Journal of Statistical Software
April 2011, Volume 40, Issue 6. http://www.jstatsoft.org/

DEoptim: An R Package for Global Optimization by Differential Evolution

Katharine M. Mullen
National Institute of Standards and Technology

David Ardia
aeris CAPITAL AG

David L. Gil
National Institute of Standards and Technology

Donald Windover
National Institute of Standards and Technology

James Cline
National Institute of Standards and Technology

Abstract

This article describes the R package **DEoptim**, which implements the differential evolution algorithm for global optimization of a real-valued function of a real-valued parameter vector. The implementation of differential evolution in **DEoptim** interfaces with C code for efficiency. The utility of the package is illustrated by case studies in fitting a Parratt model for X-ray reflectometry data and a Markov-switching generalized autoregressive conditional heteroskedasticity model for the returns of the Swiss Market Index.

Keywords: global optimization, evolutionary algorithm, differential evolution, R software.

1. Introduction

Optimization algorithms inspired by the process of natural selection have been in use since the 1950s (Mitchell 1998), and are often referred to as *evolutionary algorithms*. The genetic algorithm is one such method, and was invented by John Holland in the 1960s (Holland 1975). Genetic algorithms apply logical operations, usually on bit strings of fixed or variable length, in order to perform crossover, mutation, and selection on a population. Over the course of successive generations, the members of the population are more likely to represent a minimum of an objective function. Genetic algorithms have proven themselves to be useful heuristic methods for global optimization, in particular for combinatorial optimization problems. Evolution strategies are another variety of evolutionary algorithm, in which

Program	Language	Authors	Cross-platform
`DeApp`	Java	Storn (1999)	Yes
DeWin	MS Visual C++	Storn (2004)	No
DeMat	MATLAB	Storn *et al.* (2004)	No
DiffEvol	Scilab	Di Carlo and Jarausch (2006)	Yes
`DESolver`	MS Visual C++	Godwin (1998)	No
`DE_Fortran90`	Fortran 90	Wang (2004)	Yes
DeMat for Pascal	Pascal	Geldon and Gauden (2006)	Yes
DEoptim	R	Ardia and Mullen (2010)	Yes

Table 1: Implementations of DE for general purpose optimization.

members of the population are represented with floating point numbers, and the population is transformed over successive generations using arithmetic operations. See Price, Storn, and Lampinen (2006, Section 1.2.3) for a detailed overview of evolutionary algorithms.

In the 1990s Rainer Storn and Kenneth Price developed an evolution strategy they termed *differential evolution* (DE) (Storn and Price 1997). DE is particularly well-suited to find the global optimum of a real-valued function of real-valued parameters, and does not require that the function be either continuous or differentiable. In the roughly fifteen years since its invention, DE has been successfully applied in a wide variety of fields, from computational physics to operations research, as Price *et al.* (2006) catalogue.

Many implementations of DE are currently available. A web-based list of DE programs for general purpose optimization is maintained Storn (2010). A selection of programs from this list for which the source code is readily available are summarized in Table 1.

Commercial software such as Mathematica, MATLAB's **GA** toolbox, and a variety of special-purpose programs for optical and X-ray physics also implement DE.

The **DEoptim** implementation of DE was motivated by our desire to extend the set of algorithms available for global optimization in the R language and environment for statistical computing (R Development Core Team 2009). R enables rapid prototyping of objective functions, access to a wide array of tools for statistical modeling, and ability to generate customized plots of results with ease (which in many situations makes use of R preferable over the use of programs in languages like Java, MS Visual C++, Fortran 90 or Pascal). Furthermore, R is released in open-source form under the terms of the GNU General Public License, meaning that packages implemented for it do not require the purchase of commercial software. R also has a large and growing user base interested in optimization. **DEoptim** has been published on the Comprehensive R Archive Network and is available at `http://CRAN.R-project.org/package=DEoptim`. Since becoming publicly available it has been used by a variety of authors, e.g., Börner, Higgins, Kantelhardt, and Scheiter (2007), Higgins, Kantelhardt, Scheiter, and Boerner (2007), Cao, Vilar, and Devia (2009), Opsina Arango (2009), and Ardia, Boudt, Carl, Mullen, and Peterson (2010) to solve optimization problems arising in diverse domains.

In the remainder of this manuscript we elaborate on **DEoptim**'s implementation and use. In Section 1.1, the package is introduced via a simple example. Section 2 describes the underlying algorithm. Section 3 describes the R implementation and serves as a user manual. **DEoptim** is then illustrated via two cases studies, involving fitting a Parratt recursion model

for X-ray reflectometry data (in Section 4) and a Markov-switching generalized autoregressive conditional heteroscedasticity (MSGARCH) model for log-returns of the Swiss Market Index (in Section 5).

1.1. An introductory example

Minimization of the Rastrigin function in $x \in \Re^D$

$$f(x) = \sum_{j=1}^{D} \left(x_j^2 - 10\cos\left(2\pi x_j\right) + 10\right)$$

for $D = 2$ is a common test for global optimization algorithms.

This function is possible to represent in R as

```
R> rastrigin <- function(x) 10 * length(x) + sum(x^2 - 10 * cos(2 * pi * x))
```

As shown in Figure 1, for $D = 2$ the function has a global minimum $f(x) = 0$ at the point $(0, 0)$.

In order to minimize this function using **DEoptim**, the R interpreter is invoked, and the package is loaded with the command

```
R> library("DEoptim")

DEoptim package
Differential Evolution algorithm in R
Authors: David Ardia and Katharine Mullen
```

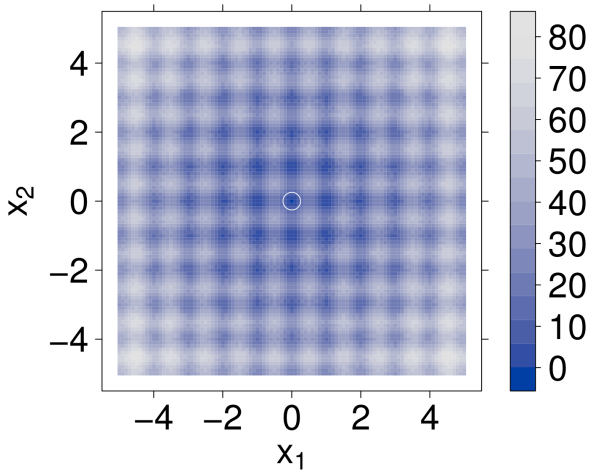

Figure 1: A contour plot of the two-dimensional Rastrigin function $f(x)$. The global minimum $f(x) = 0$ is at $(0, 0)$ and is marked with an open white circle.

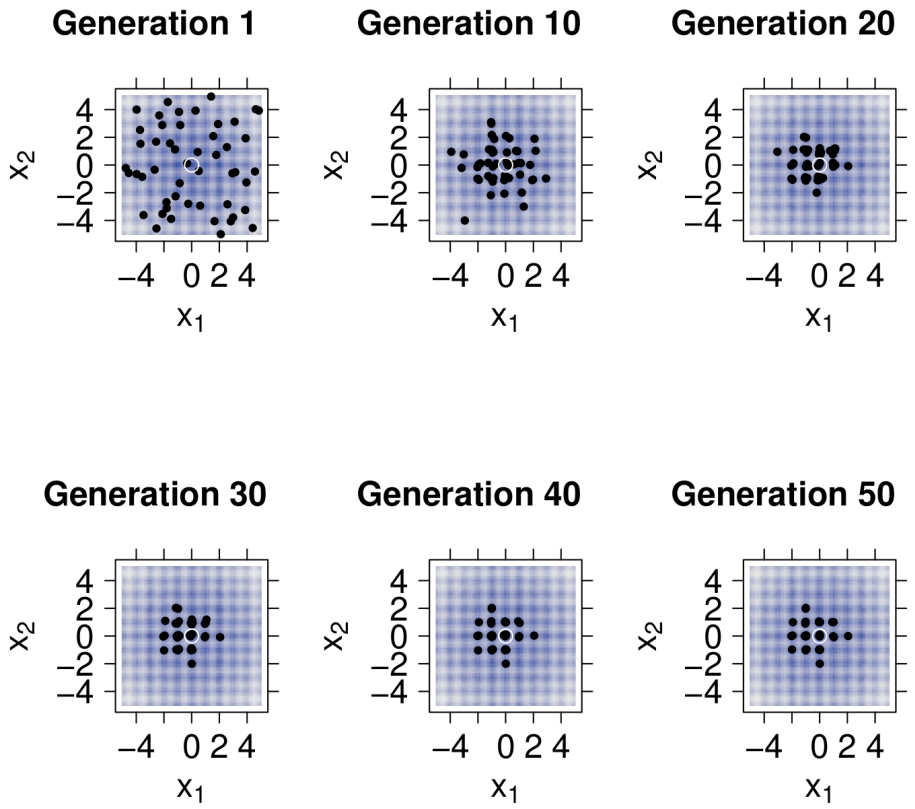

Figure 2: The population associated with various generations of a call to `DEoptim` as it searches for the minimum of the Rastrigin function (marked with an open white circle). The minimum is consistently determined within 200 generations using the default settings of `DEoptim`.

