Flexible Global Optimization with Simulated-Annealing

by Kai Husmann, Alexander Lange and Elmar Spiegel

Abstract An abstract of less than 150 words.

Introduction

As early computer-based optimization methods developed contemporaneously with the first digital computers (Corana et al., 1987), nowadays numerous optimization methods for various purposes are available (Wegener, 2005). One of the main challenges in Operations Research is therefore to match the optimization problem with a reasonable method. The complexity of the problem thereby qualifies the possible methods. Optimization procedures in general can be distinguished into exact methods and heuristics (Kirkpatrick et al., 1983). For simple optimization problems, exact methods are often meaningful tools of choice. If all assumptions on model loss and restrictions are met, these methods will obligatorily find the exact solution without need for further parameters. They are the easiest way of solving optimization problems. The *Linear Simplex-Method* (Dantzig et al., 1959) as an example which only needs the loss-function and optional restrictions as model input. If, however, any of the model assumptions, e.g. linearity or unimodality, is violated, exact methods are unable to solve problems validly. With developing computer power, heuristics like the Savings-Algorithm (Clarke and Wright, 1964) and metaheuristics like *Simulated Annealing* (SA) (Kirkpatrick et al., 1983) became popular. They enable solving more complex optimization problems. Metaheuristics are a generalization of heuristics with aim to be even more flexible and efficient (Blum and Roli, 2003) such that they can solve complex optimization problems, like nonlinear problems. Depending on the method of choice, more or less assumptions on the loss function can be neglected on the one hand. On the other hand, heuristics and metaheuristics will always solve problems approximately. Precision of the solution depends on the optimization method and further parameters. There is even no guaranty of approximating the actual optimum since the solution also depends, by contrast to exact methods, on parameterization (Blum and Roli, 2003). Definition of proper parameters is thus a crucial point of those methods. The complexity of parameterization will by trend increase with flexibility of the method while efficiency trends to decrease. Direct search heuristics like the *Nelder-Mead* (NM) algorithm are comparatively efficient methods which directly converge to the functions optimum and need relatively less settings (Geiger and Kanzow, 1999). Random search methods are able to cope with multimodal objective functions. The efficiency and accuracy of such models is strongly sensitive to their parameter specification (Corana et al., 1987). Heuristics are often programmed for multi-purpose usage such that there is a suitable method for many optimization problems. For complex optimization problems, however, multi-purpose optimizers often fail to find solutions. Furthermore multi-purpose optimizers are usually not suitable or not efficient for highly complex problems like problems with restricted parameter space. Whenever general-purpose optimizers are too restrictive or inflexible to solve a problem properly, specific methods are advantageous. Those offer a lot of variable parameters such that they be parameterized very specific to the optimization problem. They represent the most flexible and the most complex optimization methods (Blum and Roli, 2003).

SA (Kirkpatrick et al., 1983) is known to be one of the oldest and most flexible metaheuristic method, though the term metaheuristic was established after initial publication of SA (Blum and Roli, 2003). It is known to be favorable against many other methods for multimodal loss functions with a very high number of covariates (Corana et al., 1987). The method was applied in many studies among several fields covering e.g. chemistry (Agostini et al., 2006), econometrics (Ingber, 1993) or forest sciences (Baskent and Jordan, 2002). Since its first implementation by Kirkpatrick et al. (1983), many authors have modified the algorithm in order to adopt it for specific problems (e.g. DeSarbo et al., 1989; Goffe et al., 1996) or make it more general (e.g. Xiang et al., 2013). It basically combines systematic and a stochastic components, thus enables escaping local optima. It is hence typically used for global optimization of multimodal functions. As it offers a lot of options, SA can be named as hybrid method between a general optimizer (when default values are chosen) and a problem specific optimization algorithm (Wegener, 2005). Corana et al. (1987) developed a dynamic adoption method for the variation of the stochastic component during the optimization process. Their modification affects the efficiency as well as the accuracy of the SA algorithm. It has potential to substantially improve the method. Pronzato et al. (1984) suggest to decrease the search-domain of the random component with increasing number of iterations. The stochastic component in general is the most sensitive part of the method since it actually determines the loss variables modification during the iterations.

