathbf{x})$ as the realization of a random variable. Then, the probability of an improvement $I = y_{\min} - \mathbf{Y}(\mathbf{x})$ can be calculated as

$$P(I(\mathbf{x})) = \frac{1}{2}\left\{1 + \operatorname{erf}\left(\frac{y_{\min} - \hat{\mathbf{y}}(\mathbf{x})}{\hat{s}\sqrt{2}}\right)\right\}.$$

The EI does not calculate the probability that there will be some improvement, it calculates the amount of expected improvement. The rationale of using this expectation is that we are less interested in highly probable improvement if the magnitude of that improvement is very small. The EI is defined as follows.

Definition 4.3 (*Expected Improvement*)

$$E(I(\mathbf{x})) = \begin{cases} (y_{\min} - \hat{y}(\mathbf{x}))\Phi\left(\frac{y_{\min} - \hat{y}(\mathbf{x})}{\hat{s}(\mathbf{x})}\right) + \hat{s}\phi\left(\frac{y_{\min} - \hat{y}(\mathbf{x})}{\hat{s}(\mathbf{x})}\right) & \text{if } \hat{s} > 0 \\ 0 & \text{if } \hat{s} = 0 \end{cases}, \tag{4.10}$$

where $\Phi(.)$ and $\phi(.)$ are the Cumulative Distribution Function (CDF) and Probability Distribution Function (PDF), respectively.

The EI is evaluated as

$$E(I(\mathbf{x})) = \left(y_{\min} - \hat{y}(\mathbf{x})\right)\frac{1}{2}\left\{1 + \operatorname{erf}\left(\frac{y_{\min} - \hat{\mathbf{Y}}(\mathbf{x})}{\hat{s}\sqrt{2}}\right)\right\} + \hat{s}\frac{1}{\sqrt{2\pi}}\exp\left\{\frac{-(y_{\min} - \hat{y}(\mathbf{x}))^2}{2\hat{s}^2}\right\}. \tag{4.11}$$

4.6.4 Infill Criteria with Noisy Data

The EI infill criterion was formulated under the assumption that the true underlying function is deterministic, smooth, and continuous. In deterministic settings, the Kriging predictor should interpolate the data. Noise can complicate the modeling process: predictions can become erratic, because there is a high MSE in regions far away from observed data. Therefore, the interpolation property should be dropped to filter noise. A regression constant, λ, is added to the diagonal of $\mathbf{\Psi}$ and $\mathbf{\Psi} + \lambda\mathbf{I}$ is used. Then, $\mathbf{\Psi} + \lambda\mathbf{I}$ does not contain $\boldsymbol{\psi}$ as a column and the data is not interpolated. The same method of derivation as in interpolating Kriging (Eq. 4.9) can be used for regression Kriging. The regression Kriging prediction is given by

$$\hat{y_r}(\mathbf{x}) = \hat{\mu_r} + \boldsymbol{\psi}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu_r}),$$

where

$$\hat{\mu_r} = \frac{\mathbf{1}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}\mathbf{y}}{\mathbf{1}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}\mathbf{1}}.$$

Including the regression constant λ the following equation allows the calculation of an estimate of the error in the Kriging regression model for noisy data:

$$\hat{s_r^2}(\mathbf{x}) = \hat{\sigma_r^2}\left\{1 + \lambda - \boldsymbol{\psi}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}\boldsymbol{\psi} + \frac{(1 - \mathbf{1}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}\boldsymbol{\psi})^2}{\mathbf{1}^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}\mathbf{1}}\right\}, \tag{4.12}$$

where

$$\hat{\sigma_r^2} = \frac{(\mathbf{y} - \mathbf{1}\hat{\mu_r})^T(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu_r})}{n}.$$

Note: Eq. (4.12) includes the error associated with noise in the data. There is nonzero error in all areas which leads to nonzero EI in all areas. As a consequence, resampling can occur. Resampling can be useful if replicates result in different outcomes. Although the possibility of resampling can destroy the convergence to the global optimum, resampling can be a wanted feature in optimization with noisy data. In a deterministic setting, resampling is an unwanted feature, because new evaluations of the same point do not provide additional information and can stall the optimization process.