The `DEoptim` function of the package **DEoptim** searches for minima of the objective function between lower and upper bounds on each parameter to be optimized. Therefore in the call to `DEoptim` we specify vectors that comprise the lower and upper bounds; these vectors are the same length as the parameter vector. The call to `DEoptim` can be made as

```
R> est.ras <- DEoptim(rastrigin, lower = c(-5, -5), upper = c(5, 5),
+    control = list(storepopfrom = 1, trace = FALSE))
```

Note that the vector of parameters to be optimized must be the first argument of the objective function `fn` passed to `DEoptim`. The above call specifies the objective function to minimize, `rastrigin`, the lower and upper bounds on the the parameters, and, via the `control` argument, that we want to store intermediate populations from the first generation onwards (`storepopfrom = 1`), and do not want to print out progress information each generation (`trace = FALSE`). Storing intermediate populations allows us to examine the progress of the optimization in detail. Upon initialization, the population is comprised of 50 vectors x of length two (50 being the default value of `NP`), with x_i a random value drawn from the uniform

distribution over the values defined by the associated lower and upper bound. The operations of crossover, mutation, and selection explained in Section 2 transform the population so that the members of successive generations are more likely to represent the global minimum of the objective function. The members of the population generated by the above call are plotted at the end of different generations in Figure 2. `DEoptim` consistently finds the minimum of the function within 200 generations using the default settings. We have observed that **DEoptim** solves the Rastrigin problem more efficiently than the simulated annealing method found in the R function `optim`.

We note that as the dimensionality of the Rastrigin problem increases, **DEoptim** may not be able to find the global minimum in the default number of generations. Heuristics to help ensure that the global minimum is found include re-running the problem with a larger population size (value of `NP`), and increasing the maximum allowed number of generations.

1.2. Problems suitable for DE

Differential evolution does not require derivatives of the objective function. It is therefore useful in situations in which the objective function is stochastic, noisy, or difficult to differentiate. DE, however, may be inefficient on smooth functions, where derivative-based methods generally are most efficient.

In the example below, a generalized Rosenbrock function is considered. This function is differentiable and has a single local minima. It is often more efficient to apply methods other than DE to optimization of such functions. Functions that are smooth but have many local minima, however, may still be good candidates for optimization with DE, since alternative algorithms for local, as opposed to global, optimization may converge to a sub-optimal solution.

A generalized Rosenbrock function is possible to represent in R as

```
R> genrose.f <- function(x) {
+     n <- length(x)
+     fval <- 1 + sum(100 * (x[1:(n - 1)]^2 - x[2:n])^2 + (x[2:n] - 1)^2)
+     return(fval)
+ }
```

This function has a global minimum at 1, which **DEoptim** finds for $n = 10$ with a call like:

```
R> n <- 10
R> ans <- DEoptim(fn = genrose.f, lower = rep(-5, n), upper = rep(5, n),
+    control = list(NP = 100, itermax = 4000, trace = FALSE))
```

The minimum can be determined with far fewer function evalutations with a gradient-based method such as "BFGS" (Nash 1990), e.g., with the call

```
R> ans1 <- optim(par = runif(10, -5, 5), fn = genrose.f, method = "BFGS",
+     control = list(maxit = 4000))
```

Note further that users interested in exact reproduction of results should set the seed of their random number generator before calling **DEoptim**. DE is a randomized algorithm, and the results may vary between runs.

2. The differential evolution algorithm

We sketch the classical DE algorithm here and refer interested readers to the work of Storn and Price (1997) and Price *et al.* (2006) for further elaboration. The algorithm is an evolutionary technique which at each generation transforms a set of parameter vectors, termed the population, into another set of parameter vectors, the members of which are more likely to minimize the objective function. In order generate a new parameter vector, DE disturbs an old parameter vector with the scaled difference of two randomly selected parameter vectors.

The variable *NP* represents the number of parameter vectors in the population. At generation 0, *NP* guesses for the optimal value of the parameter vector are made, either using random values between upper and lower bounds for each parameter or using values given by the user. Each generation involves creation of a new population from the current population members $x_{i,g}$, where i indexes the vectors that make up the population and g indexes generation. This is accomplished using *differential mutation* of the population members. A trial mutant parameter vector $v_{i,g}$ is created by choosing three members of the population, $x_{r0,g}$, $x_{r1,g}$ and $x_{r2,g}$, at random. Then $v_{i,g}$ is generated as

$$v_{i,g} \doteq x_{r0,g} + \mathrm{F} \cdot (x_{r1,g} - x_{r2,g}) \tag{1}$$

where F is a positive scale factor. Effective values of F are typically less than 1.

After the first mutation operation, mutation is continued until either length(x) mutations have been made or $rand > CR$, where CR is a crossover probability $CR \in [0, 1]$, and where here and throughout *rand* is used to denote a random number from $\mathcal{U}(0, 1)$. The crossover probability CR controls the fraction of the parameter values that are copied from the mutant. CR approximates but does not exactly represent the probability that a parameter value will be inherited from the mutant, since at least one mutation always occurs. Mutation is applied in this way to each member of the population.

If an element v_j of the parameter vector is found to violate the bounds after mutation and crossover, it is reset, where here and throughout we use j to index into a parameter vector. In the implementation of **DEoptim**, if $v_j > \mathrm{upper}_j$, it is reset as $v_j \doteq \mathrm{upper}_j - rand \cdot (\mathrm{upper}_j - \mathrm{lower}_j)$, and if $v_j < \mathrm{lower}_j$, it is reset as $v_j \doteq \mathrm{lower}_j + rand \cdot (\mathrm{upper}_j - \mathrm{lower}_j)$. This ensures that candidate population members found to violate the bounds are set some random amount away from them, in such a way that the bounds are guaranteed to be satisfied. Then the objective function values associated with the children v are determined. If a trial vector $v_{i,g}$ has equal or lower objective function value than the vector $x_{i,g}$, $v_{i,g}$ replaces $x_{i,g}$ in the population; otherwise $x_{i,g}$ remains. The algorithm stops after some set number of generations, or after the objective function value associated with the best member has been reduced below some set threshold.

Variations on this theme are possible, some of which are described in the following section. Values of *NP* and *CR* that have been found to be most effective for a variety of problems are described in Price *et al.* (2006, Section 2). Reasonable default values for many problems are given in the following section.

3. Implementation

DEoptim was first published on the Comprehensive R Archive Network (CRAN) in 2005 by David Ardia. Early versions were written in pure R. Since version 2.0-0 (published to CRAN

in 2009 by Katharine Mullen) the package has relied on an interface to a C implementation of DE, which is significantly faster on most problems as compared to the implementation in pure R. Since version 2.0-3 the C implementation dynamically allocates the memory required to store the population, removing limitations on the number of members in the population and length of the parameter vectors that may be optimized.

The implementation is used by calling the R function `DEoptim`, the arguments of which are:

- `fn`: The objective function to be minimized. This function should have as its first argument the vector of real-valued parameters to optimize, and return a scalar real result.
- `lower`, `upper`: Vectors specifying scalar real lower and upper bounds on each parameter to be optimized, so that the ith element of `lower` and `upper` applies to the ith parameter. The implementation searches between `lower` and `upper` for the global optimum of `fn`.
- `control`: A list of control parameters, discussed below.
- `...`: allows the user to pass additional arguments to the function `fn`.

The `control` argument is a list, the following elements of which are currently interpreted:

- `VTR`: The value to reach. Specify the global minimum of `fn` if it is known, or if you wish to cease optimization after having reached a certain value. The default value is `-Inf`.
- `strategy`: This defines the differential evolution strategy used in the optimization procedure, described below in the terms used by Price *et al.* (2006):
  - `1`: DE / rand / 1 / bin (classical strategy). This strategy is the classical approach described in Section 2.
  - `2`: DE / local-to-best / 1 / bin. In place of the classical DE mutation given in (1), the expression

$$v_{i,g} \doteq \text{old}_{i,g} + \text{F} \cdot (\text{best}_g - \text{old}_{i,g}) + \text{F} \cdot (x_{r1,g} - x_{r2,g})$$

    is used, where $\text{old}_{i,g}$ and best_g are the ith member and best member, respectively, of the previous population. This strategy is currently used by default.
  - `3`: DE / best / 1 / bin with jitter. In place of the classical DE mutation given in (1), the expression

$$v_{i,g} \doteq \text{best}_g + \text{jitter} + \text{F} \cdot (x_{r1,g} - x_{r2,g})$$

    is used, where jitter is defined as $0.0001 \cdot rand + F$.
  - `4`: DE / rand / 1 / bin with per vector dither. In place of the classical DE mutation given in (1), the expression

$$v_{i,g} \doteq x_{r0,g} + \text{dither} \cdot (x_{r1,g} - x_{r2,g})$$

    is used, where dither is calculated as $\text{dither} \doteq F + rand \cdot (1 - F)$.
  - `5`: DE / rand / 1 / bin with per generation dither. The strategy described for `4` is used, but dither is only determined once per-generation.

– any value not above: variation to DE / rand / 1 / bin: either-or algorithm. In the case that $rand < 0.5$, the classical strategy described for `1` is used. Otherwise, the expression

$$v_{i,g} \doteq x_{r0,g} + 0.5 \cdot (F + 1.0) \cdot (x_{r1,g} + x_{r2,g} - 2 \cdot x_{r0,g})$$

is used.

- `bs`: If `FALSE` then every mutant will be tested against a member in the previous generation, and the best value will survive into the next generation. This is the standard trial vs. target selection described in Section 2. If `TRUE` then the old generation and `NP` mutants will be sorted by their associated objective function values, and the best `NP` vectors will proceed into the next generation (this is best-of-parent-and-child selection). The default value is `FALSE`.

- `NP`: Number of population members. The default value is `50`.

- `itermax`: The maximum iteration (population generation) allowed. The default value is `200`.