The R software environment provides platform for simple and effective distribution of statistical models to a huge user community (Xiang et al., 2013). Thus, not surprisingly, several optimization packages of high quality can currently be purchased via *Comprehensive R Archive Network* (Theussl and Borchers, 2016) where even the SA method is recently listed 5 times. We, however, believe, there is need for a specific stochastic optimization package for complex optimization problems. A package coping with very flexible user definable loss functions with multiple options could be an advantageous extension for R. There is demand for specifically definable optimization problems. We therefore present the package **optimization** which is basically a modified version of SA. We used the properties of SA to program a random search optimization method for specific purposes. We therefore focused on flexibility and implemented many user specifiable parameters. Our method is, in its default configuration, usually not immediately efficient on the one but flexibly adoptable to specific purposes on the other hand. Main advantages of the package are the possibilities to specifically adjust covariate changing rules as well as the the robustness of the loss function. The changing rule allows the user e.g. to define an integer parameter space. The loss function can return any value, even Na or NaN are possible. Several further user specifications help to parameterize the model problem-specific such that the user can influence accuracy and speed very detailed. It is moreover the first R function where the improvements of Corana et al. (1987) and Pronzato et al. (1984) are implemented into an SA based optimization software. This means, the search domain of the stochastic component of SA dynamically shrinks with increasing iterations. We also implemented a generic plot function for post-hoc inspection of the model convergence assessment and the solution quality. In the following, we briefly introduce the algorithm methodologically and explain the most relevant parameters. We show in 4 examples where our model is favorable against existing methods and explain how it can be parameterized. We develop examples illustrating the basic model behavior with ficus on the covariate changing rule. We give hints how to adopt the numerous options to specific problems. Two practical examples where our function is recently used underpin the relevance of our specific-purpose optimization method.

The package optimization

In this section, theory of our SA interpretation and resulting parameters are explained.

Method

Since the basic idea of classic SA is derived from the physical process of metal annealing, the nomenclature of SA comes particularly from metallurgy. Just as the classic SA, our function is composed of an inner for and an outer while loop (Kirkpatrick et al., 1983). The number of iterations in both loops can be defined by the user. For better overview, we displayed the important steps of our function in a pseudocode (Algorithm 1).

Function of the *inner* for loop (Algorithm 1, lines 4 to 29) is basically to draw covariate combinations randomly and to compare the returns. The loop repeats n_{inner} times. First operation of the inner loop (line 5) is saving the covariate combinations of the last inner iteration as x_j. In the first iteration x_j is the vector with user defined initial covariates. Secondly (line 6), the covariates are changed. This changing process is an essential difference between our approach and any other SA based method in the R framework. Shape and behavior of the variation process may be defined by the user. Hints and examples for specifying this variation, which is a user defined R function, will also be given in the following sections. The variation function is used to create a temporary vector of covariates x_{i*}. Next to the former covariate combination x_j, the variation function is allowed to depend on a vector with random factors *rf* and the temperature *t*. Since *rf* and *t* change over time, the variation function thus can have dynamic components. Adjustment of *rf* and *t* is done in the outer loop which will be explained explicitly in the following paragraph. In the classical SA approach, the covariates x_{i*} are generated by adding or subtracting a uniform distributed random number to x_j (Kirkpatrick et al., 1983). The range of the uniform random number is, in our function, determined by *rf* whereat *rf* is relative to x_j. A *rf* of 0.1 and a covariate expression of 3 e.g. leads to a uniform random number between 2.7 and 3.3. This usual variation function is also default in our approach. A very simple exemplary modification of the variation function could e.g. be a normally distributed random number. Users could define a variation function that draws normally distributed random numbers around x_j with standard deviation *rf*. After generating x_{i*}, the boundaries are checked (lines 7 to 15). If all entries in x_{i*} are within their respective boundaries, the response is calculated. Otherwise the invalid entries of x_{i*} are drawn again until all entries are valid. According to Corana et al. (1987), the number of invalid trials can be a useful information to assess quality of the search domain. The numbers of invalid trials are thus counted and stored (line 13) in order to make this information accessible for the outer loop. Counting of valid as well as invalid trials is not reseted after each inner loop repetition. Both are only initialized in iteration 1 of the inner loop and increase till its last iteration. Next step is

the comparison of loss function returns (lines 16 and 17). If the return of current variables combination $f(x_{i*})$ is better than $f(x_j)$, x_{i*} and $f(x_{i*})$ are stored into x_i and $f(x_i)$. x_i are the initial covariates for the next iteration. Core idea of the classical SA approach is to cope with local optima. Thus, even if $f(x_{i*})$ is worse than $f(x_j)$, there is a chance of storing x_{i*} into x_i (lines 18 to 25). The likelihood of keeping worse responses depends on the Metropolis distribution M (Metropolis et al., 1953).

