CIplus, Band 1/2012

Thomas Bartz-Beielstein, Wolfgang Konen, Horst Stenzel, Boris Naujoks

A Gentle Introduction to Sequential Parameter Optimization

Thomas Bartz-Beielstein, Martin Zaefferer

Fachhochschule Köln
Cologne University of Applied Sciences

A Gentle Introduction to Sequential Parameter Optimization

Thomas Bartz-Beielstein and Martin Zaefferer
Department of Computer Science,
Cologne University of Applied Sciences, Germany.
Schriftenreihe CIplus
TR 1/2012. ISSN 2194-2870

Abstract

There is a strong need for sound statistical analysis of simulation and optimization algorithms. Based on this analysis, improved parameter settings can be determined. This will be referred to as *tuning*. Model-based investigations are common approaches in simulation and optimization. The *sequential parameter optimization toolbox* SPOT package for `R` [5] is a toolbox for tuning and understanding simulation and optimization algorithms. The toolbox includes methods for tuning based on classical regression and analysis of variance techniques; tree-based models such as classification and regressions trees (CART) and random forest; Gaussian process models (Kriging), and combinations of different meta-modeling approaches. This article exemplifies how an existing optimization algorithm, namely simulated annealing, can be tuned using the SPOT framework.

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* in the remainder of this article. This article illustrates how an existing search heuristic can be tuned using the SPOT framework.

In order to keep the setup as simple as possible, we will use the simulated annealing implementation, which is freely available in the `R` system [5]. This implementation of the simulated annealing heuristic will be referred to as SANN in the following.

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.

This paper is structured as follows. First, the optimization framework is described in Sec. 2. This framework consists of three levels, which are useful for distinguishing several problem domains. Section 3 provides some background information about SANN. A simple example of tuning SANN is described in Sec. 4. This tuning example can be used as a starting point for beginners. SPOT's file mode is discussed in Sec. 5. Section 6 presents a step-by-step walk through the SPOT procedure. How SPOT can be applied to tune and analyze arbitrary algorithms is demonstrated in Sec. 7. In contrast to all those non-deterministic optimization problems, Sec. 8 explains what changes need to be made if a deterministic problem has to be solved with SPOT. Finally, a short summary is presented in Sec. 9.

2 Levels during the Tuning Procedure

When tuning an optimization algorithm, the following three levels can be used to describe the experimental setup (Fig. 1).

(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 ES. 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).

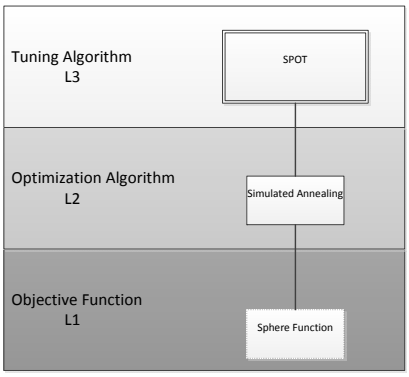


Figure 1: The three levels that occur while tuning an algorithm that optimizes a test function.

3 Simulated Annealing SANN

Simulated annealing is a generic probabilistic heuristic for global optimization [4]. 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 solutions. 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 temperature is gradually decreased during the optimization.

We consider the `R` implementation of SANN, which is available via the general-purpose optimization function `optim()` from the `stats` package in the `R` system. 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`. `method` denotes the *optimization method* to be used. We will use SANN in our examples. `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. `control` defines a relatively long *list of control parameters*. We will use the following parameters from this list: `maxit`, i.e., the *maximum number of iterations*, which is for SANN the maximum number of function evaluations. This is the stopping criterion. `temp` controls the SANN algorithm. It is the *starting temperature* for the cooling schedule with a default value of 10. 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.

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

```
> sphere <- function (x){
+     sum(x^2)
+     }
```

To obtain reproducible results, we will set the seed.

```
> set.seed(123)
```

Using a three-dimensional objective function and the starting point (initial values for the parameters to be optimized over) $(-1, 1, -1)$, we can execute the optimization runs as follows:

```
> res <- optim(c(-1,1,-1), sphere, method="SANN",
+              control=list(maxit=100, temp=10, tmax = 10))
> res


$par
[1] -0.3209798  0.6608565  0.4414664

$value
[1] 0.7346519

$counts
function gradient
     100       NA

$convergence
[1] 0

$message
NULL
```

The best, i.e., smallest, function value, which was found by SANN, reads 0.7346519. The corresponding point in the search space is approximately (-0.3209798, 0.6608565, 0.4414664). 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 $t_{\max}$. 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 $t_{\max}$=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 $t_{\max}$= 5. 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.