- `CR`: Crossover probability from interval $[0, 1]$. The default value is `0.9`.

- `F`: Stepsize from interval $[0, 2]$. The default value is `0.8`.

- `trace`: Logical value indicating whether printing of progress occurs at each iteration. The default value is `TRUE`.

- `initialpop`: An initial population used as a starting population in the optimization procedure, specified as a matrix in which each row represents a population member. May be useful to speed up convergence. Defaults to `NULL`, so that the initial population is generated randomly within the lower and upper boundaries.

- `storepopfrom`: From which generation should the following intermediate populations be stored in memory. Default to `itermax + 1`, i.e., no intermediate population is stored.

- `storepopfreq`: The frequency with which populations are stored. The default value is `1`, i.e., every intermediate population is stored.

- `checkWinner`: Logical value indicating whether to re-evaluate the objective function using the winning parameter vector if this vector remains the same between generations. This may be useful for the optimization of a noisy objective function. If `checkWinner = TRUE` and `avWinner = FALSE` then the value associated with re-evaluation of the objective function is used in the next generation. Default to `FALSE`.

- `avWinner`: Logical value. If `checkWinner = TRUE` and `avWinner = TRUE` then the objective function value associated with the winning member represents the average of all evaluations of the objective function over the course of the 'winning streak' of the best population member. This option may be useful for optimization of noisy objective functions, and is interpreted only if `checkWinner = TRUE`. The default value is `TRUE`.

The default value of `control` is the return value of `DEoptim.control()`, which is a list with the above elements and specified default values.

The return value of the `DEoptim` function is a member of the S3 class `DEoptim`. Members of this class have a `plot` method that accepts the argument `plot.type`. When `retVal` is an object returned by **DEoptim**, calling `plot(retVal, plot.type = "bestmemit")` results in a plot of the parameter values that represent the lowest value of the objective function each generation. Calling `plot(retVal, plot.type = "bestvalit")` plots the best value of the objective function each generation. Calling `plot(retVal, plot.type = "storepop")` results in a plot of stored populations (which are only available if these have been saved by setting the `control` argument of `DEoptim` appropriately). A summary method for objects of S3 class `DEoptim` also exists, and returns the best parameter vector, the best value of the objective function, the number of generations optimization ran, and the number of times the objective function was evaluated.

A note on recommended settings: We have set the default values to the methods recommended by Price *et al.* (2006) as starting points. We use `strategy = 2` by default; the user should consider trying as alternatives `strategy = 6` and `strategy = 1`, though the best method will be highly problem-dependent. Generally, the user should set the lower and upper bounds to exploit the full allowable numerical range, i.e., if a parameter is allowed to exhibit values in the range [-1, 1] it is typically a good idea to pick the initial values from this range instead of unnecessarily restricting diversity. Increasing the value for `NP` will mean greater likelihood of finding the minimum, but run-time will be longer.

4. Application I: X-ray reflectometry

X-ray reflectometry (XRR) is a measurement method that uses the interference of X-rays (i.e., photons with a wavelength in the approximate range of 0.01 nm–10 nm) caused by changes in a material's electron density to characterize thin films or other layered structures at the nanometer to micrometer scale. The data collected consists of pairs of incident/scattered angle and scattered X-ray intensities, $\{(\theta_r, I_r)\}$, typically over a range of about 5 degrees. Information regarding the density and thickness of each layer, and on the roughness of the interface between layers and at the surface of the material is extracted by fitting a parametric model to the measurements.

In the supplementary information we provide the full description of a model function used by **DEoptim** to obtain physically realistic parameter estimates from the data shown in Figure 3. This model is based on the Parratt recursion (Parratt 1954), which, as Als-Nielsen and McMorrow (2001) describe in detail, is often used to model each of the layers in multilayered materials. For the data here, the Parratt recursion is used to describe reflection and transmission of X-rays from two thin layers of Pt (with each layer having a possibly distinct thickness, density, and roughness at the interface) atop an infinitely thick layer of SiO_2. Figure 4 is a schematic description of the model for this multilayered material.

The free parameters of the applied Parratt recursion model are the thicknesses d_1 and d_2 of each Pt layer, the density α_1 and α_2 of each Pt layer, terms ρ_1, ρ_2 and ρ_3 descriptive of the roughness of the interfaces between layers and at the surface, a parameter b describing a linear background, and a multiplicative scaling parameter m. The model function can be understood qualitatively by considering the the case of a single layer on a substrate. In this case, the

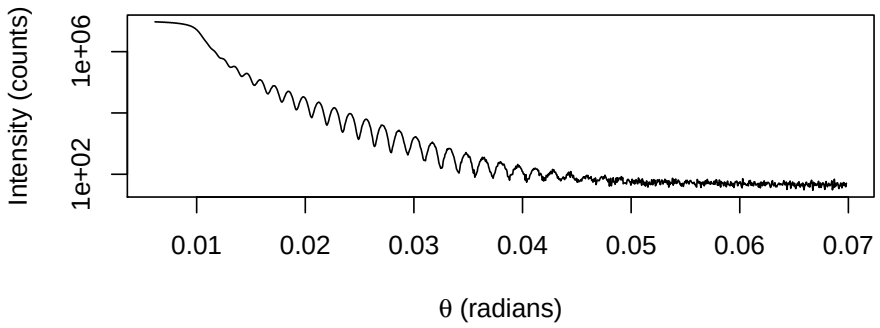

Figure 3: XRR measurements of Pt layers on SiO_2 substrate.

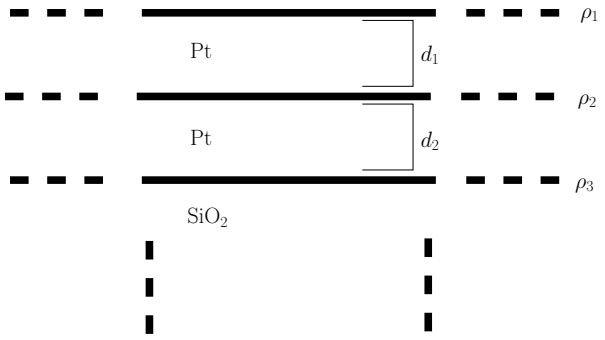

Figure 4: Schematic description of two layers of Pt on a substrate of SiO_2. A Parratt recursion model representing this structure will be fit to the XRR measurements, with free parameters including the thickness of the Pt layers (d_1 and d_1), and terms (ρ_1, ρ_2, and ρ_3) describing the roughness of the interfaces between layers.

position of the abrupt drop-off in scattered intensity after the initial plateau is determined by the density of the layer. The period of the subsequent oscillation fringes is set by the thickness of the layer, whereas the decay of the oscillations is a function of the roughness of the layer. Because the amplitudes of reflected and transmitted waves interfere, this qualitative view cannot be extended to multilayered systems and model fitting is a necessity.

The objective function R to minimize is formulated as the sum of the squared differences between the log of the data and the log of the Parratt recursion model function. The surface of objective function values in the 9-dimensional parameter space contains many local minima. Discovery of parameter estimates that represent a qualitatively good fit requires a global optimization algorithm such as DE. Treatment of global optimization problems such as these

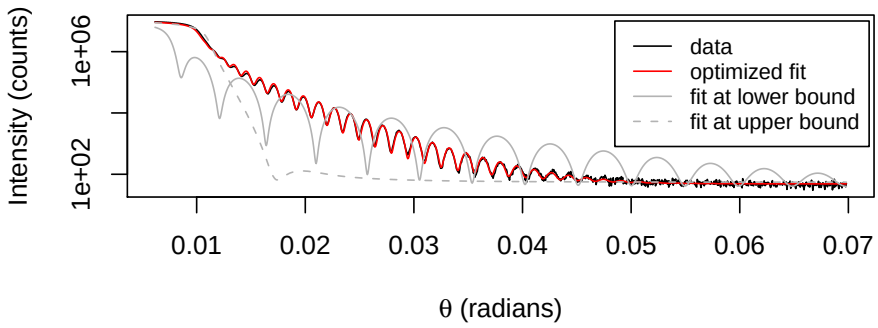

Figure 5: XRR measurements (black) of Pt layers on SiO_2 substrate with model fit (red). For comparison, the model has also been evaluated at the lower and upper bounds on the parameters used in the call to `DEoptim` (solid and dashed grey, respectively).

have been successfully addressed for many years in the XRR community using DE as, e.g., Wormington, Panaccione, Matney, and Bowen (1999), Taylor, Wall, Loxley, Wormington, and Lafford (2001), and Bowen and Tanner (2006) describe. Special purpose programs, e.g., the *GenX* program developed by Björck and Andersson (2007) and the *MOTOFIT* program developed by Nelson (2006), have been implemented for XRR model fitting problems of this sort.