$$M = exp\left(-\frac{|\, f(i) - f(j)\,|}{kt}\right), \tag{1}$$

with k being a user definable constant. We adopted this strategy from classic SA without modifications. M decreases with decreasing temperature t. The likelihood of keeping worse responses thus decreases with decreasing temperature t (lines 19 and 24). t does not change during the entire inner for loop. The likelihood of keeping worse values is thus equal for each response until the inner loop is finished. Modification of t is part of the outer loop which will be explained in the next paragraph. In case a worse result is chosen the former optimal covariate combination is, of course, stored before it is overwritten since otherwise there is a sound chance of overwriting that actual global optimum. More details of the Metropolis distribution properties can i.a. be found in Kirkpatrick et al. (1983) and Metropolis et al. (1953). Storing informations on development of covariates and response can be help improving the performance of SA (Lin et al., 1995; Hansen, 2012). We implemented a threshold accepting strategy (Dueck and Scheuer, 1990) into our SA interpretation (lines 26 to 28). This criterion is the only module that allows reducing the inner loop repetitions without direct user influence. It is simply a vector where the absolute differences of $f(x_i)$ and $f(x_j)$ are sored. If the response oscillates for user defined number of repetitions within a user defined threshold, the inner loop breaks.

```
1  initialize t, vf with user specifications
2  calculate f(x_0) with initial parameter vector x_0
3  while t > t_min do
4      for i in c(1: n_inner) do
5          x_j ← x_(i−1)
6          call the variation function to generate x_i* in dependence of x_j, rf and t
7          check if all entries in x_i* are within the boundaries
8          if all x_i valid then
9              calculate f(x_i*)
10         else
11             while any(x_i* invalid) do
12                 call the variation function again
13                 count invalid combinations
14             end
15         end
16         if f(x_i*) < f(x_j) then
17             x_i ← x_i*; f(x_i) ← f(x_i*)
18         else
19             calculate Metropolis Probability M (Equation 1)
20             if uniformly distributed random number [0,1] < M then
21                 x_i ← x_i*; f(x_i) ← f(x_i*)
22             else
23                 x_i ← x_j; f(x_i) ← f(x_j)
24             end
25         end
26         if threshold accepting criterion fulfilled then
27             break inner loop
28         end
29     end
30     reduce t for the next iteration
31     rf adaptation for the next iteration
32 end
33 return optimized parameter vector, function value and some additional informations
```

Algorithm 1: Pseudocode of the `optim_sa` function in the **optimization** package exemplary for a minimization.

Main functions of the *outer* while loop (lines 3 to 32) are calling the inner loop (lines 3 to 29) and modifying the parameters that are needed in the inner loop (lines 30 and 31). t and rf therefore only change after completely finishing an inner loop. Both cannot be changed within the inner loop. The outer loop repeats till t is smaller than the user defined minimum temperature t_0. After finishing

the inner loop, firstly t is adjusted (line 30). t is necessary for the stochastic part in the inner loop (line 19, Equation 1). In our function, t is obligatory decreasing as it is calculated by multiplying the temperature of the current iteration by r which is a user defined real number between 0 and 1. The number of outer loops repetitions is thus implied by initial temperature t_0, t_{min} and r. In the following rf changes (line 31). The dynamic adaption of rf after Corana et al. (1987) and Pronzato et al. (1984) is another major novelty of our function. As each covariate can have its own random factor, rf is a vector of the same size as x_i. rf is needed for the covariate variation in the inner loop (lines 6 and 12). Dividing the numbers of invalid trials which were counted in the inner loop (line 13) distinctively for each covariate by the total number of trials of the respective covariate gives the ratio of invalid trials for each covariate. According to (Corana et al., 1987), this ratio can be used to find a trade-off between accuracy on the one and size of the search domain on the other side. They argument that if only valid covariate combinations are drawn, the search domain could be too small for multimodal problems. For this, the ratio of the current ratio is used to generate the rf for the following outer loop repetition. They suggest ratios between 0.4 and 0.6. If any observed ratio is < 0.4 or > 0.6, the respective random factors are modified following the suggested equation by Corana et al. (1987) *Formel darstellen?*. This strategy allows a adjustment of rf for the next iteration. Pronzato et al. (1984), who developed the *Adaptive Random Search method*, propose a time decreasing search domain. They thus suggest a search domain adjustment that does not depend on former information, as Corana et al. (1987) did, but on the number of iterations. They argument that the search domain should be wide at the beginning to give the algorithm the chance of coping local optima and small in the end to allow higher precisions since the algorithm should converge to the global optimum at the end. Later iteration thus need smaller search domains than earlier iterations. We combined the idea of Corana et al. (1987) into the dynamically search domain adjustment of (Corana et al., 1987) by linearly shrinking the favorable range of ratios from the suggested ratios (0.4-0.6) to 0.04-0.06. As mentioned in the inner loop explanations, the variation function can be user defined (lines 6 and 12). The user thus has the chance of defining flexibly in which way t and rf influence the covariate variation. Per default, the search domain around the covariates shrinks by trend with increasing number of outer loop iterations.

The function optim_sa

As the function is mainly written in C++, it requires **Rcpp**. `optim_sa` shall be able to solve very specific optimization problems, several parameters can be defined by the user. Quality of solution and speed of convergence will thus substantially depend on accurate parametrization. In the following, we will explain the most important parameters briefly giving hints for useful specification. A complete parameter list can be found in the vignette.