4 Tuning with SPOT: A First Example

4.1 The Objective Function (L1): Sphere

To keep the situation as simple as possible, we will use the sphere test function again:

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

SANN will be used to determine its minimum function value. As above, the sphere function is defined as

```
> sphere <- function(x){
+   sum(x^2)
+ }
```

4.2 The Optimization Algorithm (L2): SANN

Again, several settings have to be specified for the algorithm to be tuned. The problem design requires the specification of the starting point for the search. Note, its length defines the problem dimension.

```
> x0 <- c(-1,1,-1) #starting point that SANN uses when optimizing sphere()
```

Since `x0` has three elements, we are facing a three dimensional optimization problem. The budget, i.e., the maximum number of function evaluations that can be used by SANN is specified via

```
> maxit <- 100
```

As above, the SANN implementation from the `R` system will be used via the `optim()` function. We will consider two parameters: the initial temperature (`temp`) and the number of function evaluations at each temperature ($t_{\max}$). Both

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. If no seed is specified, `1234` *is used as the default seed value.* SPOT *uses two random seeds: the first is used by* SPOT *itself, e.g., for generating randomized designs. The second is used by the algorithm*

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	`1234`
Budget	maxit = 100	`maxit`

are integer values. To interface with SPOT, a wrapper function `spot2Sann()` can be defined as follows.

```
> spot2Sann <- function(pars,x0,fn,maxit){
+         temp<-pars[1]
+         tmax<-pars[2]
+         y <- optim(x0, fn, method="SANN", control=list(maxit=maxit, temp=temp, tmax=tmax))
+         return(y$value)
+ }
```

Note, as explained in Sect. 3,

```
optim(x0, fn, method="SANN", control=list(maxit=maxit, temp=temp, tmax=tmax))
```

is the standard way for starting SANN as an `R` optimizer. All parameters and settings of SANN as used for this simple example are summarized in Table 1.

4.3 The Tuning Procedure (L3): SPOT

The SPOT package can be installed from within `R` using the

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

command. Alternatively, SPOT can downloaded from the comprehensive `R` archive network at `http://CRAN.R-project.org/package=SPOT`. The latter procedure is recommended for the experienced `R` user only. SPOT is one possible implementation of the *sequential parameter optimization* (SPO) framework introduced in [1]. For a detailed documentation of the functions from the SPOT package, the reader is referred to the package help manuals.

SPOT has to be loaded to the workspace and a *region of interest* (ROI) has to be defined. The ROI specifies SPOT's search intervals for the SANN parameters, i.e., for $t_{\max}$ and `temp`.

```
> require(SPOT)
> roi<-spotROI(c(1,1),c(100,100),type=c("INT","INT"))
```

Here, both parameters `temp` and $t_{\max}$ will be tuned in the region between one and 100.

Finally, before calling SPOT, the tuning procedure has to be configured. This is done by setting up a config list. Here, we specify information about the initial design size (`init.design.size=4`) and the number of meta models which are build by SPOT (`auto.loop.steps=5`). More configurations can be chosen, but remain at default values for this simple example.

```
> config<-list(alg.func=spot2Sann,
+                        alg.roi=roi,
+                        init.design.size=4,
+                        auto.loop.steps=5)
```

Now we are ready to start SPOT via `spot()`.

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit)


spot.R::spot started
```

4.4 Results

Output from the SPOT run is stored in the `res` variable which is a list. Results from each SANN evaluation are stored in the `res$alg.currentResult` element.

```
> str(res$alg.currentResult)


'data.frame':        53 obs. of  9 variables:
 $ Function: Factor w/ 1 level "UserSuppliedFunction": 1 1 1 1 1 1 1 1 1 1 ...
 $ XDIM    : num  2 2 2 2 2 2 2 2 2 2 ...
 $ YDIM    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ STEP    : num  0 0 0 0 0 0 0 0 1 1 ...
 $ SEED    : num  1234 1235 1234 1235 1234 ...
 $ CONFIG  : int  1 1 2 2 3 3 4 4 5 5 ...
 $ VARX1   : num  100 100 54 54 2 2 40 40 11 11 ...
 $ VARX2   : num  43 43 99 99 57 57 7 7 57 57 ...
 $ Y       : num  3 2.964 3 2.964 0.176 ...
```

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

```
> best <- res$alg.currentBest[nrow(res$alg.currentBest),]
> print(best)