Re-interpolation can be used to eliminate the errors due to noise in the data from the model. Re-interpolation bases the estimated error on an interpolation of points predicted by the regression model at the sample locations. It proceeds as follows: calculate values for the Kriging regression at the sample locations using

$$\hat{\mathbf{y}}_r = \mathbf{1}\hat{\mu} + \mathbf{\Psi}(\mathbf{\Psi} + \lambda\mathbf{I})^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu}).$$

This vector can be substituted into Eq. (4.9), which is substituted into (4.6). This results in

$$\hat{\sigma}^2_{\mathrm{ri}} = \frac{(\mathbf{y} - \mathbf{1}\hat{\mu})^T(\boldsymbol{\Psi} + \lambda\mathbf{I})^{-1}\boldsymbol{\Psi}(\boldsymbol{\Psi} + \lambda\mathbf{I})^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu})}{n}.$$

Using the interpolating Kriging error estimate (4.12), the re-interpolation error estimate reads

$$\hat{s}^2_{\mathrm{ri}}(\mathbf{x}) = \hat{\sigma}^2_{\mathrm{ri}}\left\{1 - \boldsymbol{\psi}^T\boldsymbol{\Psi}^{-1}\boldsymbol{\psi} + \frac{(1 - \mathbf{1}^T(\boldsymbol{\Psi} + \lambda\mathbf{I})^{-1}\boldsymbol{\psi})^2}{\mathbf{1}^T(\boldsymbol{\Psi} + \lambda\mathbf{I})^{-1}\mathbf{1}}\right\}.$$

4.6.5 *spot's Workhorse: Kriging*

This section explains the implementation of the function `buildKriging` in `SPOT`.

(K-1) *Set Parameters.* `buildKriging` uses the parameters shown in Table 4.7. It returns an object of class `kriging`, which is basically a list, with the options and found parameters for the model which has to be passed to the `predict` method of this class.

Program Code: Step (K-1)

```
buildKriging <- function(x, y, control = list()) {
  ## (K-1) Set Parameters
  k <- ncol(x) # dimension
  n <- nrow(x) # number of observations
  con <- list(
    thetaLower = 1e-4,
    thetaUpper = 1e2,
    types = rep("numeric", k),
    algTheta = optimDE,
    budgetAlgTheta = 200,
    optimizeP = FALSE,
    useLambda = TRUE,
    lambdaLower = -6,
    lambdaUpper = 0,
    startTheta = NULL,
    reinterpolate = TRUE,
    target = "y"
  )
  fit <- control
  fit$x <- x
  fit$y <- y
  LowerTheta <- rep(1, k) * log10(fit$thetaLower)
  UpperTheta <- rep(1, k) * log10(fit$thetaUpper)
```

Table 4.7 `buildKriging`: besides the design matrix `x` with corresponding observations `y`, the function accepts a list with the parameters shown below

Parameter	Value, type	Description
`types`	character vector	A character vector giving the data type of each variable. All but `factor` will be handled as numeric, `factor` (categorical) variables will be subject to the Hamming distance
`thetaLower`	1e-4, numerical	Lower boundary for `theta`
`thetaUpper`	1e2	Upper boundary for `theta`
`algTheta`	`optimDE`, function	Algorithm used to find `theta` via MLE
`budgetAlgTheta`	`200`, integer	Budget for the algorithm `algTheta`. The value will be multiplied with the length of the model parameter vector to be optimized
`optimizeP`	`FALSE`, logical	Specifies whether the exponents (`p`) should be optimized. Otherwise, they will be set to two
`useLambda`	`TRUE`, logical	Whether to use the regularization constant lambda (nugget effect)
`lambdaLower`	`-6`, numerical	Lower boundary for `log10(lambda)`
`lambdaUpper`	`0`, numerical	Upper boundary for `log10(lambda)`
`startTheta`	`NULL`, numerical	Optional start value for theta optimization
`reinterpolate`	`TRUE`, logical	Whether re-interpolation should be performed
`target`	`"y"`, character vector	Values of the prediction. Each element specifies a value to be predicted, e.g., `"y"` for mean, `"s"` for standard deviation, `"ei"` for EI

(K-2) *Normalization.*
The function `normalizeMatrix` is used to normalize the data, i.e., each column of the (n, k)-matrix X has values in the range from zero to one.