- `fun`: Obligatory loss function to be optimized. The function must depend on a vector of covariates and return one numeric value. There are no assumptions on covariates and return. The covariates even need not be continuous. Missing (`NA`) or undefined (`NaN`) returns are allowed as well. Any restriction on the parameter space, e.g. specific invalid covariate values within the boundaries, can be integrated in the loss function directly by simply returning NA. We will be more specific on this in the practical examples.
- `start`: Obligatory numeric vector with initial covariate combination. It must be ensured that at least the initial covariate combination leads to a defined numeric response. The loss function at the initial variables combination must therefore return a defined numeric value. This might be relevant when the starting values are determined stochastically.
- `trace`: If `TRUE`, the last inner loop iteration of each outer loop iteration is stored as row in the trace matrix. This might help evaluating the solutions quality. However, storing interim results increases calculation time up to 10 %. Disabling `trace` can thus improve efficiency when the convergence of an optimization problem is known to be stable.
- `lower,upper`: Numeric vector with lower boundaries of the covariates. The boundaries are obligatory since the dynamic `rf` adjustment (Corana et al., 1987; Pronzato et al., 1984) depends on the number of invalid covariate combinations.
- `control`: A list with optional further parameters.

All parameters in the list with `control` arguments have a default value. They are pre-parameterized for loss functions of medium complexity. `control` arguments are:

- `vf`: `vf` allows the user to restrict the parameter space. It is one of the most important differences to classic SA. The function determines the variation of covariates during the iterations. It is allowed to depend on `rf`, `temperature` and a vector of covariates of the current iteration. The variation function is a crucial element of `optim_sa` which enables flexible programming. It is (next to the loss function itself) the second possibility to define restrictions. The parameter space

of the optimization program can be defined `vf`. Per default, the covariates are changed by a continuous, uniform distributed random number. It must be considered that defining specific `rf` can increase the calculation time. The default `rf` is a compiled C++ function whereas user specified `rf` must be defined as R functions. User specified are e.g. useful for optimization problems with non-continuous parameter space.

- `rf`: Numeric vector with random factors. The random factors determine the range of the random number in the variation function vf relative to the dimension of the function variables. The rf can be stated separately for each variable. Default is a vector of ones. If `dyn_rf` is enabled, the entries in `rf` change dynamically over time.
- `dyn_rf`: If `TRUE`, `rf` change dynamically over time to ensure increasing precision with increasing number of iterations. `rf` determines whether the adjustments of Corana et al. (1987); Pronzato et al. (1984) are enabled (see method section for theoretical background). `dyn_rf` ensures a relatively wide search domain at the beginning of the optimization process that shrinks over time. Disabling `dyn_rf` can be useful when `rf` with high performance are known. The development of `rf` is documented in the `trace` matrix. Evaluation of former optimizations with dynamic `rf` can thus help finding efficient and reasonable fixed `rf`. Self-specified `vf` may not depend on `rf`. In this cases activating `dyn_rf` makes no sense.
- `t0`: Initial temperature. The temperature directly influences the likelihood of accepting worse responses thus the stochastic part of the optimization. `t0` should be adopted to the loss function complexity. Higher temperatures lead to higher ability of coping with local optima on the one but also to more time-consuming function calls on the other hand.
- `t_min`: numeric value that determines the temperature where outer loop stops. As there is practically no chance of leaving local optima in iterations with low temperature `t_min` mainly affects accuracy of the solution. Higher `t_min` yields to lower accuracy and less function calls.
- `nlimit`: Integer value which determines the maximum number of inner loops iterations. If the break criterion `stopac` is not fulfilled, `nlimit` is the exact number of inner loops repetitions. It is therefore an important parameter for determining the number of iterations.
- `r`: Numeric value that determines the reduction of the temperature at the end of each outer loop. Slower temperature reduction leads to increasing number of function calls. It should be parameterized with respect to `nlimit`. High `nlimit` in combination with low `r` lead to many iterations with the same acceptance likelihood of worse responses. Low `nlimit` in combination with `r` near 1, by contrast, lead to continuously decreasing acceptance likelihood of worse responses. It is thus the second crucial parameter for determining the number of iterations.

Examples

We briefly explain where the `optim_sa` function is advantageous.

Himmelblau Function with continuous parameter space

Himmelblau's function (Himmelblau, 1972) was chosen as initial example since it is a very simple multimodal equation which is widely known in Operations Research. It has 4 equal minimum values ($min(f(x_1, x_2)) = 0$)) at {-2.8, 3.1}, {3.0, 2.0}, {3.6, -1.8} and {-3.8, -3.3} (Equation 2). In order to display the advantages and basic behavior of **optimization**, it was compared with 2 other SA methods from the packages **stats** and **GenSA**. Himmelblau's function is relatively simple. We therefore also included the NM optimization method (Nelder and Mead, 1965) which is default of `optim` from the **stats** package to examine the advantages of random against direct search.