           Y VARX1 VARX2 COUNT CONFIG STEP
32 0.1578842     2    57     4      3    6
```

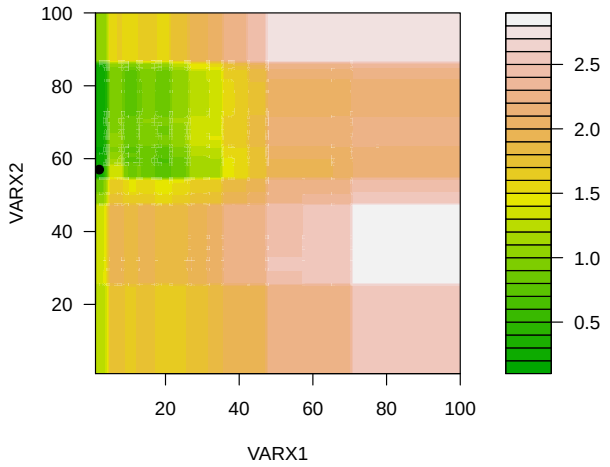


Figure 2: Contour plot of the last meta model (Random Forest) used by the SPOT run when tuning SANN. VARX1 is `temp`*, VARX2 is* `tmax`*.*

Results can also be used to illustrate the algorithm's performance. Several pre-defined report functions come with SPOT, e.g., the `spotReportContour`, which generates the contour plot from Fig. 2.

```
> spot(spotConfig=append(list(
+  report.func="spotReportContour",
+  report.interactive=F),
+  res),
+  spotTask="rep")
```

5 SPOT File Mode

SPOT can be run in two different modes[1]:

- File mode enabled: Information are stored in several files, which allow an extensive analysis. Preferred mode, if SPOT is used interactively or if the run is expected to take rather long.
- File mode disabled: Should be used to improve performance. Basically, a good choice when final results are important.

The file mode only concerns the SPOT output. These files will be referred to as output files. Regardless of the chosen file mode, the user can select either to provide the SPOT settings and parameters as R variables in the workspace, or to provide everything in files. These files will be referred to as input files.

In this section it is explained how to use SPOT using input and output files. For this purpose, the same setup as in the previous section will be used.

[1] This section contains advanced material and can be skipped during the first reading.

5.1 Enabling and Disabling SPOT's File Mode

SPOT can be run without generating any output to the file system. The variable `spot.fileMode` can be used to change SPOT's basic I/O behavior. The user can change its value in SPOT's configuration list as follows.

```
> config<-list(alg.func=spot2Sann, alg.roi=roi, init.design.size=4, auto.loop.nevals=20,
+         spot.fileMode = FALSE)
```

Using this setting, SPOT will only use the workspace to store variables.

5.2 File structure used in SPOT

SPOT files use a naming convention, which allows the definition of *projects*. Each configuration file belongs to one SPOT project, if the same basename is used for corresponding files. SPOT uses simple text files as interfaces from the algorithm to the statistical tools.

1. The user can provide the following input files:
   (i) *Region of interest* (ROI) files specify the region over which the algorithm parameters are tuned. Categorical variables such as the recombination operator in ES, can be encoded as factors, e.g., "intermediate recombination" and "discrete recombination."
   (ii) *Algorithm design* (APD) files are used to specify parameters used by the algorithm, e.g., problem dimension, objective function, starting point, or initial seed. APD files are not mandatory if custom target functions are used.
   (iii) *Configuration* files (CONF) specify SPOT specific parameters, such as the prediction model or the initial design size. If the file mode is disabled, this information is stored directly in the `config` variable.
2. SPOT will generate the following output files (if `spot.fileMode` is set to TRUE):
   (i) *Design* files (DES) specify algorithm designs. They are generated automatically by SPOT and will be read by the optimization algorithms.
   (ii) After the algorithm has been started with a parametrization from the algorithm design, the algorithm writes its results to the *result file* (RES). Result files provide the basis for many statistical evaluations/visualizations. They are read by SPOT to generate prediction models. Additional prediction models can easily be integrated into SPOT.
   (iii) In each sequential step, the best result will be stored in a *best file* (BST). This is in essence a product of the RES file, providing easy access to progress information.

5.3 SPOT Configuration

A *configuration* (CONF) file stores information about SPOT specific settings. These are the same settings that can be provided to spot in a list variable, as described in the second part of section 4.3. All settings not specified in the file (or list) will be set to default values.

For example, the number of SANN algorithm runs, i.e., the available budget, can be specified via `auto.loop.nevals`. SPOT implements a sequential approach, i.e., the available budget is not used in one step. Evaluations of the algorithm on a subset of this budget, the so-called initial design, is used to generate a coarse grained meta model F. This meta model is used to determine promising algorithm design points which will be evaluated next. Results from these additional SANN runs are used to refine the meta model F. The size of the initial design can be specified via `init.design.size`. To generate the meta model, we use random forest [3]. This can be specified via `seq.predictionModel.func = "spotPredictRandomForest"`.

Random forest was chosen, because it is a robust method which can handle categorical and numerical variables.

5.3.1 Setup of the CONF for the SANN

The corresponding CONF file for the SANN contains the following lines.

File: sannExample.conf:

```
alg.func =  function(pars){
temp<-pars[1]
tmax<-pars[2]
y <- optim(x0, fn, method="SANN",
control=list(maxit=maxit, temp=temp, tmax=tmax))
return(y$value)
}
spot.fileMode=TRUE
auto.loop.steps = 5
init.design.size = 4
```

As can be seen from this example, the setup of the CONF file is very simple. Most basically, it requires the specification of the function call to the algorithm, which should be tuned. Additionally, the file mode, the number of sequential steps and the initial design size are specified.

One difference between the wrapper used here, and in the previous example is that settings like the budget and the target function of SANN are not directly passed to the function, but are handled in the global environment of R. This way, they can be defined in an APD file (see Sec. 5.5).

5.3.2 An Extended CONF File

The setup used in this introductory example is based on a very simply configuration. More options can be specified. Simply execute help with `spotGetOptions` as an argument, i.e.,

```
> help(spotGetOptions)
```

For example, a more extensive CONF file might looks as follows.

```
alg.func =  function(pars){
temp<-pars[1]
tmax<-pars[2]
y <- optim(x0, fn, method="SANN",
control=list(maxit=maxit, temp=temp, tmax=tmax))
return(y$value)
}
alg.seed = 1235
auto.loop.steps = Inf;
auto.loop.nevals = 100;
init.design.func = "spotCreateDesignLhs";
init.design.size = 10;
init.design.repeats  = 2;
io.columnSep = " ";
seq.design.maxRepeats = 10;
seq.design.size = 100
seq.predictionModel.func = "spotPredictRandomForest"
spot.fileMode=TRUE
spot.seed = 125
report.func = "spotReportSens"
report.io.pdf = TRUE
```

The settings can be explained as shown in Table 2. There are more settings available, and explained in the help document mentioned above. They will be used with default values if not specified.

5.4 The Region of Interest

A *region of interest* (ROI) file specifies algorithm parameters and associated lower and upper bounds for the algorithm parameters.

- Values for `temp` are chosen from the interval $[1; 100]$.
- Values for $t_{\max}$are from the interval $[1; 100]$.

The corresponding ROI file looks as follows.

File: sannExample.roi:

```
name low high type
TEMP 1 100 INT
TMAX 1 100 INT
```

Table 2: SPOT *configuration*

alg.func	Specify the name of the algorithm to be tuned	STRING
alg.seed	Seed passed to the algorithm	INT
auto.loop.steps	SPOT Termination criterion. Number of meta models to be build by SPOT	INT
init.design.func	Name of the function to create an initial design	STRING
init.design.size	Number of initial design points to be created	INT
init.design.repeats	Number of repeats for each design point from the initial design	INT
seq.design.maxRepeats	Maximum number of repeats for design points	INT
seq.design.size	Number of design points evaluated by the meta model	INT
seq.predictionModel.func	Meta model	STRING
spot.seed	Seed used by SPOT, e.g., for generating LHD	INT
report.func	Name of the report function	STRING
report.io.pdf	Write report to pdf	BOOLEAN

5.5 The Algorithm and Problem Design File

Parameters related to the algorithm or the optimization problem are stored in the APD file. This file contains information about the problem and might be used by the algorithm. For example, the starting point `x0 = (-1,1,-1)` can be specified in the APD file.

File: sannExample.apd:

```
assign("x1", c(-1,1,-1), envir = .GlobalEnv)
assign("maxit", 100, envir = .GlobalEnv)
assign("fn",  function(x){sum(x^2)}, envir = .GlobalEnv)
```

The variables have to be assigned to the global environment, to make sure that they are available for the wrapper function defined in the CONF file. It is not mandatory to use an APD file (for an alternative, see the simple example in the previous Sec. 4). However, it might help to clearly specify the problem settings. The APD file provides a nice way to look up settings of past experiments.

5.6 Running SPOT—Automatic Tuning of SANN

Now that the interface to SANN has been defined, and the experimental setup has been specified, the SPOT run can be performed. Consider the following situation: The user has created a working directory for running the experiments, say `myspot`, which is a subfolder of his `Documents` folder. This directory contains the following files.

- SPOT configuration (`sannExample.conf`),
- Region of interest (`sannExample.roi`),
- Algorithm and problem parameters (`sannExample.apd`)

`R` should be started in the `myspot` directory. You can use `R`'s menu to *change the working directory*. Alternatively, you can set the working directory using the `R` command:

```
> setwd("c:/users/yourname/documents/myspot")
```

Sometimes is it required to start a clean `R` session, because data from previous runs are in the workspace. Execute

```
> rm(list=ls());
```

to perform a cleanup before SPOT is loaded and run. We are using the configuration file `sannExample.conf` to start SPOT's automatic tuning procedure.

