AdequacyModel: An R Package for Probability Distributions and General Purpose Optimization

Pedro R. D. Marinho [1*], Rodrigo B. Silva[1†], Marcelo Bourguignon[2§], Gauss M. Cordeiro [3‡], Saralees Nadarajah[4✦],

1 Department of Statistics, Federal University of Paraíba, João Pessoa, Paraíba, Brazil
2 Department of Statistics, Federal University of Rio Grande do Norte, Natal, Rio Grande do Norte, Brazil
3 Department of Statistics, Federal University of Pernambuco, Recife, Pernambuco, Brazil
4 School of Mathematics, University of Manchester, Manchester, United Kingdom

* pedro.rafael.marinho@gmail.com

Abstract

Several lifetime distributions have been played an important role to fit survival data. However, for some of these models, it is quite difficult to calculate the maximum likelihood estimators due to the evidence of flat regions in the search space, among other factors. It makes several well-known derivative-based optimization tools unsuitable for obtaining such estimates. To circumvent this problem, we introduce the **AdequacyModel** computational library version 2.0.0 for R statistical environment with two major contributions: a general optimization technique based on the Particle Swarm Optimization (PSO) method (with a minor modification of the original algorithm) and a set of statistical measures for assessment of the adequacy of the fitted model. This library is very useful for researchers in probability and statistics and has been cited in various papers in these areas. It serves as the basis for the **Newdistns** library (version 2.1) published in an impact journal in the area of computational statistics, see https://CRAN.R-project.org/package=Newdistns. It is also the basis of the **Wrapped** library (version 2.0) at https://CRAN.R-project.org/package=Wrapped. More recently, a third package, package **sglg**, makes use of the **AdequacyModel** library. Details regarding ssg can be obtained at https://CRAN.R-project.org/package=sglg. In addition, the proposed library has proved to be very useful for maximizing log-likelihood functions with complex search regions. We provide a greater control of the optimization process by introducing a stop criterion which is based on a minimum number of iterations and the variance of a given proportion of optimal values. We emphasize that the new library can be used not only in statistics but in physics and mathematics as proved in several examples throughout the paper.

1 Introduction

In survival analysis, practitioners are usually interested in choosing the distribution that provides the best fit from a broad class of candidate models. In this sense, lifetime distributions are continually evolving in parallel with computer-based tools, which allow for using more complex distributions with a larger number of parameters to better study sizable masses of data. The last two decades have been very prolific in generating new parametric models for lifetime data and several methods to generate new distributions can be found

in the literature. In addition to extending traditional models, the relevance of new distributions relies on the fact that some of them can provide better fits to real data sets. For a survey on the most important recent lifetime distributions, the readers are referred to [**?**] and [**?**].

The main concern about recent proposed models is that in several cases one can obtain different solutions from different initial values when optimizing the corresponding likelihood functions, thus indicating the presence of flat regions in the search space. The term "flat" is used here to indicate that the minimum modulus of a function in a region is (in some sense) of the same order as the maximum modulus. In this case, most derivative-based optimization tools usually encounter difficulties such as getting trapped in local minima, which makes such approaches unsuitable to obtain the corresponding maximum likelihood estimates (MLEs). This is not, however, an exclusive problem of recent lifetime models. Several univariate and multivariate functions present the same issue. To circumvent this problem, some optimization algorithms based on swarm intelligence have been proposed over the last decades. This class of methods have shown efficiency and robustness, although simple to implement. One of very popular swarm intelligence methods is the Particle Swarm Optimization (PSO) for finding optimized solutions. The PSO is a stochastic search method introduced by [**?**] based on simple social behavior exhibited by birds and insects and, due to its simplicity in implementation, it has gained great popularity in optimization. It also has high level of convergence and low computational cost if compared with other heuristic search methods. It traditionally uses a random sampling to find the optimums, but it is superior, if compared with derivative-based methods, when the information about localization of the minimum (or maximum) is poor, which is the case when we have functions with flat regions. Further details on the PSO method can be found in [**?**].

Some variants of the PSO algorithm have been studied in the literature in order to fit different types of problems. [**?**] proposed a mirror-extended Curvelet transform and PSO to solve the problem of speckle noise and low contrast in Synthetic Aperture Radar images. Since data mining demands fast and precise partitioning of large data sets, it usually comes with a wide range of attributes or features, which requires serious computational restrictions on the relevant clustering techniques. [**?**] presented an overview of PSO techniques for cluster analysis. The issue of choosing the most adequate values in the Support Vector Machine (SVM) methodology can be structured in terms of an optimization problem in order to minimize a prediction error. [**?**] introduced an integrated PSO algorithm (PSO + SVM) to solve this problem. [**?**] presented a PSO overview under a Bayesian perspective thus providing a formal framework for incorporation of prior knowledge about the problem that is being solved. [**?**] adopted maximum likelihood via PSO algorithm to estimate the mixture of two Weibull parameters with complete and multiple censored data. The main idea behind the proposed R package is to provide a set of tools for the assessment of the adequacy of lifetime models through a robust optimization method for determining the MLEs for lifetime distributions, in special those with approximately flat regions. Our contribution to the PSO algorithm consists to replace the particles that eventually fall outside the search region, which is a subtle variation of the original approach. By doing this, we expect to keep the initial variability of the algorithm and prevent all particles from converging to a local optimum. Further, we provide more control over some aspects of the algorithm, such as the number of particles and iterations and a conditional stop criterion, which is based on a minimum number of iterations and the variance of a given proportion of optimal values. However, rather than focusing in the PSO itself, we provide an easy-to-use set of statistical measures to assess the adequacy of lifetime models for a given dataset. In addition to the MLEs, the package provides some useful statistics to assess the goodness-of-fit of probabilistic models including Cramér-von Mises and Anderson-Darling statistics. These statistics are often used to compare non-nested models. The proposed