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \tag{2}$$

We performed 10,000 optimizations with each function in order investigate quality and speed of the solutions using parameters for relatively simple optimization problems for all examined methods. The optimizations were performed having the following parameters:

```
# stats package: optim (NM)
stats::optim(fn = hi, par = c(10, 10), method = "Nelder-Mead")

# optimization package: optim_sa
optimization::optim_sa(fun = hi, start = (c(10, 10)), trace = TRUE,
        lower = c(-40, -40), uppe r= c(40, 40),
                control = list(t0 = 500, nlimit = 50,r = 0.85,
```

Table 1: Relative frequencies of covariate combinations in % after optimization for the 4 examined methods. Number of repetitions: 10,000. We used the parameters given in the example except for deactivated trace option.

		Result (rounded)			
		{-2.8, 3.1}	{3.0, 2.0}	{3.6, -1.8}	{-3.8, -3.3}
Method	optim_sa	22.19	33.49	28.05	16.27
	optim (SA)	25.91	30.89	24.08	19.12
	GenSA	0.00	0.00	0.00	100.00
	optim (NM)	0.00	0.00	0.00	100.00

```
            rf = 3, ac_acc = 0.1, dyn_rf = TRUE))

# stats package optim (SA)
stats::optim(fn = hi, par = c(10, 10), method = "SANN",
        control = list(tmax = 500, reltol = 0.1, temp = 50, trace = TRUE))

# GenSA package: GenSA
GenSA::GenSA(fn = hi, par = c(10, 10), lower = c(-40, -40), upper = c(40, 40),
        control = list(temperature = 50, nb.stop.improvement = 30, maxit = 500))
```

Since we have a multimodal optimization problem with multiple equal solutions, evaluation of solutions quality is composed of response accuracy and covariate combination. With fixed starting parameters, the methods should be able to find all possible solutions. We defined the problem to be solved with sufficient accuracy when the response was ≤ 0.01. We also looked at the frequency distribution of the covariate combination after minimization. Subsequent to investigation of quality, we also compared the efficiencies by measuring calculation times using **microbenchmark**.

It became clear that parameterization of NM was quite easy. It only needed a vector with starting values. The other functions needed much more settings. With view to accuracy each method performed well. All functions returned in mean of 10,000 calls responses with values ≤ 0.01. With view to frequency distribution, the functions performed indifferent (Table 1). `optim_sa` and `optim (SA)` returned each possible solutions. The combination {-3.8, -3.3} was consistently least frequent. `GenSa` and `optim (NM)`, however, solely returned {-3.8, -3.3}. Further investigation revealed the solutions of `GenSa` and `optim (NM)` to be sensitive of the starting values. If `GenSa` and `optim (NM)` were parameterized with randomly drawn starting values, all 4 results would have been possible. Advantage of `optim_sa` and `optim (SA)` against `GenSa` and `optim (NM)` is, in this example, thus its independence from starting values.

As all functions were practically able to minimize equation 2, comparison of calculation times appears to be another important point for quality assessment. As expected, the direct search method `optim NM` was by far faster then all random search methods (Figure 1). The within-group ranking of random search methods revealed `GenSA` to be fastest. It was in average 3.5 times faster than `optim_sa`. The 2 functions able coping with equally valued optima (`optim (SA)` and `optim_sa`) were slower than `GenSA` (Table 1, Figure 1).

To conclude, all functions were generally able to solve the problem. The flexible random search-grid of `optim (SA)` and `optim_sa` enabled archiving each of the 4 solutions. Practically, in case users are in doubt whether a problem has multiple solutions with equal responses, `optim (SA)` and `optim_sa` can simply be repeated without re-parameterization. If well-specified, they will return all possible solutions. They are thus advantageous for complex multimodal problems of that kind on the one hand, but slower on the other hand. Due to the multiple additional options, which all have to be called in the code, `optim_sa` is slowest. This example clearly reveals that higher flexibility and generality leads to significantly higher calculation times. Specific algorithms are advantageous for complexer problems while more general functions are useful for optimization problems with simple responses.

Himmelblau Function with discrete parameter space

A second example illustrating the advantage of flexibly defining the parameter spaces also bases on the function by Himmelblau (1972). It is an exemplary optimization problem which cannot be solved with `optim()` or `GenSA`. If a user is interested in integer covariate combinations only, the variation function `vf` option can be used to restrict the parameter space accordingly. The following simple example shows a variation rule for integer programming problems. The simple `var_fun_int` from the example returns a vector of integer covariates varying in dimension of `vf` around the passed covariates `para_0`. The `fun_length` is passed explicitly, although it is implicitly given by the length of `para_0`