```
> library(SPOT)
> res<-spot("sannExample.conf")


spot.R::spot started


> best <- res$alg.currentBest[nrow(res$alg.currentBest),]
> print(best)


           Y VARX1 VARX2 COUNT CONFIG STEP
32 0.1578842     2    57     4      3    6
```

As can be seen, the result is the same as in the simple example without files.

Since `spot.fileMode=TRUE` is set, a result file (RES), which contains important information from the tuning process, has been written to the working directory, as well as a BST and DES file. They have the same name as the configuration file but different file extensions, according to their type.

- Results of each evaluation of the algorithm (SANN) (`sannExample.res`),
- Best values from each sequential step of SPOT (`sannExample.bst`),
- Last design as planned by SPOT (`sannExample.des`)

6 A Step-by-step Walk through

Previous examples showed how SPOT can be used for automated tuning. Additionally, it can also be used in a step-by-step manner. This section demonstrates the different options in detail.

6.1 Sequential Steps

SPOT sequentially performs the following steps during a tuning process.

1. Init: The initial design is generated. If spot.fileMode is enabled, the resulting design is written to the design file.

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit,spotTask="init")
```

2. Run: Using the parameters from the design file, the algorithm is run. If spot.fileMode is enabled, the resulting design is written to the result file.

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit,spotTask="run")
```

3. Seq: Using data from the result file, a meta model is generated. This model is used to determine additional design points, which are written to the design file, if spot.fileMode is enabled.

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit,spotTask="seq")
```

4. Report. A report is generated.

```
> spot(spotConfig=append(list(report.func="spotReportContour",report.interactive=F),res),
+      spotTask="rep")
```

In an automated SPOT run (i.e. spotTask="auto"), step two and three are repeated as long as no stopping criterion is fulfilled. Only when a stopping criterion is fulfilled, the fourth step is executed. In the step-by-step procedure, the user can choose to create reports at any point, and continue with more tuning steps afterwards.

6.2 Running SPOT automatically

SPOT can execute the tasks init → run → seq → run → seq → ... until the budget, which is specified via `auto.loop.nevals`, is exhausted. This automatic mode is SPOT's default mode. It can be explicitly set via

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit,spotTask="auto")
```

The call without any argument to the `spotTask` variable

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit)
```

leads to exactly the same result.

6.3 The Simple Example Revisited—Step-by-step

We will use the example from Sec. 4 to illustrate the four steps (init, run, seq, and rep), which were used by SPOT.

```
> sphere <- function(x){
+   sum(x^2)
+ }
> x0 <- c(1,-1,1) #starting point that \SANN uses when optimizing sphere()
> maxit <- 100 #number of evaluations of sphere() allowed for SANN
> spot2Sann <- function(pars,x0,fn,maxit){
+         temp<-pars[1]
+         tmax<-pars[2]
+         y <- optim(x0, fn, method="SANN",
+                         control=list(maxit=maxit,
+                         temp=temp, tmax=tmax))
+         return(y$value)
+ }
> require(SPOT)
> roi<-spotROI(c(1,1),c(100,100),type=c("INT","INT"))
> config<-list(alg.func=spot2Sann,
+         alg.roi=roi,
+         init.design.size=10,
+         auto.loop.nevals=20,
+         spot.fileMode=T
+ )
```

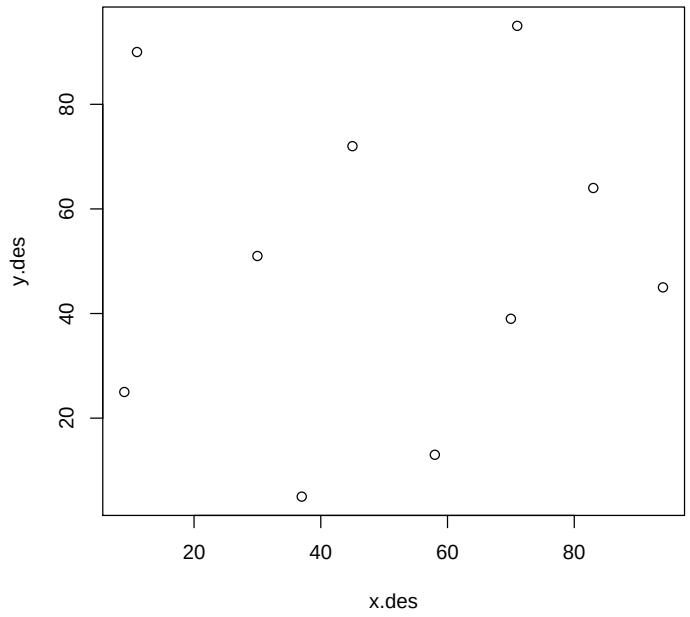
Now, we will execute the first step (init) only.

```
> res<-spot(spotConfig=config,x0=x0,fn=sphere,maxit=maxit,spotTask="init")


spot.R::spot started
```