Program Code: Step (K-2)

```
## (K-2) Normalize input data
fit$normalizeymin <- 0
fit$normalizeymax <- 1
res <- normalizeMatrix(fit$x, ymin, ymax)
fit$scaledx <- res$y
fit$normalizexmin <- res$xmin
fit$normalizexmax <- res$xmax
```

(K-3) *Correlation Matrix.* Prepare correlation matrix Ψ (Eq. (4.3)) and start points for the optimization. The distance matrix is determined. The i-th row of (k, n^2)-

matrix A contains the distances between the elements of the i-th column (dimension). $A(1, 1)$ is the distance of the first element to the first element in the first dimension, $A(1, 2)$ the distance of the first element to the second element in the first dimension, $A(1, n + 1)$ is the distance of the second element to the first element in the first dimension, and so on.

Program Code: Step (K-3)

```
## (K-3) Prepare distance/correlation matrix
A <- matrix(0, k, n * n)
for (i in 1:k) {
  if (control$types[i] != "factor") {
    A[i, ] <-
      as.numeric(as.matrix(dist(fit$scaledx[, i]))) # euclidean distance
  } else {
    tmp <-
      outer(fit$scaledx[, i], fit$scaledx[, i], "!=") # hamming distance
    class(tmp) <- "numeric"
    A[i, ] <- tmp
  }
}
```

(K-4) *Prepare Starting Points.*

(K-4.1) θ. The starting point for the optimization of θ is determined. If no explicit starting point is specified, then

$$\theta_0 = n/(100k) \tag{4.13}$$

is chosen.

(K-4.2) p. The parameter `optimizeP` determines whether p should be optimized or not. In the latter case, $p = 2$ is set and the matrix A is squared. Otherwise, the starting point for the optimization of p is chosen as $p_0 = 1.9$ and the search interval is set to $[0.01, 2]$.

(K-4.3) *Nugget.* If a nugget effect should be integrated, the starting point for the optimization of λ is set to

$$\lambda_0 = \frac{\lambda_{\text{lower}} + \lambda_{\text{upper}}}{2} \tag{4.14}$$

(K-4.4) *Penalty.* The penalty value is set to

$$\phi = n \times \log(\text{Var}(y)) + 1e4. \tag{4.15}$$

Note: this penalty value should not be a hard constant. The scale of the likelihood, i.e., $n \times \log(SigmaSqr) + LnDetPsi$ at least depends on $\log(\mathrm{Var}(y))$ and the number of samples. Hence, large number of samples may lead to cases where the penalty is lower than the likelihood of most valid parameterizations. A suggested penalty is therefore $\phi = n \times \log(\mathrm{Var}(y)) + 1e4$. Currently, this penalty is set in the `buildKriging` function, when calling `krigingLikelihood`.

Program Code: Step (K-4)

```
## (K-4) Prepare starting points, search bounds and penalty value
## for MLE optimization
## 4.1 theta
x1 <- rep(n / (100 * k), k) # start point for theta
## 4.2 p
LowerTheta <- c(LowerTheta, rep(1, k) * 0.01)
UpperTheta <- c(UpperTheta, rep(1, k) * 2)
x3 <- rep(1, k) * 1.9 # start values for p
x0 <- c(x1, x3)
## 4.3 lambda
# start value for lambda:
x2 <- (fit$lambdaUpper + fit$lambdaLower) / 2
x0 <- c(x0, x2)
# append regression constant lambda (nugget)
LowerTheta <- c(LowerTheta, fit$lambdaLower)
UpperTheta <- c(UpperTheta, fit$lambdaUpper)
x0 <- matrix(x0, 1) # matrix with one row
opts <- list(funEvals = fit$budgetAlgTheta * ncol(x0))
## 4.4 penalty
penval <- n * log(var(y)) + 1e4
```

(K-5) *Objective.* The objective function `fitFun` for the MLE optimizer `algTheta` is defined in this step.

(K-6) *krigingLikelihood.* The function `krigingLikelihood`, see Sect. 4.6.6, is called.