The XRR measurements shown in Figure 3 are included in **DEoptim** as the dataset `xrrData`, with the vector of data to be fit represented by the vector `counts`. We have encoded the objective function R as the function `rss`. Using knowledge of the physical system underlying the measurements in order to set plausible lower and upper bounds on the parameters to optimize, and to set fixed values for `beta`, `wavelength`, `theta_r` and `delta`, the objective function is minimized with the call

```
R> parrattFit <- DEoptim(lower = c(d_1 = 5.5e-10, d_2 = 1.5e-08,
+    rho_1 = 2.1e-10, rho_2 = 5.0e-12, rho_3 = 2.2e-10, alpha_1 = 10,
+    alpha_2 = 10, b = 40, m = .90e7), upper = c(d_1 = 5.5e-09,
+      d_2 = 1.5e-07, rho_1 = 2.1e-09, rho_2 = 5.0e-11, rho_3 = 2.2e-09,
+      alpha_1 = 21.46, alpha_2 = 21.46, b = 55,  m = 1.1e7),
+    fn = rss, theta_r = theta_r, delta = delta, beta = beta,
+    wavelength = wavelength, data = counts,
+    control = list(itermax = 1500, NP = 90))
```

Table 2 gives parameter estimates arrived at via the above call, along with the associated lower and upper bounds. The resulting fit of the model to the data is shown in Figure 5. The upper bound for the density of the Pt layers was set at 21.46 g·cm^{-3}, the density of Pt in bulk. The estimates for the densities (19.6 g·cm^{-3} and 20.9 g·cm^{-3}) are slightly lower than for the bulk material. The remaining parameter estimates are also plausible from physical first principles, though the evaluation of the ability of the model to describe the material underlying the XRR measurements is beyond the scope of this paper. As shown in Figure 5,

	d_1	d_2	ρ_1 / nm	ρ_2	ρ_3	α_1 / g·cm^{-3}	α_2	b / counts	m
`lower`	0.55	15.0	0.21	0.005	0.22	10.0	10.0	40.0	0.90e7
`upper`	5.50	150.0	2.10	0.050	2.20	22.0	22.0	55.0	1.20e7
`bestmem`	1.69	45.6	0.62	0.0053	0.69	20.0	21.0	48.0	1.1e7

Table 2: Parameter estimates (`bestmem`) and lower and upper bounds associated with the call to **DEoptim** that results in the fit of Parratt recursion model to XRR data shown in Figure 5. Parameters d_1 and d_2 represent the thickness of the Pt layers, parameters ρ_1, ρ_2, ρ_3 describe the roughness of the interfaces between layers, and parameters α_1 and α_2 represent the density of the Pt layers. Parameter b represents an additive background term, and parameter m represents a multiplicative scaling factor for the intensity. Estimates are reported to two significant figures, except for t_1 and t_2, which are reported to three.

the model fit captures the qualitative features of the dataset well. The robustness of the estimates has been validated via initialization of DE using a variety of starting populations; the estimates presented in Table 2 reliably represent the best results obtained.

The function `rss` encoding the objective function can easily be customized to the dataset at hand, allowing, for instance, inclusion of more or fewer free parameters. Note that in this example, the population size, `NP`, was set to 90 since in practice it has been observed that convergence to the global optimum is facilitated if `NP` is at least ten times the number of parameters being optimized (Price *et al.* 2006). Determination of whether the best member returned by `DEoptim` with this call represents a unique global minimum is beyond the scope of this paper, but would be interesting to check for the purpose of developing a physical interpretation of the model fit.

5. Application II: Log-returns of the Swiss Market Index

Volatility plays a central role in empirical finance and financial risk management. Research on changing volatility (i.e., conditional variance) using time series models has been active since the creation of the original ARCH (autoregressive conditional heteroscedasticity) and GARCH (generalized ARCH) models. Since then, GARCH type models grew rapidly into a rich family of empirical models for volatility forecasting during the last twenty years. They are now widespread and essential tools in financial econometrics.

In the GARCH(p, q) specification introduced by Bollerslev (1986), the conditional variance at time t of the log-return y_t (of a financial asset or a financial index), denoted by σ_t^2, is postulated to be a linear function of the squares of past q log-returns and past p conditional variances. More precisely:

$$\sigma_t^2 \doteq \alpha_0 + \sum_{i=1}^{q} \alpha_i \, y_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-1}^2 \,, \tag{2}$$

where the parameters $\alpha_0 > 0$, $\alpha_i \geq 0$ $(i = 1, \ldots, q)$ and $\beta_j \geq 0$ $(j = 1, \ldots, p)$ in order to ensure a positive conditional variance. In most empirical applications it turns out that the

simple specification $p = q = 1$ is able to reproduce the volatility dynamics of financial data. This has led the GARCH$(1, 1)$ model to become the *workhorse* model by both academics and practitioners.

Numerous extensions and refinements of the GARCH$(1, 1)$ model have been proposed to mimic additional stylized facts observed in financial markets, such as nonlinearity, asymmetry, and long memory properties in the volatility process; see Bollerslev, Chou, and Kroner (1992) and Bollerslev, Engle, and Nelson (1994) for a review. Among them, the class of Markov-switching GARCH (MSGARCH) has gained particular attention in recent years. In these models, the parameters of the scedastic function (2) can change over time according to a latent (i.e., unobservable) variable taking values in the discrete space $\{1, \ldots, K\}$ where K is an integer defining the number of regimes or states. The interesting feature of these models lies in the fact that they provide an explanation of the high persistence in volatility, i.e., nearly unit root process for the conditional variance, observed with single-regime GARCH models (Lamoureux and Lastrapes 1990). Furthermore, these models are apt to react quickly to changes in the volatility level (unconditional volatility) which leads to significant improvements in volatility forecasts as shown by Dueker (1997) or Klaassen (2002) for instance. These features make the models attractive for various applications in financial modeling, such as risk management.

While MSGARCH models are attractive for the description of a variety of phenomena, we face practical difficulties when attempting to fit their parameters to data. The maximization of the likelihood function is a constrained optimization problem since some (or all) of the model parameters must be positive to ensure a positive conditional variance. It is also common to require that the covariance stationarity condition holds; this leads to additional non-linear inequality constraints which render the optimization procedure cumbersome. Optimization results are often sensitive to the choice of starting values. Finally, convergence is hard to achieve if the true parameter values are close to the boundary of the parameter space and if the underlying process is nearly non-stationary. For these reasons, a robust optimizer is required. DE offers an adequate approach to finding the maximum likelihood parameter estimates in this framework.

In order to illustrate the robustness of **DEoptim** compared to traditional estimation techniques, we consider a two-state asymmetric MSGARCH model investigated in Ardia (2008, Chapter 7). The author illustrated the poor performance of traditional local optimizers when estimating such sophisticated models. Only computationally demanding Markov chain Monte Carlo techniques were able to provide meaningful results.

A two-state Markov-switching asymmetric GARCH$(1, 1)$ model with Student-t innovations for the log-returns $\{y_t\}$ may be written as

$$\begin{aligned} y_t &= \varepsilon_t \sqrt{\tfrac{\nu-2}{\nu}\,\sigma^2_{s_t,t}} \quad t = 1, \ldots, T\,, \\ \varepsilon_t &\overset{i.i.d.}{\sim} \mathcal{S}(0, 1, \nu)\,, \\ \sigma^2_{i,t} &\doteq \begin{cases} \omega_1 + \left(\alpha_1^+ 1_{\{y_{t-1}\geq 0\}} + \alpha_1^- 1_{\{y_{t-1}<0\}}\right) y^2_{t-1} + \beta_1\, \sigma^2_{1,t-1} & \text{when } s_t = 1 \\ \omega_2 + \left(\alpha_2^+ 1_{\{y_{t-1}\geq 0\}} + \alpha_2^- 1_{\{y_{t-1}<0\}}\right) y^2_{t-1} + \beta_2\, \sigma^2_{2,t-1} & \text{when } s_t = 2\,, \end{cases} \end{aligned} \tag{3}$$

where $\omega_i > 0$, $\alpha_i^+, \alpha_i^-, \beta_i \geq 0$ $(i = 1, 2)$ and $\nu > 2$. The restriction on the degrees of freedom parameter ν ensures that the conditional variance $\sigma^2_{i,t}$ remains finite; the restrictions on the GARCH parameters $\omega_i, \alpha_i^+, \alpha_i^-$ and β_i guarantee its positivity. t is the time index and T denotes the total number of observations. $1_{\{\cdot\}}$ denotes the indicator function which is equal

to one if the constraint holds and zero otherwise. The sequence $\{s_t\}$ is assumed to be a stationary, irreducible Markov process with discrete state space $\{1, 2\}$ and transition matrix $P \doteq [p_{ij}]$ where $p_{ij} \doteq \mathsf{P}(s_{t+1} = j \,|\, s_t = i)$ is the transition probability of moving from state i to state j $(i, j \in \{1, 2\})$. Finally, $\mathcal{S}(0, 1, \nu)$ denotes the standard Student-t density with ν degrees of freedom and $\sqrt{(\nu-2)/\nu}$ is a scaling factor which ensures that the conditional variance of y_t is $\sigma^2_{s_t,t}$.

Model specification (3) allows reproduction of the so-called *volatility clustering* observed in financial returns, i.e., the fact that large changes tend to be followed by large changes (of either sign) and small changes tend to be followed by small changes. Moreover, it allows for sudden changes in the unconditional variance of the process; in the ith regime, the unconditional variance is

$$\bar{\sigma}^2_i \doteq \frac{\omega_i}{1 - (\alpha_i^+ + \alpha_i^-)/2 - \beta_i}\,, \tag{4}$$

provided that $(\alpha_i^+ + \alpha_i^-)/2 + \beta_i < 1$ (i.e., the process is covariance stationary); see Bollerslev (1986, page 310). Finally, it allows determination of whether or not an asymmetric response is present (i.e., $\alpha_i^- > \alpha_i^+$ for at least one i) and is different between the regimes (i.e., $\alpha_i^- \neq \alpha_{i'}^-$). This asymmetric response, referred to as the leverage effect in the financial literature, reflects the fact that the volatility tends to rise more in response to bad news (i.e., $y_{t-1} < 0$) than to good news (i.e., $y_{t-1} \geq 0$).