The initial design can be visualized as follows.

```
> x.des<-res$alg.currentDesign$VARX1
> y.des<-res$alg.currentDesign$VARX2
> plot(y.des ~ x.des, type="p")
```



Based on these design points, SPOT will run the algorithm. The results of the algorithms runs can then be accessed.

```
> res<-spot(spotConfig=res,x0=x0,fn=sphere,maxit=maxit,spotTask="run")
```

```
spot.R::spot started
```

```
> df.res <-res$alg.currentResult
> print(df.res)
```

```
               Function XDIM YDIM STEP SEED CONFIG VARX1 VARX2         Y
1  UserSuppliedFunction    2    1    0 1234      1    71    95 3.0000000
2  UserSuppliedFunction    2    1    0 1235      1    71    95 0.4653657
3  UserSuppliedFunction    2    1    0 1234      2    11    90 0.3945913
4  UserSuppliedFunction    2    1    0 1235      2    11    90 0.2432183
5  UserSuppliedFunction    2    1    0 1234      3    70    39 1.5021431
6  UserSuppliedFunction    2    1    0 1235      3    70    39 0.4653657
7  UserSuppliedFunction    2    1    0 1234      4    58    13 3.0000000
8  UserSuppliedFunction    2    1    0 1235      4    58    13 0.4653657
9  UserSuppliedFunction    2    1    0 1234      5    94    45 3.0000000
10 UserSuppliedFunction    2    1    0 1235      5    94    45 0.4653657
11 UserSuppliedFunction    2    1    0 1234      6     9    25 0.4996259
12 UserSuppliedFunction    2    1    0 1235      6     9    25 1.8288022
13 UserSuppliedFunction    2    1    0 1234      7    30    51 0.6428562
14 UserSuppliedFunction    2    1    0 1235      7    30    51 0.4653657
15 UserSuppliedFunction    2    1    0 1234      8    83    64 1.4493019
```

```
16 UserSuppliedFunction    2    1    0 1235      8    83    64 0.4653657
17 UserSuppliedFunction    2    1    0 1234      9    37     5 3.0000000
18 UserSuppliedFunction    2    1    0 1235      9    37     5 1.4660851
19 UserSuppliedFunction    2    1    0 1234     10    45    72 3.0000000
20 UserSuppliedFunction    2    1    0 1235     10    45    72 0.4653657
```

We can use results from the first run to generate a report.

```
> spot(spotConfig=append(list(
+  report.func="spotReportContour",
+  report.interactive=F),
+  res),
+  spotTask="rep")
```

Or else we can use more steps, to continue tuning the algorithm

```
> res<-spot(spotConfig=res,x0=x0,fn=sphere,maxit=maxit,spotTask="seq")
> res<-spot(spotConfig=res,x0=x0,fn=sphere,maxit=maxit,spotTask="run")
> res<-spot(spotConfig=res,x0=x0,fn=sphere,maxit=maxit,spotTask="seq")
> res<-spot(spotConfig=res,x0=x0,fn=sphere,maxit=maxit,spotTask="run")
```

If this is done, the results are comparable to the automated tuning, unless the user modifies some variables. And this is where one of the main advantages of the step-by-step procedure comes in. The user can decide to alter the next design, instead of simply using what the "seq" step suggests. The user can try to remove outliers which are visible to him from the results of the "run" step. Or he might decide to change SPOT settings like the used meta model during the tuning, for instance switching from a simple linear model to a more advanced Kriging model. It might even be reasonable to change the region of interest, if the user observes that another region might be more interesting for the tuning procedure.

In contrast to the automated tuning, the step-by-step approach allows for user knowledge to be taken into consideration. Of course this needs more expertise of the user, but can potentially improve results significantly.

7 Interfacing SPOT: An Extended Example

The earlier sections described how to use SPOT in different ways to tune functions or algorithms in R. However, often algorithms are already implemented in other programing languages.

Therefore, this section illustrates how SPOT can be used for tuning and analyzing an arbitrary algorithm, which is executed with a call to the system command line. As an example, a simple evolution strategy implemented in JAVA will be used, namely the (1+1)-ES.

The (1+1)-ES creates one offspring by mutation in each generation. If the offspring is better than the parent, it will become the next generations parent. This strategy has an internal step size used for mutation. The step size is multiplied with the *step size multiplier* if the success rate is higher than $\frac{1}{5}$, and divided by the *step size multiplier* if it is smaller. If it is exactly $\frac{1}{5}$, it remains unchanged.