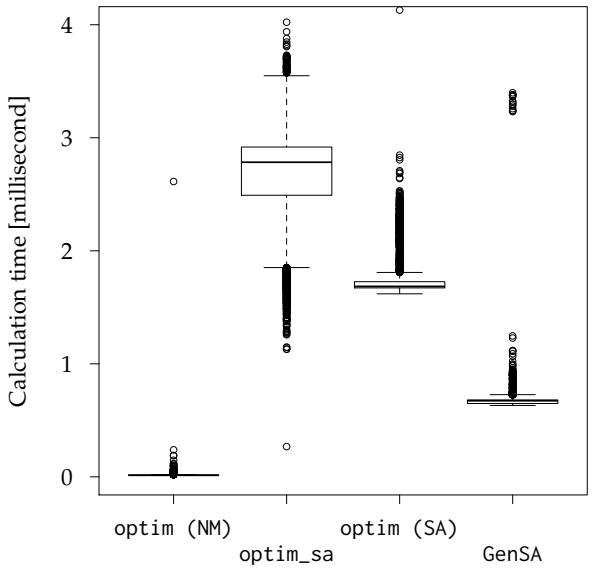

Figure 1: Calculation times of the 4 examined optimization algorithms. Note that the y-axis is truncated. `optim_sa` sowed 101 and `optim (SA)` 70 outliers between 4 and 7 milliseconds.

because it might help simplifying the `vf`. Dependence of `rf` and `temp` can be implemented. If `vf` does not need a dynamic part, i. e. if the stochastic part shall be fixed during the entire iterations, `temp` and `rf` can separately be disabled by passing `NA`. If `vf` does not depend on `rf`, the option `dyn_rf` will not be useful any more.

```
# Define vf
var_fun_int <- function (para_0, fun_length, rf, temp = NA) {
        ret_var_fun <- para_0 + sample.int(rf, fun_length, replace = TRUE) *
        ((rbinom(fun_length, 1, 0.5) * -2) + 1)
        return (ret_var_fun)
}

# Call optim_sa
int_programming <- optimization::optim_sa(fun = hi, start = c(10, 10), trace = TRUE,
        lower = c(-40, -40), upper=c(40, 40),
        control = list(t0 = 500, nlimit = 50,r = 0.85, rf = 3, ac_acc = 0.1,
                dyn_rf = TRUE, vf = var_func_int))
```

Calling the minimization function always leaded to the only integer solution {3, 2}. The generic plot function of `optimization` (Figure 2) helps interpreting the convergence and thus the solution quality as well as the algorithm behavior. Since the used example is 2-dimensional, a `contour_plot` (Figure 2. right) could be created. It shows the state space of the Himmelblau function in continuously changing colors. The results at the end of each outer loop iteration are shown as points within the parameter space. Reducing the actual parameter space from the example improves presentation. It became obvious that only integer covariate combinations were calculated during optimization (Figure 2, right). Figure 2 additionally helps explaining the stochastic part. It becomes clear that though response of iteration 12 (parameter combination {-2, 3}) was already quite near 0, following iteration ended with a relatively worse intermediate results. With response > 100, iteration 21 was much worse than iteration 12. The likelihood of keeping worse solutions shrined over time till it became practically 0 after iteration 30 (Figure 2, left). For problems of such kind, 40 iterations therefore should be far enough. This information could help the user parameterizing the function for the next problem.

`optim_sa` is advantageous against all other `SA` based algorithms when the parameter space of the optimization problem underlies extended restrictions. The user can define the valid parameter space within the lower and upper limits. To archive this, the user must define a changing rule in form of a R function. Users must thus examine their optimization problem very carefully and translate it into suitable functions. This example exhibits the properties of `optim_sa` as an example for specific optimization. Its parameterization is quite complex and time extensive but enables adopting the optimizer to problem specific needs. Other restrictions such as mixed integer problems may be

defined analogously. `vf` can thus be used to flexibly restrict the parameter space. Further hints for interpretation can be found in the package documentation.