In order to write the likelihood function corresponding to model (3), we define the vector of log-returns $y \doteq (y_1, \ldots, y_T)'$ and we regroup the eleven model parameters into the vector

$$\theta \doteq (\omega_1, \omega_2, \alpha_1^+, \alpha_2^+, \alpha_1^-, \alpha_2^-, \beta_1, \beta_2, p_{11}, p_{22}, \nu)'\,.$$

The conditional density of y_t in state $s_t = i$ given θ and the information set $I_{t-1} \doteq \{y_{t-1}, \ldots, y_1\}$ is

$$f(y_t \,|\, s_t = i, \theta, I_{t-1}) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{\pi(\nu-2)\sigma^2_{i,t}}}\left[1 + \frac{y_t^2}{\sigma^2_{i,t}(\nu-2)}\right]^{(\nu+1)/2}\,,$$

where $\Gamma(\cdot)$ denotes the Gamma function. This stems from the fact that in state i, y_t follows a Student-t distribution with mean zero, variance $\sigma^2_{i,t}$ and degrees of freedom ν.

By integrating out the state variable s_t, we can obtain the density of y_t given θ and I_{t-1} only. The (discrete) integration is obtained as follows:

$$f(y_t \,|\, \theta, I_{t-1}) = \sum_{i=1}^{2}\sum_{j=1}^{2} p_{ij}\,\eta_{i,t-1}\,f(y_t \,|\, s_t = j, \theta, I_{t-1})\,, \tag{5}$$

where $\eta_{i,t-1} \doteq \mathsf{P}(s_{t-1} = i \,|\, \theta, I_{t-1})$ is the *filtered probability* of state i at time $t-1$ and where we recall that p_{ij} denotes the transition probability of moving from state i to state j. The filtered probabilities $\{\eta_{i,t}; i = 1, 2; t = 1, \ldots, T\}$ are obtained by an iterative algorithm similar in spirit to a Kalman filter; we refer the reader to Hamilton (1989) and Hamilton (1994, Chapter 22) for details.

Finally, the log-likelihood function corresponding to model specification (3) is obtained from (5) as follows:

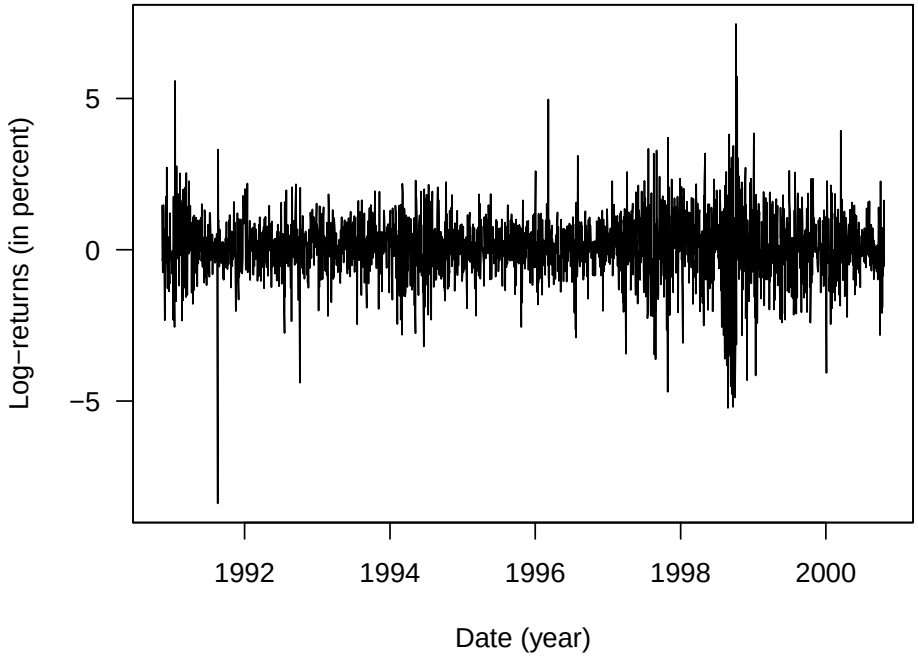

Figure 6: SMI daily log-returns.

$$\mathcal{L}(\theta \,|\, y) \doteq \sum_{t=2}^{T} \ln f(y_t \,|\, \theta, I_{t-1}) \,. \tag{6}$$

The maximum likelihood estimator $\hat{\theta}$ is obtained by maximizing (6) (or minimizing its negative value).

To illustrate the utility of **DEoptim**, we fit the MSGARCH model (3) to daily log-returns of the Swiss Market Index (SMI), displayed in Figure 6. The sample period is from November 12, 1990, to October 20, 2000, for a total of 2500 observations and the log-returns are expressed in percent. The data set was downloaded from `http://finance.yahoo.com/` and is available when **DEoptim** is loaded using the command `data("SMI")`. Note that the two-regime specification is used for illustrative purposes only; checking for possible model misspecification is beyond the scope of the present paper.

In addition to the positivity constraints on the model parameters, we require covariance stationarity to hold in the two regimes, i.e., $(\alpha_i^+ + \alpha_i^-)/2 + \beta_i < 1$ for $i = 1, 2$ and that the unconditional variance (4) in state 1 is smaller than in state 2, i.e., $\bar{\sigma}_1^2 < \bar{\sigma}_2^2$. We also require the transition probabilities p_{11} and p_{22} of the state variable to lie within the $[0, 1]$ interval. The constraints on the domain are set using the arguments `lower` and `upper` of `DEoptim`, while the covariance stationarity and unconditional variance constraints are tested within the

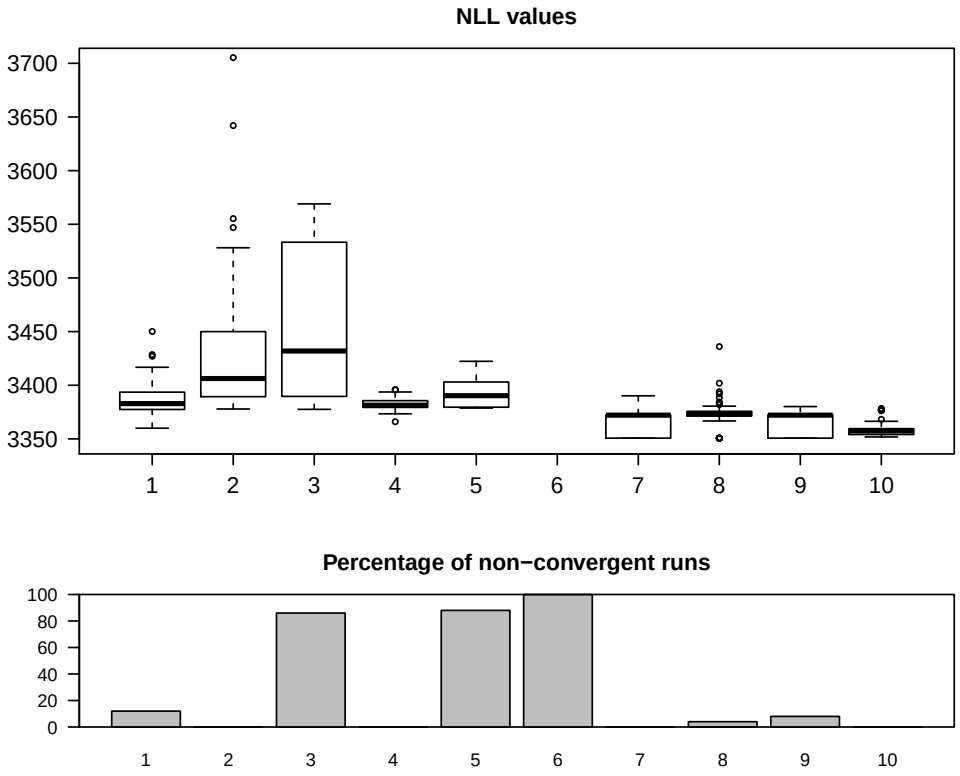

Figure 7: Top: Boxplots of the 50 negative values of the log-likelihood function (NLL) at optimum $\hat{\theta}$ obtained by the various optimizers. (1) function `optim` with method `"Nelder-Mead"`, (2) method `"BFGS"`, (3) method `"CG"`, (4) method `"L-BFGS-B"`, (5) method `"SANN"` with control parameter `itermax = 1e5`, (6) function `nlminb`, (7) function `constrOptim`, (8) function `constrOptim.nl` of the package **alabama**, (9) function `solnp` of the package **Rsolnp**, (10) function `DEoptim` with control parameters `NP = 110` and `itermax = 500`. Starting values are generated randomly in the feasible parameter set. Lower boundaries were set to 0.0 (2.0 for ν) and upper boundaries to 1.0 (50 for ν). Bottom: Percentage of non-convergent runs of the different optimization methods (either `NaN` output or convergence flag indicating non-convergence).

objective function which then returns a very large value (in our case `1e10`) if not satisfied. In the DE optimization, we set the `control` parameters of `DEoptim` to `itermax = 500` and `NP = 110`. Note that the objective function (6) is implemented in C to speed up the optimization procedure. We refer the reader to the Appendix for details on the R implementation.