Program Code: Steps (K-5) and (K-6)

```
## (K-5) MLE objective function
fitFun <-
  function(x, fX, fy, optimizeP, useLambda, penval) {
    krigingLikelihood(x, fX, fy, optimizeP, useLambda, penval)$NegLnLike
  }

## (K-6) See krigingLikelihood
```

(K-7) *Performing the Optimization with* `fitFun`. The optimizer is called as follows:

Program Code: Step (K-7)

```
## (K-7)  MLE optimization
res <- fit$algTheta(
  x = x0,
  fun =
    function(x, fX, fy, optimizeP, useLambda, penval) {
      apply(x, 1, fitFun, fX, fy, optimizeP, useLambda, penval)
    },
  lower = LowerTheta,
  upper = UpperTheta,
  control = opts,
  fX = A,
  fy = fit$y,
  optimizeP = fit$optimizeP,
  useLambda = fit$useLambda,
  penval = penval
)
```

(K-8) *Compile Results.* Step Compile return values: The return values from the optimization run, which are stored in the list `res`, are added to the list `fit` that specifies the object of the class `kriging`. The list `fit` contains the following optimized values: θ^* as `Theta`, 10^{θ^*} as `dmodeltheta`, p^*, as `P`, λ^*, as `Lambda` and 10^{λ^*}, as `dmodellambda`.

Program Code: Step (K-8)

```
## (K-8) Compile results from MLE optimization (to fit object)
Params <- res$xbest
nevals <- as.numeric(res$count[[1]])
fit$Theta <- Params[1:k]
fit$dmodeltheta <- 10^Params[1:k]
fit$P <- Params[(k + 1):(2 * k)]
fit$Lambda <- Params[length(Params)]
fit$dmodellambda <- 10^Params[length(Params)]
```

(K-9) *Use Results to Determine Likelihood and Best Parameters.* The function `krigingLikelihood` is called with these optimized values, θ^*, p^*, and λ^* to determine the values used for the fit of the Kriging model.

Program Code: Step (K-9)

```
## (K-9) Evaluate with optimized parameters
res <-
  krigingLikelihood(
    c(fit$Theta, fit$P, fit$Lambda),
    A,
    fit$y,
    fit$optimizeP,
    fit$useLambda
  )
```

(K-10) *Compile the fit Object.* The return values from this call to `krigingLikelihood` are added to the `fit` object.

Program Code: Step (K-10)

```
## (K-10) Add results from MLE evaluation to fit object
fit$yonemu <- res$yonemu
fit$ssq <- as.numeric(res$ssq)
fit$mu <- res$mu
fit$Psi <- res$Psi
fit$Psinv <- res$Psinv
fit$nevals <- nevals
```

```
fit$like <- res$NegLnLike
fit$returnCrossCor <- FALSE
```

(K-11) *Calculate the mean objective function value.* In addition to the results from the MLE optimization, the mean objective function value of the best x value, $y_{\min}$, is calculated and stored in the `fit` list as `min`. This value is needed for the EI computation.
Now the `fit` is available and can be used for predictions. The corresponding code is shown below.

4.6.6 krigingLikelihood

Step: MLE optimization with `krigingLikelihood`. The objective function accepts the following parameters: `x`, a vector, which contains the parameters log10(theta), log10(lambda), and p, `AX`, a three-dimensional array, constructed by `buildKriging` from the sample locations, `Ay`, a vector of observations at sample locations, `optimizeP`, logical, which specifies whether or not to optimize parameter `p` (exponents) or fix at two, `useLambda`, logical, which specifies whether to use the nugget, and `penval`, a penalty value which affects the value returned for invalid correlation matrices or configurations. The function `krigingLikelihood` performs the following calculations: The θ and λ values are updated:

$$\theta_j = 10^{\theta_0} \quad (j = 1, \ldots, n) \tag{4.16}$$

$$\lambda = 10^{\lambda} \tag{4.17}$$

$$AX[j,] = |(|\,AX)_j^p \quad (j = 1, \ldots, n) \tag{4.18}$$

(L-1) *Starting Points.*
(L-2) *Correlation Matrix* Ψ. The matrix Ψ can be calculated. If `useLambda == TRUE`, the nugget effect λ is added.
(L-3) *Cholesky Factorization.* Since $\Psi > 0$, its Cholesky factorization is computed.
(L-4) *Determinant.* The natural log of the determinant of Ψ, `LnDetPsi` is calculated, because it is numerically more reliable and also faster than using `det` or `determinant`.
(L-5) *Matrix inverse, mean, error, and likelihood.*
Using `chol2inv`, the following values can be calculated: $\ln(L)$ (Eq. (4.7)), $\hat{\mu}$ (Eq. (4.5)), and $\hat{\sigma}$ (Eq. (4.6)). Together with the matrices Ψ and Ψ^{-1}, and the vector 1μ, these values are combined into a list, which is returned from the function `krigingLikelihood`.

The following code illustrates the main components of `krigingLikelihood`.

Program Code: krigingLikelihood Function

```
krigingLikelihood <-
  function(x,
           AX,
           Ay,
           optimizeP = FALSE,
           useLambda = TRUE,
           penval = 1e8) {
    ## (L-1) Starting Points
    nx <- nrow(AX)
    theta <- 10^x[1:nx]
    if (optimizeP) {
      AX <- abs(AX)^(x[(nx + 1):(2 * nx)])
    }
    lambda <- 0
    if (useLambda) {
      lambda <- 10^x[length(x)]
    }
    n <- dim(Ay)[1]
    ## (L-2) Correlation Matrix Psi
    Psi <- exp(-matrix(colSums(theta * AX), n, n))
    if (useLambda) {
      Psi <- Psi + diag(lambda, n)
    }
    ## (L-3) cholesky decomposition
    cholPsi <- try(chol(Psi), TRUE)
    ## (L-4) Determininant
    LnDetPsi <- 2 * sum(log(abs(diag(cholPsi))))
    ## (L-5.1) Psi Inverted
    Psinv <- try(chol2inv(cholPsi), TRUE)
    psisum <- sum(Psinv)
    ## (L-5.2) Mean
    mu <- sum(Psinv %*% Ay) / psisum
    ## (L-5.3) yoneMu, SigmSqr
    yonemu <- Ay - mu
    SigmaSqr <- (t(yonemu) %*% Psinv %*% yonemu) / n
    ## (L-5.4) Log Likelihood
    NegLnLike <- n * log(SigmaSqr) + LnDetPsi
    ## (L-5.5) Compile Result
    list(
      NegLnLike = NegLnLike,
      Psi = Psi,
      Psinv = Psinv,
      mu = mu,
      yonemu = yonemu,
      ssq = SigmaSqr
    )
  }
```

4.6.7 Predictions

The `buildKriging` function from the R package `spot` provides two Kriging predictors: prediction with and without re-interpolation. Re-interpolation is presented here, because it prevents an incorrect approximation of the error which might cause a poor global convergence. Re-interpolation bases the computation of the estimated error on an interpolation of points predicted by the regression model at the sample locations, see Forrester et al. (2008a).

The function `predictKrigingReinterpolation` requires two arguments: (i) `object`, the Kriging model (settings and parameters) of class `kriging`, and (ii) `newdata`, the design matrix to be predicted.

The function `normalizeMatrix2` is used to normalize the data. It uses information from the normalization performed during the Kriging model building phase, namely `normalizexmin` and `normalizexmax` to ensure the same scaling of the known and new data. Furthermore, the following optimized parameters from the Kriging model are extracted: `scaledx`, `dmodeltheta`, `dmodellambda`, `Psi`, `Psinv`, `mu`, and `yonemu`.