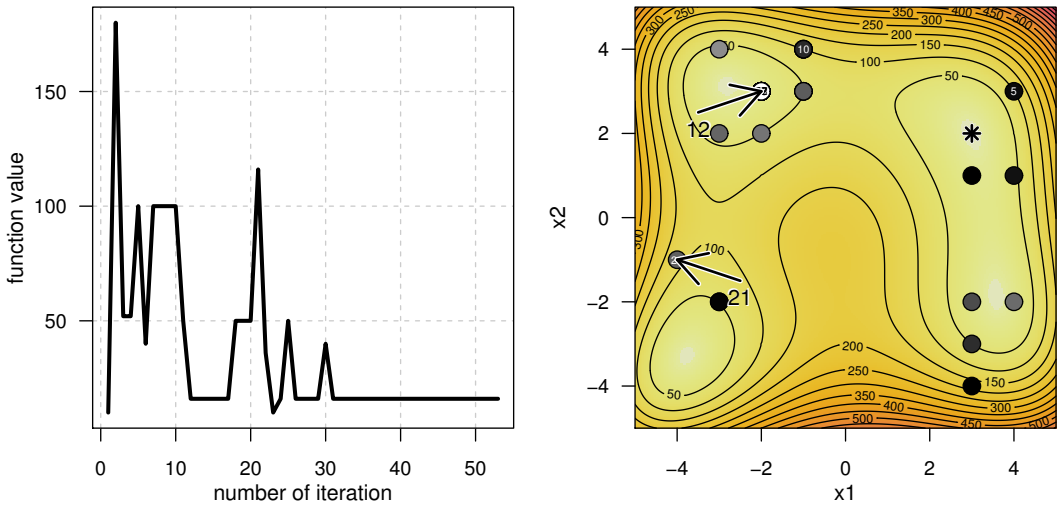

Figure 2: Examination plots created with the generic plot function. The left diagram shows the current optimal response over iteration of the outer loop. The left diagram displays the succession of the covariate values. The star points the covariate combination at optimum.

A complexer function?

Hat jemand eine Idee für eine weiteres Bsp.? Evtl. multimodal mit lokalen, also nicht gleichwertigen, Minima.

SVAR

Alex

Forest harvesting schedule optimization

Forestry is traditionally knowledge-based field with optimization playing only minor role. However popularity of optimization methods is steadily rising. While Linear Programs are nowadays used in some states, e.g. Finland (Redsven et al., 2012), stochastic optimization programs are quite novel in forestry (Kangas et al., 2015). Our function is integral part of the first stochastic optimization software of forest operation in Germany and one of the first softwares worldwide. Optimization of forest harvesting planning represents an interesting and innovative practical example where `optim_sa` is recently used. **optimization** is part of the ForestPlanner based decision support software for forest enterprises by Hansen and Nagel (2014). ForestPlanner is a user front end for TreeGrowthOpenSource-Software (TreeGrOSS) which is a complex Java written tree growth and yield simulation software used to forecast the developments of forest management areas (stands). It is a tool able to simulate forest enterprises with hundreds of forest stands simultaneously where the smallest simulation element is the single tree. Each tree in the system is thus simulated individually. Optimization of forest activities is accordingly not trivial since TreeGrOSS is a complex network of rules and functions which are predominantly nonlinear. The entire optimization process is composed of TreeGrOSS, **optimization** and a data warehouse. This multi-factorial composition implies high demand for flexibility of the optimization function. Loss function, which represents in this example the interface between the 3 elements of the optimization system, must be composed of R, Java (using **rJava**) and SQL (using **RPostgreSQL**). Main tasks of the interface are enabling communication between TreeGrOSS and optimization algorithm and rating the TreeGrOSS output in terms of costs and revenue such that the TreeGrOSS output is translated into an optimizable response. Flexibility of loss and variation functions is hence a prerequisite for forest harvesting schedule optimization via TreeGrOSS simulations. Each loss function call causes a TreeGrOSS simulation and a database operation. In order to save time, parts of the loss are therefore programmed parallel. The response is, accordingly, nonlinear and particularly non-continuous. Random search methods are of therefore favorable for the problem. Obligatory sustainability of forest stand treatment is also considered in form of restrictions. Each

function call returns, next to the actual response, a sustainability index. This index is used to restrict the optimization by simply defining `NA` responses whenever sustainability is violated.

It showed that forest treatment optimization was actually possible using **optimization**. We developed a loss function able translating the TreeGrOSS in- and output into interpretable variables for `optim_sa`. Sensitivity analysis using an exemplary forest enterprise comprised of 5 forest stands, with known global maximum, reinforced reliability of `optim_sa` for harvesting optimization. A solution sufficiently near the global maximum was found in arguable time on a standard personal computer. To test the practical usability, the optimization system was additionally tested on a real forest enterprise with 100 forest stands. The problem was solved within 12 hours using a cluster computer. Repeating the optimization 3 times leaded to equal responses but different harvesting volumes (thus covariate combinations). This again reveals the complexity of the problem and further reinforces the need for specific stochastic methods.

Discussion and outlook

It does not guarantee, of course, to find the global minimum, but if the function has many good near-optimal solutions, it should find one. In particular, this method is able to discriminate between gross behavior of the function and finer wrinkles. First, it reaches an area in the function domain where a global minimum should be present, following the gross behavior irrespectively of small local minima found on the way. It then develops finer details, finding a good, near-optimal local minimum, if not the global minimum itself.

Every method has advantages and disadvantages: GenSA for example is by far most efficient. It should be used for problems with 1 global optimum. optim_sa has its niche -> mult. equal opt. = felxible loss; NA returns -> enrichment of the optimization procedures in R

Example 1 shows -> Our is slowest but together with optim (SA) best Ex 2 shows -> Our is the only one that solves the problem –> Slow but flexible The (2?) practical examples show usability Xiang et al. (2013) calculated the likelihood of keeping worse result by the XX formula which sustantially improved the calculation time. This could be one explanation for the very performant ... Similar approaches could improve performance of our function as well.