For comparison, the objective function (6) is also optimized using various unconstrained and constrained optimization routines available in R. More specifically, we use the function `optim` with methods `"Nelder-Mead"` (unconstrained optimization), `"BFGS"` (unconstrained), `"CG"` (unconstrained), `"SANN"` (unconstrained), `"L-BFGS-B"` (box constrained), the function `nlminb` (box constrained). More details on the underlying algorithms for these methods can be found in the `optim` and `nlminb` manuals. We also consider optimization routines which handle more complicated constraints such as `constrOptim`, `constrOptim.nl` of the package

alabama (Varadhan 2010) and `solnp` of the package **Rsolnp** (Ghalanos and Theussl 2011). The former relies on an adaptive barrier algorithm and handles linear inequality constraints. The two latter belong to the class of indirect solvers and implement the augmented Lagrange multiplier methods, in which linear and non-linear equality and inequality constraints are allowed; see Ye (1987) for details. For all methods we use ten times the default values of the control parameters related to the maximum number of iterations and function evaluations. For `optim` with method `"SANN"` we set `itermax = 1e5`. For unconstrained methods, the constraints on the model parameters were tested within the function which returns `1e10` is not satisfied, as for `DEoptim`. For functions which handled inequality constraints, we implement the constraints explicitly in the inputs of the functions; see the Appendix for an example with `solnp`.

We run the estimation 50 times for all optimization routines (including `DEoptim`) and use random starting values in the feasible parameter set when needed (using the same random starting values for the various methods). Boxplots of the negative log-likelihood value (NLL) at optimum for convergent estimations is displayed in the upper part of Figure 7. The lower part reports the percentage of non-convergent optimizations (defined as `NaN` output or non-convergent flag, usually `convergence > 0`). We notice that the unconstrained and box constrained methods 1-6 perform poorly compared to the optimizers which can handle more complicated constraints. In particular, we note that `nlminb` does not converge over the 50 runs. Overall, the methods `solnp` and `constrOptim` are the best performers in terms of lowest NLL values reached over the 50 runs. However, we notice that for both the convergence is not achieved in about 20% of the time. `DEoptim` compares favorably with the two best competitors in terms of NLL and is more stable over the runs. It does not however reach the lowest value obtained by `solnp` after 500 iterations. The tradeoff for the stability afforded by `DEoptim` is in runtime; it requires considerably more CPU time than either `solnp` or `constrOptim` (though significantly less runtime than the other stochastic optimization algorithm tested, `optim` with method `"SANN"`).

Figure 8 displays the boxplots of the NLL values obtained over the 50 runs of `DEoptim` with control parameter `itermax = 500` on the left-hand side and `itermax = 1000` on the right-hand side. The horizontal red lines reports the lowest values obtained by `solnp`. We notice that increasing the number of generated population in `DEoptim` leads to the convergence toward the global optimum.

Figure 9 displays the boxplots of the parameter values obtained with the 50 runs of `DEoptim` together with the parameter values corresponding to its best run (in blue squares), i.e., the run leading to the minimum NLL, and the parameter values corresponding to the best run of `solnp` (in red dots). The parameters at the *global* optimum (NLL = 3350.6979) obtained by `solnp` and `DEoptim` (after a longer run with `itermax = 2500`) are $\hat{\omega}_1 = 0.2062$, $\hat{\omega}_2 = 0.0930$, $\hat{\alpha}_1^+ = 0.0000$, $\hat{\alpha}_2^+ = 0.0043$, $\hat{\alpha}_1^- = 0.2123$, $\hat{\alpha}_2^- = 0.1566$, $\hat{\beta}_1 = 0.5295$, $\hat{\beta}_2 = 0.8717$, $\hat{p}_{11} = 0.9981$, $\hat{p}_{22} = 0.9969$ and $\hat{\nu} = 9.2480$. The parameter estimates clearly indicate two different regimes for the conditional variance process. More precisely, the values of $\hat{\omega}_i$ and $\hat{\beta}_i$ are far apart between the regimes. We note the presence of leverage effect in both regimes (i.e., $\hat{\alpha}_i^+ < \hat{\alpha}_i^-$ for $i = 1, 2$), with similar levels. The estimated transition probabilities $\hat{p}_{11}$ and $\hat{p}_{22}$ very close to one indicate infrequent mixing between states. Finally, the estimated degrees of freedom parameter suggests heavy tails for the conditional distribution of the log-returns.

Finally, Figure 10 displays the estimated filtered probabilities of the second state (high unconditional volatility state), $\{\mathsf{P}(s_t = 2 \,|\, \hat{\theta}, I_t); t = 1, \ldots, T\}$, implied by the best model parameters

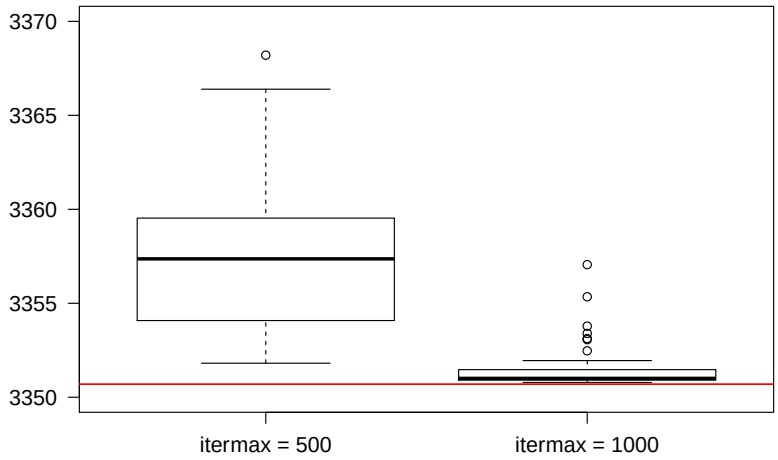

Figure 8: Top: Boxplots of the 50 negative values of the log-likelihood function (NLL) at optimum $\hat{\theta}$ obtained by `DEoptim` with control parameters `NP = 110` and `itermax = 500` (left) and `itermax = 1000` (right). The horizontal red line reports the lowest NLL value obtained by `solnp` over the 50 runs.

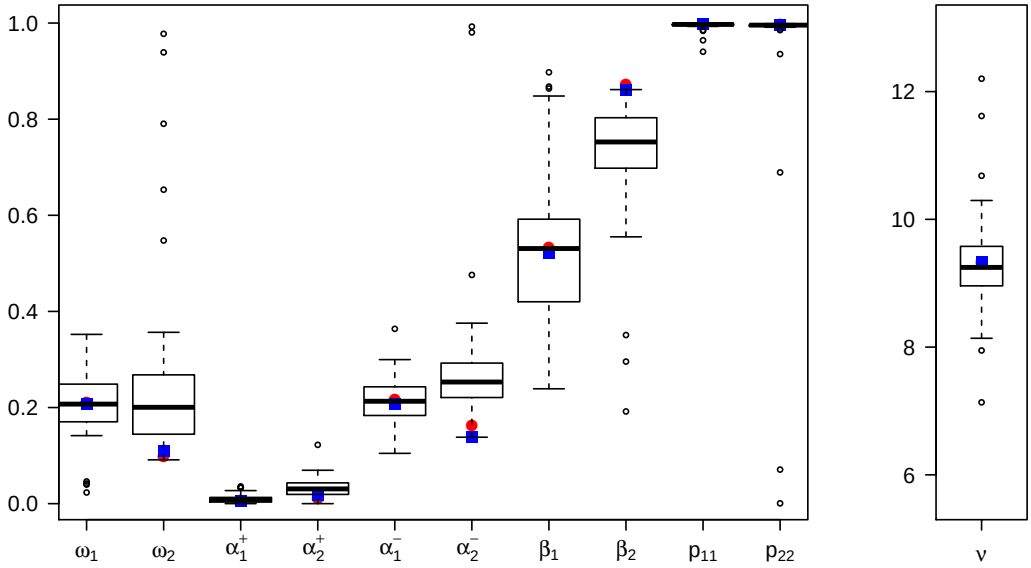

Figure 9: Boxplot of the parameters obtained over the 50 runs of `DEoptim`, together with the parameters of its best run (blue squares), i.e., the run leading to the lowest NLL, and the parameters corresponding to the best run of `solnp` (red dots).

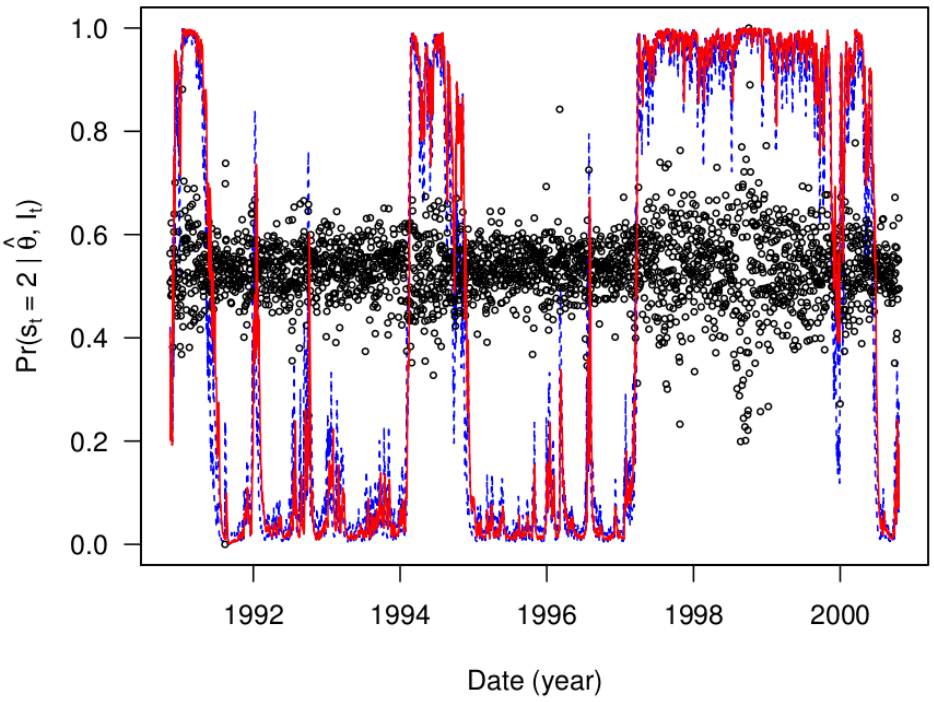

Figure 10: Estimated filtered probabilities of the second state, $\{\mathsf{P}(s_t = 2 \,|\, \hat{\theta}, I_t); t = 1, \ldots, T\}$, implied by the best model parameters obtained by `solnp` (in red) over the 50 runs. The dashed blue lines delimit the 50% area of the paths obtained over the 50 runs of `DEoptim`. The small black circles depict the log-returns.