For this strategy, three parameters can be varied:

1. initial step size - Initial value for the step size.
2. step size multiplier - Parameter for the step size adaption.
3. history length - The number of past steps considered for calculation of the success rate.

7.1 Problem Definition

As earlier described for SANN, the first step is to define the problem to be solved. Here, the problem is to tune the (1+1)-ES on the two dimensional sphere function (called "Ball" in the JAVA code). The starting point is chosen as (-2,3), and the (1+1)-ES has a budget of 100 evaluations on the sphere function to find the optimum.

```
> #choose target function which is defined in the java code
> fn<-"de.fhkoeln.spot.objectivefunctions.Ball"
> #dimension of target function
> n=2
> #starting point that the (1+1)-ES uses when optimizing the target functino
> xp0 <- "[-2.0,3.0]"
> px <- 0 #individual printing mode
> py <- 1 #objective function value printing mode
> #stopping criterion: budget of (1+1)-ES
> steps <- 100
> #stopping criterion: minimum value
> target <- 1e-20
> #seed to be passed to the algorithm
> alg.seed <<- 1
```

7.2 Interfacing with SPOT

The second step is now to define a wrapper function to be called by SPOT, which will in turn call the JAVA code. The .jar file of the JAVA code used here has to be in the current working directory.

```
> spot2opoes <- function(pars,f,n,xp0,px,py,steps,target){
+         sigma0<-pars[1] #initial step size
+         a<-pars[2] #step size muliplier
+         g<-round(pars[3]) #history length, has to be integer!
+         alg.seed<<-alg.seed+1
+         #Note: the used jar file has to be in the current working directory (see also: setwd(), getwd())
+         callString <- paste("java -jar simpleOnePlusOneES.jar"
+                 , alg.seed, steps, target, f, n, xp0, sigma0, a, g, px, py, sep = " ")
+         print(callString)
+         y <-system(callString, intern= TRUE) #evaluate callstring
+         return(as.numeric(as.character(y))) #return as numeric
+ }
```

7.3 Configuring and Running SPOT

The parameters have to be assigned to a region of interest, in which they are tuned. Here, the first two parameters are floats (initial step size and step size multiplier), the third is an integer (i.e. history length).

```
> require(SPOT)
> roi<-spotROI(c(0.1,1,2),c(5,2,100),type=c("FLOAT","FLOAT","INT"))
```

Finally, SPOT can be configured and started. A Kriging model is selected for the sequential model optimization. With this setting, new design points will be found by applying Covariance Matrix Adaption Evolution Strategy ("cmaes") to the build model.

```
> config<-list(alg.func=spot2opoes,
+                         alg.roi=roi,
+                         seq.predictionModel.func="spotPredictForrester",
+                         seq.predictionOpt.func="spotPredictOptMulti",
+                         seq.predictionOpt.method="cmaes",
+                         seq.predictionOpt.budget=1000,
+                         report.func="spotReportSens",
+                         spot.fileMode=F,
+                         io.verbosity=3,
+                         auto.loop.nevals=100)
> spotResults<-spot(spotConfig=config,f=fn,n=n,xp0=xp0,px=px,py=py,steps=steps,target=target)
```

The results of this tuning run are stored in the *spotResults* variable, and can be processed for further analysis. Also, the "spotReportSens" Report function will present a small summary of influence and performance of the different tuned parameters.

8 Deterministic Problems

Previous sections discussed the tuning of non-deterministic, i.e., noisy, algorithms, which is the usual case when tuning evolutionary algorithms. After going through these examples in the previous sections, this section presents an application of SPOT in a simple setting: We will describe how SPOT can be used for tuning deterministic algorithms. 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 sphere function is defined as usual:

```
> sphere <- function(x){
+   sum(x^2)
+ }
```

SPOT requires an ROI. The interval $[-5; 5] \times [-5; 5]$ was chosen as the region of interest, i.e., we are considering a two-dimensional optimization problem.

```
> roi<-spotROI(c(-5,-5),c(5,5))
```

We can start with a configuration list that looks very much like earlier examples

```
> config<-list(alg.func=sphere,
+                         alg.roi=roi,
+                         seq.predictionModel.func="spotPredictForrester",
+                         seq.predictionOpt.method="L-BFGS-B",
+                         spot.fileMode=F,
+                         io.verbosity=0,
+                         auto.loop.nevals=25)
```

However, there are certain changes to be made to configuration for deterministic problems.

1. Optimal Computation Budget Allocation is neither necessary nor possible, so it has to be deactivated. Hence, we set `spot.ocba` to FALSE.