For re-interpolation, the error in the model excluding the error caused by noise is computed. The following modifications are made:

```
PsiB <-
  Psi - diag(lambda, n) + diag(.Machine$double.eps, n)
SigmaSqr <-
  as.numeric(t(yonemu) %*% Psinv %*% PsiB %*% Psinv %*% yonemu) /
    n
Psinv <- try(solve.default(PsiB), TRUE)
if (class(Psinv)[1] == "try-error") {
  Psinv <- ginv(PsiB)
}
```

The MLE for $\hat{y}$ is

$$\hat{y}(x) = \hat{\mu} + \psi^T \Psi^{-1}(y - 1\hat{\mu}). \tag{4.19}$$

This is Eq. (2.40) in Forrester et al. (2008a). It is implemented as follows:

```
psi <- matrix(0, k, n)
for (i in 1:nvar) {
  tmp <- expand.grid(AX[, i], x[, i])
  if (object$types[i] == "factor") {
    tmp <- as.numeric(tmp[, 1] != tmp[, 2])^p[i]
  } else {
    tmp <- abs(tmp[, 1] - tmp[, 2])^p[i]
  }
  psi <- psi + theta[i] * matrix(tmp, k, n, byrow = TRUE)
}

psi <- exp(-psi)


f <- mu + as.numeric(psi %*% (Psinv %*% yonemu))
```

Depending on the setting of the parameter `target`, the values `y` and `s` or `y`, `s`, and `ei` are returned.

```
res <- list(y = f)
if (any(object$target %in% c("s", "ei"))) {
  #
  Psinv <- try(solve.default(PsiB), TRUE)
  if (class(Psinv)[1] == "try-error") {
    Psinv <- ginv(PsiB)
  }
  #
  SSqr <-
    SigmaSqr * (1 - diag(psi %*% (Psinv %*% t(psi))))
  s <- sqrt(abs(SSqr))
  res$s <- s
  if (any(object$target == "ei")) {
    res$ei <- expectedImprovement(f, s, object$min)
  }
}
if (object$returnCrossCor) {
  res$psi <- psi
}
res
```

4.7 Program Code

One complete `spot` run is shown below. To increase readability, only one iteration of the `spotLoop` is performed.