of `solnp` (in red solid line) together with the log-returns (in small circles). In addition, we report in dashed blue lines, the 50% area of the paths obtained over the 50 runs of `DEoptim`. The parameters obtained with `DEoptim` and `solnp` lead to a clear separation of regimes in the filtering probabilities. The beginning of year 1991 is associated with the high unconditional volatility state. Then, from the second half of 1991 to 1997, the returns are clearly associated with the low unconditional volatility regime, with the exception of 1994. From 1997 to 2000, the model remains in the high unconditional volatility regime with a transition during the second semester 2000 to the low unconditional volatility state.

6. Summary and conclusions

Differential evolution is a heuristic evolutionary method for global optimization that is effective on many problems of interest in science and technology. By implementing the package **DEoptim** we have made this algorithm possible to easily apply in the R language and environment. As Section 3 details, we have also made available many variations on the classical DE strategy. These variations as well as the classical strategy are due to Price, Storn and Lampinen, and we have referred the interested reader to their textbook (Price *et al.* 2006) on

DE for details.

We have described herein the use of the package for fitting the Parratt recursion models for X-ray reflectometry and an MSGARCH model for the log-returns of the Swiss Market Index. These case studies showcase the power of the DE algorithm underlying **DEoptim**. We hope that readers will find the package to be a valuable tool for optimization. If you use R or **DEoptim**, please cite the software in publications.

Current work in extension of the package includes the addition of a framework for adaptive differential evolution (Zhang and Sanderson 2009). Future work will also be directed at parallelization of the implementation. The **DEoptim** project hosted on R-Forge (`http://R-Forge.R-project.org/projects/deoptim/`) links to development versions of the package.

Computational details

The results in this paper were obtained using R 2.10.0 (R Development Core Team 2009) with the package **DEoptim** version 2.0-4 (Ardia and Mullen 2010). Computations were performed on a Genuine Intel dual core CPU T2400 1.83Ghz processor and on a quad core Intel Xeon Processor E5410.

DEoptim relies on repeated evaluation of the objective function in order to move the population toward a global minimum. Users interested in making **DEoptim** run as fast as possible should ensure that evaluation of the objective function is as efficient as possible. Using pure R code, this may often be accomplished using vectorization. Writing parts of the objective function in a lower-level language like C or Fortran may also increase speed.

Acknowledgments

Many **DEoptim** users have sent us comments that helped improve the package. We would like to thank in particular Hans Werner Borchers, Kris Boudt, Eugene Demidenko, Tarmo Leinonen, Soren Macbeth, Dorothée Pages, Brian Peterson, Enrico Schumann, and Joshua Ulrich. We would also like to thank Juan David Ospina Arango, who inspired us to present the Rastrigin function as an example.

We thank the two reviewers of the paper for insightful suggestions.

Finally, we would like to thank Rainer Storn for his advocacy of DE and making his code publicly available, which was a great help to us in the implementation of **DEoptim**.

The first two authors (KMM and DA) contributed equally to testing and development, the remaining authors contributed expertise regarding the development of the XRR application described in Section 4 and the second author (DA) was the sole developer of the MSGARCH application described in Section 5.

Disclaimer

The views expressed in this paper are the sole responsibility of the authors and do not necessarily reflect those of NIST and aeris CAPITAL AG.

Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the

National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

Als-Nielsen J, McMorrow D (2001). *Elements of Modern X-Ray Physics*. John Wiley & Sons, Hoboken.

Ardia D (2008). *Financial Risk Management with Bayesian Estimation of GARCH Models: Theory and Applications*, volume 612 of *Lecture Notes in Economics and Mathematical Systems*. Springer-Verlag, Berlin. ISBN 978-3-540-78656-6.

Ardia D, Boudt K, Carl P, Mullen K, Peterson B (2010). "Differential Evolution (**DEoptim**) for Non-Convex Portfolio Optimization." *Technical report*, Social Science Research Network. URL http://ssrn.com/abstract=1584905.

Ardia D, Mullen K (2010). ***DEoptim**: Differential Evolution Optimization in R*. R package version 2.0-4, URL http://CRAN.R-project.org/package=DEoptim.

Björck M, Andersson G (2007). "**GenX**: An Extensible X-Ray Reflectivity Refinement Program Utilizing Differential Evolution." *Journal of Applied Crystallography*, **40**(6), 1174–1178.

Bollerslev T (1986). "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*, **31**(3), 307–327.

Bollerslev T, Chou RY, Kroner K (1992). "ARCH Modeling in Finance: A Review of the Theory and Empirical Evidence." *Journal of Econometrics*, **52**(1–2), 5–59.

Bollerslev T, Engle RF, Nelson DB (1994). "ARCH Models." In *Handbook of Econometrics*, chapter 49, pp. 2959–3038. North Holland.

Börner J, Higgins SI, Kantelhardt J, Scheiter S (2007). "Rainfall or Price Variability: What Determines Rangeland Management Decisions? A Simulation-Optimization Approach to South African Savanas." *Agricultural Economics*, **37**(2–3), 189–200.

Bowen DK, Tanner BK (2006). *X-Ray Metrology in Semiconductor Manufacturing*. CRC Press.

Cao R, Vilar JM, Devia A (2009). "Modelling Consumer Credit Risk via Survival Analysis." *Statistics & Operations Research Transactions*, **33**(1), 3–30.

Di Carlo W, Jarausch H (2006). ***DiffEvol**: Differential Evolution for Scilab*. Scilab code version 2006/03/01, URL http://www.icsi.berkeley.edu/~storn/DiffEvol.zip.

Dueker MJ (1997). "Markov Switching in GARCH Processes and Mean-Reverting Stock-Market Volatility." *Journal of Business & Economic Statistics*, **15**(1), 26–34.

Geldon H, Gauden PA (2006). ***DeMat** for Pascal*. Pascal translation of original MATLAB code, URL http://www.icsi.berkeley.edu/~storn/storn_pascal.zip.

Ghalanos A, Theussl S (2011). ***Rsolnp**: General Non-Linear Optimization*. R package version 1.0-8, URL http://CRAN.R-project.org/package=Rsolnp.

Godwin LE (1998). `DESolver`: *Differential Evolution Solver Class*. C++ code revision 1.0, URL http://www.icsi.berkeley.edu/~storn/devcpp.zip.

Hamilton JD (1989). "A New Approach to the Economic Analysis of Nonstationary Time Series and the Business Cycle." *Econometrica*, **57**(2), 357–384.

Hamilton JD (1994). *Time Series Analysis*. Princeton University Press, Princeton. ISBN 0691042896.

Higgins SI, Kantelhardt J, Scheiter S, Boerner J (2007). "Sustainable Management of Extensively Managed Savanna Rangelands." *Ecological Economics*, **62**(1), 102–114.

Holland JH (1975). *Adaptation in Natural Artificial Systems*. University of Michigan Press, Ann Arbor.

Klaassen F (2002). "Improving GARCH Volatility Forecasts with Regime-Switching GARCH." *Empirical Economics*, **27**(2), 363–394.

Lamoureux CG, Lastrapes WD (1990). "Persistence in Variance, Structural Change, and the GARCH Model." *Journal of Business & Economic Statistics*, **8**(2), 225–243.

Mitchell M (1998). *An Introduction to Genetic Algorithms*. The MIT Press.

Nash JC (1990). *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Adam Hilger.

Nelson A (2006). "Co-Refinement of Multiple-Contrast Neutron/X-Ray Reflectivity Data Using **MOTOFIT**." *Journal of Applied Crystallography*, **39**(2), 273–276.

Opsina Arango JD (2009). *Estimacion de un Modelo de Difusion con Saltos con Distribucion de Error Generalizada Asimetrica usando Algorithmos Evolutivos*. Master's thesis, Universidad Nacional de Colombia.

Parratt LG (1954). "Surface Studies of Solids by Total Reflection of X-Rays." *Physical Review*, **95**(2), 359–369.

Price KV, Storn RM, Lampinen JA (2006). *Differential Evolution: A Practical Approach to Global Optimization*. Springer-Verlag, Berlin.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Storn R (1999). ***DEApp** – An Application in Java for the Usage of Differential Evolution*. Java applet version 1.0.3, URL http://www.icsi.berkeley.edu/~storn/devol.zip.

Storn R (2004). ***DeWin**: Differential Evolution Engine*. C++ code version 4.0, URL http://www.icsi.berkeley.edu/~storn/DeWin.zip.