Bibliography

F. P. Agostini, D. D. O. Soares-Pinto, M. A. Moret, C. Osthoff, and P. G. Pascutti. Generalized simulated annealing applied to protein folding studies. *Journal of computational chemistry*, 27(11):1142–1155, 2006. [p1]

E. Z. Baskent and G. A. Jordan. Forest landscape management modeling using simulated annealing. *Forest Ecology and Management*, 165(1):29–45, 2002. [p1]

C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003. [p1]

G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964. [p1]

A. Corana, M. Marchesi, C. Martini, and S. Ridella. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. *ACM Trans. Math. Softw.*, 13(3):262–280, 1987. [p1, 2, 4, 5]

G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. On a linear-programming, combinatorial approach to the traveling-salesman problem. *Operations Research*, 7(1):58–66, 1959. [p1]

W. S. DeSarbo, R. L. Oliver, and A. Rangaswamy. A simulated annealing methodology for clusterwise linear regression. *Psychometrika*, 54(4):707–736, 1989. [p1]

G. Dueck and T. Scheuer. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of computational physics*, 90(1):161–175, 1990. [p3]

C. Geiger and C. Kanzow. Das nelder-mead-verfahren. *Numerische Verfahren zur Lösung unregestrierter Optimierungsaufgaben*, 1999. [p1]

W. L. Goffe et al. Simann: A global optimization algorithm using simulated annealing. *Studies in Nonlinear Dynamics and Econometrics*, 1(3):169–176, 1996. [p1]

J. Hansen and J. Nagel. *Waldwachstumskundliche Softwaresysteme auf Basis von TreeGrOSS: Anwendung und theoretische Grundlagen*. Beiträge aus der Nordwestdeutschen Forstlichen Versuchsanstalt / Nordwestdeutsche Forstliche Versuchsanstalt. - Göttingen : Univ.-Verl. Göttingen, 2007- ; 11Universitätsdrucke Göttingen. Univ.-Verl., Göttingen, 2014. ISBN 978-3-86395-149-8. [p8]

J. H. Hansen. *Modellbasierte Entscheidungsunterstützung für den Forstbetrieb: Optimierung kurzfristiger Nutzungsoptionen und mittelfristiger Strategien unter Verwendung metaheuristischer Verfahren und parallelen Rechnens*. Cuvillier, Göttingen, 1. aufl. edition, 2012. [p3]

D. M. Himmelblau. *Applied nonlinear programming*. McGraw-Hill Companies, 1972. [p5, 6]

L. Ingber. Simulated annealing: Practice versus theory. *Mathematical and computer modelling*, 18(11): 29–57, 1993. [p1]

A. Kangas, M. Kurttila, T. Hujala, K. Eyvindson, and J. Kangas. *Decision Support for Forest Management*. Number 30 in Managing Forest Ecosystems. Springer International Publishing, Cham, 2nd ed edition, 2015. [p8]

S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598): 671–680, 1983. [p1, 2, 3]

C. Lin, K. Haley, and C. Sparks. A comparative study of both standard and adaptive versions of threshold accepting and simulated annealing algorithms in three scheduling problems. *European Journal of Operational Research*, 83(2):330–346, 1995. [p3]

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953. [p3]

J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7(4):308–313, 1965. [p5]

L. Pronzato, E. Walter, A. Venot, and J.-F. Lebruchec. A general-purpose global optimizer: Implimentation and applications. *Mathematics and Computers in Simulation*, 26(5):412–422, 1984. ISSN 03784754. doi: 10.1016/0378-4754(84)90105-8. [p1, 2, 4, 5]

V. Redsven, H. Hirvelä, K. Härkönen, O. Salminen, and M. Siitonen. *MELA2012 Reference Manual (2nd edition)*. The Finnish Forest Research Institute, 2012. [p8]

S. Theussl and H. W. Borchers. CRAN Task View: Optimization and Mathematical Programming. Oct. 2016. URL https://CRAN.R-project.org/view=Optimization. [p2]

I. Wegener. Simulated annealing beats metropolis in combinatorial optimization. In *International Colloquium on Automata, Languages, and Programming*, pages 589–601. Springer, 2005. [p1]

Y. Xiang, S. Gubian, B. Suomela, and J. Hoeng. Generalized simulated annealing for global optimization: the gensa package. *The R Journal Volume 5(1):13-29, June 2013*, 2013. [p1, 2, 9]

Author One
Affiliation
Address
Country
author1@work

Author Two
Affiliation
Address
Country
author2@work

Author Three
Affiliation
Address
Country
author3@work