Program Code: spot Run

```
## (S-1) Setup:
fun <- funNoise
lower <- c(-1, -1)
upper <- c(1, 1)
control <- list(
  OCBA = TRUE,
  OCBABudget = 3,
  replicates = 2,
  noise = TRUE,
  multiStart = 2,
  designControl = list(replicates = 2)
)
control <- spotFillControlList(control, lower, upper)

## (S-2) Initial design:
set.seed(control$seedSPOT)
```

```
x <- control$design(
  x = NULL,
  lower = lower,
  upper = upper,
  control = control$designControl
)
x <- repairNonNumeric(x, control$types)

## (S-3) Eval initial design
y <- objectiveFunctionEvaluation(
  x = NULL,
  xnew = x,
  fun = fun,
  control = control
)

## (S-4) Imputation
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}

## (S-5) Enter spotLoop:

## (S-6) Initial check:
initialInputCheck(x, fun, lower, upper, control, inSpotLoop = TRUE)
dimension <- length(lower)
con <- spotControl(dimension)
con[names(control)] <- control
control <- con
rm(con)
control <- spotFillControlList(control, lower, upper)

## (S-7) Imputation:
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}

## (S-8) Counter and logs:
count <- nrow(y)
modelFit <- NA
ybestVec <- rep(min(y[, 1]), count)
ySurr <- matrix(NA, nrow = 1, ncol = count)

## (S-9) Termination (while loop):

## (S-10) Subsect select:
```

```
selectRes <- control$subsetSelect(
  x = x,
  y = y[, 1, drop = FALSE],
  control = control$subsetControl
)

## (S-11) Surrogate fit:
modelFit <- control$model(
  x = selectRes$x,
  y = selectRes$y,
  control = control$modelControl
)

## (S-12) Surrogate optimization function:
funSurrogate <- evaluateModel(
  modelFit,
  control$infillCriterion,
  control$verbosity
)

## (S-13) Random starting points: surrogate optimization
x0 <- getMultiStartPoints(x, y, control)
resSurr <- matrix(NA, nrow = nrow(x0), ncol = ncol(x0) + 1)

## (S-14b) Surrogate optimization:
for (i in 1:nrow(x0)) {
  optimResSurr <- control$optimizer(
    x = x0[i, , drop = FALSE],
    funSurrogate,
    lower,
    upper,
    control$optimizerControl
  )
  resSurr[i, ] <- c(optimResSurr$xbest, optimResSurr$ybest)
}

## (S-15) Compile surrogate results:
m <- which.min(resSurr[, ncol(x) + 1])
## Determine xnew based on multi start results
xnew <- resSurr[m, 1:ncol(x), drop = FALSE]
## value on the surrogate (can be "y", "s2, "ei", "negLog10ei" etc.)
ySurrNew <- resSurr[m, ncol(x) + 1]


## (S-16) Duplicate handling:
xnew <- duplicateAndReplicateHandling(xnew, x, lower, upper, control)
# Repair non-numeric results
xnew <- repairNonNumeric(xnew, control$types)

## (S-17) OCBA
if (control$noise & control$OCBA) {
  xnew <- rbind(xnew, repeatsOCBA(
    x, y[, 1, drop = FALSE],
    control$OCBABudget
```

```
  ))
}

## (S-18) Evaluate xnew
ynew <- tryCatch(
  expr = {
    objectiveFunctionEvaluation(
      x = x,
      xnew = xnew,
      fun = fun,
      control = control
    )
  },
  error = function(e) {
    message("Error in objectiveFunctionEvaluation()!")
    print(e)
    if (!is.null(control$yImputation$handleNAsMethod)) {
      message("Error will be corrected.")
      n <- nrow(xnew)
      m <- ncol(y)
      return(matrix(rep(NA, m * n), nrow = n))
    }
  }
)

## (S-19) Impute
colnames(xnew) <- colnames(x)
x <- rbind(x, xnew)
y <- rbind(y, ynew)
if (!is.null(control$yImputation$handleNAsMethod)) {
  y <- imputeY(
    x = x,
    y = y,
    control = control
  )
}

## (S-20) Update counter, logs, etc.
ySurr <- c(ySurr, ySurrNew)
count <- count + nrow(ynew)
indexBest <- which.min(y[, 1, drop = FALSE])
ybestVec <- c(ybestVec, y[indexBest, 1, drop = FALSE])

## END while loop

## (S-21) Reporting after while loop in spotLoop
if (ncol(y) > 1) {
  logInfo <- y[, -1, drop = FALSE]
} else {
  logInfo <- NA
}
if (length(control$transformFun) > 0) {
  xt <- transformX(xNat = x, fn = control$transformFun)
} else {
```

```
  xt <- NA
}
## (S-22) OCBA-based selection of the best
if (control$noise & control$OCBA) {
  ocbaRes <- ocbaRanking(
    x = x,
    y = y,
    fun = fun,
    control = control
  )
  control$xBestOcba <- ocbaRes[1, 1:(ncol(ocbaRes) - 1)]
  control$yBestOcba <- ocbaRes[1, ncol(ocbaRes)]
}
# Compile results in spotLoop
result <- list(
  xbest = x[indexBest, , drop = FALSE],
  ybest = y[indexBest, 1, drop = FALSE],
  xBestOcba = matrix(control$xBestOcba, ncol = length(lower)),
  yBestOcba = matrix(control$yBestOcba, ncol = length(lower)),
  x = x,
  xt = xt,
  y = y[, 1, drop = FALSE],
  logInfo = logInfo,
  count = count,
  msg = "budget exhausted",
  modelFit = modelFit,
  ybestVec = ybestVec,
  ySurr = ySurr
)
## END spotLoop()

if (control$directOptControl$funEvals > 0) {
  ## (S-23) Starting point for direct optimization
  xbest <- result$xbest
  if (!is.null(control$directOptControl$eval_g_ineq) &&
    (
      control$directOptControl$opts$algorithm == "NLOPT_GN_ISRES" &
        control$directOptControl$eval_g_ineq(xbest) < 0
    )) {
    x0 <- NULL
  } else {
    x0 <- xbest
  }

  # Direct optimization on the real fun
  optimResDirect <- control$directOpt(
    x = x0,
    fun = fun,
    lower = lower,
    upper = upper,
    control$directOptControl
  )

  ## (S-24) Update results adding direct
```

```
  if (result$ybest > optimResDirect$ybest) {
    result$xbest <- optimResDirect$xbest
    result$ybest <- optimResDirect$ybest
  }
  result$x <- rbind(result$x, optimResDirect$x)
  result$y <- rbind(result$y, optimResDirect$y)
}
```

The result from one `spotLoop` is saved in the variable `result`.