Storn R (2010). "Differential Evolution Homepage." International Computer Science Institute, University of California, Berkeley. URL http://www.icsi.berkeley.edu/~storn/code.html.

Storn R, Price K (1997). "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces." *Journal of Global Optimization*, **11**(4), 341–359.

Storn R, Price K, Neumaier A, Zandt JV (2004). ***DeMat**: Differential Evolution in MATLAB*. URL http://www.icsi.berkeley.edu/~storn/DeMat.zip.

Taylor M, Wall J, Loxley N, Wormington M, Lafford T (2001). "High Resolution X-Ray Diffraction Using a High Brilliance Source, with Rapid Data Analysis by Auto-Fitting." *Materials Science and Engineering B*, **80**(1-3), 95 – 98.

Varadhan R (2010). ***alabama**: Constrained Nonlinear Optimization*. R Package version 2010.10-1, URL http://CRAN.R-project.org/package=alabama.

Wang FS (2004). `DE_Fortran90`*: Differential Evolution for Optimal Control Problems*. Department of Chemical Engineering, National Chung Cheng University, URL http://www.icsi.berkeley.edu/~storn/DE_FORTRAN90.f90.

Wormington M, Panaccione C, Matney KM, Bowen DK (1999). "Characterization of Structures from X-Ray Scattering Data Using Genetic Algorithms." *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, **357**(1761), 2827–2848.

Ye Y (1987). *Interior Algorithms for Linear, Quadratic, and Linearly Constrained Non-Linear Programming*. Ph.D. thesis, Stanford University.

Zhang J, Sanderson AC (2009). "JADE: Adaptive Differential Evolution with Optional External Archive." *IEEE Transactions on Evolutionary Computation*, **13**(5), 945–958.

A. Implementation of the MSGARCH model

We described below the major steps for implementing and estimating in R the Markov-switching asymmetric GARCH(1, 1) model described in Section 5.

First, we define the function `NLL` which computes the negative value of the log-likelihood function (6). This function will be minimized in order to find the maximum likelihood estimator $\hat{\theta}$. The function `NLL` has input parameters `theta`, a 11-dim vector containing the MSGARCH(1, 1) parameters, `y` the $(T \times 1)$ vector of log-returns $y \doteq (y_1, \ldots, y_T)'$ and `checkConstraints`, a boolean indicating if the constraints should be checked within the function (which is `TRUE` by default). The function outputs `1e10` when the constraints are not fulfilled. In the implementation below, two constraints must be fulfilled: i) the covariance-stationarity condition of the conditional variance is satisfied in the two states; ii) the unconditional variance in state 1 is smaller than in state 2. The constraints on the lower and upper bounds on the parameters will be checked by the optimization function itself (below we consider `DEoptim` and `solnp` which handle this case). For optimization functions which cannot handle box constraints (e.g., `optim` with method `Nelder-Mead`), lower and upper bounds should also be checked within the function `NLL`. If the two constraints are satisfied, the function `NLL` calls a C routine which computes the negative log-likelihood value. The C implementation is needed for speed purposes, since for each input `theta`, two conditional variance processes need to be computed as well as the filtering and updating steps of a Kalman-like algorithm (Hamilton 1989). Since `DEoptim` evaluates the objective function for a large number of `theta` values, this C implementation is recommended. We do not document the C implementation here and refer the reader to file `MSGJR.c` attached for details.

```
R> 'NLL' <- function(theta, y, checkConstraints = TRUE) {
+    constraintsOK <- TRUE
+    nll <- 1e10
+    if (isTRUE(checkConstraints)) {
+      csc1 <- (theta[3] + theta[5]) / 2 + theta[7]
+      csc2 <- (theta[4] + theta[6]) / 2 + theta[8]
+      uv1  <- theta[1] / (1 - csc1)
+      uv2  <- theta[2] / (1 - csc2)
+      constraintsOK <- (csc1 < 1.0) & (csc2 < 1.0) & (uv1 < uv2)
+    }
+    if (isTRUE(constraintsOK)) {
+      n <- length(y)
+      outC <- .C(name = "MSGJRS", theta = as.double(theta),
+        y = as.double(y), n = as.integer(n), nll = as.double(0),
+        f = vector('double', 2 * (n-1)), u = vector('double', 2 * (n-1)))
+      nll <- outC$nll
+    }
+    if (is.nan(nll)) {
+      nll <- 1e10
+    }
+    return(nll)
+ }
```

Second, we load the package, the data set and the complied C function `MSGJR.c` (under Linux,

use `dyn.load("MSGJR.dll")` under Windows), set the lower and upper boundaries, set the seed and minimize the value of the function `NLL` using the function `DEoptim`. For the control parameters of `DEoptim`, we set `itermax = 500` and `NP = 110`, giving more robustness to the evolutionary process. The number of members in the population is set to 10 times the number of parameters. The result of the optimization is stored in the object `outDE` of the class `DEoptim`. Note that in this case, the function `NLL` checks if the constraints are fulfilled and outputs `1e10` if not.

```
R> library("DEoptim")
R> data("SMI")
R> dyn.load("MSGJR.so")
R> LB <- c(rep(0.0, 10), 2)
R> UB <- c(rep(1.0, 10), 50)
R> set.seed(1111)
R> outDE <- DEoptim(NLL, lower = LB, upper = UB, y = y,
+    DEoptim.control(itermax = 500, NP = 110))

Iteration: 1 bestvalit: 3457.906087 bestmemit: 0.310703 0.293937 0.608078
0.188130 0.227485 0.350169 0.145067 0.623558 0.405052 0.313200 8.941630
Iteration: 2 bestvalit: 3457.906087 bestmemit: 0.310703 0.293937 0.608078
0.188130 0.227485 0.350169 0.145067 0.623558 0.405052 0.313200 8.941630
...
Iteration: 500 bestvalit: 3354.081090 bestmemit: 0.225172 0.170456 0.007156
0.035552 0.247096 0.233424 0.504850 0.782076 0.996851 0.996322 9.472251
```

After 500 iterations, the best population member is

```
par1   par2   par3   par4   par5   par6   par7   par8   par9  par10  par11
0.2252 0.1705 0.0072 0.0356 0.2471 0.2334 0.5048 0.7821 0.9969 0.9963 9.4723
```

for a negative log-likelihood value of 3354.08.

An alternative optimization approach considered in Section 5 is `solnp` available in the package **Rsolnp** (Ghalanos and Theussl 2011). In this case the constraints are explicitely given using the input function `ineqfun` (which must have the same inputs than `NLL`, even if not all are needed). We therefore set the input `checkConstraints` to `FALSE` in `NLL`. Moreover, we require a starting value of the optimization. The result of the optimization is stored in the object `outSOLNP` of the class `Rsolnp`.

```
R> library("Rsolnp")
R> LB <- c(rep(0.0, 10), 2)
R> UB <- c(rep(1.0, 10), 50)
R> ineqfun <- function(theta, y, checkConstraints) {
+    csc1 <- 0.5 * (theta[3] + theta[5]) + theta[7]
+    csc2 <- 0.5 * (theta[4] + theta[6]) + theta[8]
+    uv1  <- theta[1] / (1 - csc1)
+    uv2  <- theta[2] / (1 - csc2)
+    uv   <- uv2 - uv1
```

```
+    h <- c(csc1, csc2, uv)
+    return(h)
+  }
R> ineqLB <- c(0, 0, 0)
R> ineqUB <- c(1, 1, 1e5)
R> thetaStart <- c(0.1, 0.2, 0.01, 0.01, 0.15, 0.15, 0.5, 0.6, 0.5, 0.5, 20)
R> outSOLNP <- solnp(pars = thetaStart, fun = NLL, ineqfun = ineqfun,
+    ineqLB = ineqLB, ineqUB = ineqUB, LB = LB, UB = UB, y = y,
+    checkConstraints = FALSE)

Iter: 1 fn: 3374.4122 Pars: 0.0975794 0.0001383 0.0315380 0.0262597
0.3136239 0.0505717 0.7515778 0.9615000 0.5577655 0.3701970 9.0604452
Iter: 2 fn: 3374.3081 Pars: 0.1008872 0.0001061 0.0310303 0.0264484
0.3099624 0.0534042 0.7483836 0.9600354 0.5494617 0.3508670 9.1424336
Iter: 3 fn: 3374.3080 Pars: 0.1009166 0.0001053 0.0310270 0.0264577
0.3098355 0.0534334 0.7483817 0.9600163 0.5493104 0.3503742 9.1394165
solnp--> Completed in 3 iterations
```

The optimum is obtained at

```
0.10092 0.00011 0.03103 0.02646 0.30984 0.05343 0.74838 0.96002 0.54931
0.35037 9.13942
```

for a negative log-likelihood value of 3374.31. We note that in this case, the procedure gets stuck to a local minimum since the value is higher than the one obtained by `DEoptim`. Using the alternative starting value `c(0.01, 0.02, 0.01, 0.01, 0.15, 0.15, 0.8, 0.8, 0.9, 0.9, 10)` leads to the global minimum at 3350.698. Several starting values must therefore be used to diminish the risk of convergence to local minima.

Affiliation:

Katharine Mullen
Ceramics Division, National Institute of Standards and Technology (NIST)
100 Bureau Drive, MS 8520
Gaithersburg, MD, 20899, United States of America
E-mail: Katharine.Mullen@nist.gov

David Ardia
aeris CAPITAL AG Switzerland
URL: http://perso.unifr.ch/david.ardia/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 40, Issue 6 *Submitted:* 2010-03-27
April 2011 *Accepted:* 2011-02-04