2. The initial and sequential design points do not need to be evaluated repeatedly. Therefore, we set `init.design.repeats` and `seq.design.maxRepeats` to one.

```
> config$spot.ocba=FALSE
> config$init.design.repeats=1
> config$seq.design.maxRepeats=1
```

With these changes, SPOT can be started and the generated results can viewed.

```
> res<-spot(spotConfig=config)
```

```
spot.R::spot started
```

```
> print(res$alg.currentResult[,4:9])
```

```
   STEP SEED CONFIG       VARX1       VARX2           Y
1     0 1234      1  2.01353063  4.45208558 23.87537164
2     0 1234      2 -3.98847824  3.95660866 31.56271072
3     0 1234      3  1.99644571 -1.15556965  5.32113668
4     0 1234      4  0.76568967 -3.77723140 14.85375770
5     0 1234      5  4.31466261 -0.57982982 18.95251607
6     0 1234      6 -4.18798826 -2.58001706 24.19573368
7     0 1234      7 -2.00878078  0.02079085  4.03563249
8     0 1234      8  3.20075959  1.38044797 12.15049857
9     0 1234      9 -1.35396278 -4.57765157 22.78810913
10    0 1234     10 -0.56244927  2.19142606  5.11869737
11    1 1234     11 -0.31929738  0.56017576  0.41574770
12    1 1234     12  0.49380446 -0.72776007  0.77347757
13    1 1234     13  0.43639658  0.86660513  0.94144642
14    2 1234     14 -0.42955831  0.06721684  0.18903845
15    2 1234     15  0.15468483 -0.48861284  0.26266991
16    2 1234     16  0.52580978  0.15022249  0.29904272
17    3 1234     17 -0.15182185 -0.09236320  0.03158083
18    3 1234     18 -0.31333973  0.14101794  0.11806785
19    3 1234     19  0.31928189 -0.19640035  0.14051402
20    4 1234     20  0.02987370 -0.12132428  0.01561202
21    4 1234     21 -0.09071041 -0.33762395  0.12221831
22    4 1234     22 -0.28203678 -0.25850504  0.14636960
23    5 1234     23  0.58639646 -0.55008131  0.64645026
24    5 1234     24  0.81520531  0.40903780  0.83187162
25    5 1234     25  0.90825434 -0.36748141  0.95996853
```

As can be seen, each configuration tested during the SPOT run was evaluated only once. We can extract the best solution with the following command.

```
> best <- res$alg.currentBest[nrow(res$alg.currentBest),]
> print(best)

             Y     VARX1      VARX2 COUNT CONFIG STEP
201 0.01561202 0.0298737 -0.1213243     1     20    6
```

Note, if you are tuning (optimizing) deterministic algorithms or functions, OCBA has to be disabled. Otherwise, SPOT will exit with a warning message.

9 Summary

This article describes how SPOT can be used for tuning a search heuristic, i.e., SANN. We can distinguish three levels, which are involved in this tuning process. On level (L1), we can specify the real-world system, which includes the objective function. The second level (L2) contains the optimization algorithm, which can be a stochastic algorithm such as SANN. Finally, the third level (L3) describes the tuning algorithm, e.g., SPOT.

SANN's basic functionality is introduced and its parametrization described. Two parameters, namely $t_{\max}$and `temp`, are crucial for SANN's performance. Instead of manually tuning these two parameters, we use the SPOT to find good parameter values systematically. Therefore, a simple objective function, i.e., the sphere function, was used. This results in the following setting: SANN was used to optimize the sphere function, and SPOT was used to optimize SANN. To differentiate these two optimizations, the latter was referred to as *tuning*.

We also discussed SPOT's file mode and presented a step-by-step walk through SPOT's tuning process. The SPOT process is based on the following steps: Initialization (spotTask=init), run (spotTask=run), sequential (spotTask=seq), and report (spotTask=rep). If SPOT is run automatically, the following sequences is executed: init, run, seq, run, ..., and finally rep.

How to setup an interface to optimization algorithms, which were programmed in other programming languages was discussed. A wrapper function, which executes a JAVA program, was introduced. Finally, we discussed the SPOT configuration for deterministic algorithms.

The book [2] might be a good starting point for further studies. New related to SPOT are published on `www.spotseven.de`.

References

[1] Thomas Bartz-Beielstein. *Experimental Research in Evolutionary Computation—The New Experimentalism*. Natural Computing Series. Springer, Berlin, Heidelberg, New York, 2006.

[2] Thomas Bartz-Beielstein, Marco Chiarandini, Luis Paquete, and Mike Preuss, editors. *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, Berlin, Heidelberg, New York, 2010.

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

[4] S. Kirpatrick et al. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[5] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.