package also gives other goodness-of-fit measures such as the Akaike information criterion and Bayesian information criterion, as well as the some adherence tests, such as the Kolmogorov-Smirnov test, all this through the `goodness.fit()` function. Even all though the our focus lies in lifetime models, the proposed optimization package can be used in several other areas as proved in some examples throughout the paper.
This paper is organized as follows. Section 2 describes some theoretical background of swarm intelligence and general ideas underlying the PSO approach. Section 3 presents the PSO algorithm designed for the **AdequacyModel** in R package. Section 4 provides practical examples which show the effectiveness of our PSO algorithm compared to the results from other techniques, especially those based on derivatives. Section 4 contains an application using real (not simulated) data.In Section 5, a Monte Carlo simulation study is presented to verify the behavior of the optimizations obtained by the `pso()` function provided by the package. Finally, Section 6 gives some concluding remarks on the main findings of the paper and the current package usage.

2 Conceptual design of the framework

2.1 Swarm intelligence

Swarm intelligence is an exciting research field still in its infancy if compared to other paradigms in artificial intelligence. It is a branch of artificial intelligence concerned to the study of collective behavior of decentralized and self-organized systems in a social structure. These kinds of systems are composed by agents that interact in a small organization (swarm) wherein each individual is a particle. The main idea behind swarm intelligence is that an isolated particle has a very limited action in search an ideal point for the solution of an nondeterministic polynomial (NP) time complete problem. However, the joint behavior of the particles in the search region shows evidence of artificial intelligence, i.e., the ability to take decisions to respond to changes. In this sense, the swarm intelligence concept arises directly from nature and is based on, for example, the self-organizing exploratory pattern of the schools of fish, flocks of birds and ant colonies. This collective behavior can not be described simply by aggregating the behavior of each element. Such situations have encouraged practitioners to obtain a satisfactory effect in the search for solutions to complex problems by studying methods that promote intelligent behavior through collaboration and competition among individuals. Swarm-based algorithms have been widely developed in the last decade and many successful applications in a variety of complex problems make it a very promising, efficient and robust optimization tool, although very simple to implement. The idea is modeling very simple local interactions among individuals from which complex problem-solving behaviors arise.

2.2 Proposed PSO algorithm

The PSO algorithm is conceptually based on the social behavior of biological organisms that move in groups, such as birds and fishes. It has been provided good solutions for problems of global function optimization with box-constrained. The fundamental component of the PSO algorithm is a particle, which can move around in the search space in direction of an optimum by making use of its own information as well as that obtained from other particles within its neighborhood. The performance of a particle is affected by its fitness, which is evaluated by calculating the objective function of the problem to be solved. The particles movement in the search space is randomized. For each iteration of the PSO algorithm, the leader particle is set by minimizing the objective function in the corresponding iteration. The remaining particles arranged in the search region will randomly follow the leader particle and sweep the area around the leader particle. In this

local search process, another particle may become the new leader and the other particles will follow the new leader randomly.
Mathematically, a particle i is featured by three vectors, namely:

- Its current location in the n-dimensional search space denoted by $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{in})$.
- The best individual position it has held so far denoted by $\boldsymbol{p}_i = (p_{i1}, \ldots, p_{in})$.
- Its velocity $\boldsymbol{v}_i = (v_{i1}, \ldots, v_{in})$.

Usually, the current location $\boldsymbol{x}_i$ and velocity $\boldsymbol{v}_i$ are initialized by sampling from uniform distributions throughout the search space and setting a maximum velocity value $v_{\max}$.
Then, the particles move over the search space in sequential iterations driven by the following set of update equations:

- $v_{i,d}(t+1) = v_{i,d}(t) + c_1\, r_1\, [p_{i,d}(t) - x_{i,d}(t)] + c_2\, r_2\, [p_{g,d}(t) - x_{i,d}(t)]$;
- $x_{i,d}(t+1) = x_{i,d}(t) + v_{i,d}(t+1), \quad d = 1, \ldots, n,$

where c_1 and c_2 are constants, r_1 and r_2 are independent uniform random numbers generated at every update along each individual direction $d = 1, \ldots, n$ and $p_g(t)$ is the n-dimensional vector of the best position encountered by any neighbor of the particle i. The velocities and positions at time $t+1$ are influenced by the distances of the particle's current location from its individual best historical experience $p_i(t)$ and its neighborhoods best historical experience $p_g(t)$ in a cooperative way.
The proposed PSO algorithm is a small modification of the standard PSO algorithm pioneered by [**?**], where $f : \mathcal{R} \mapsto \mathbb{R}$, with $\mathcal{R} \subseteq \mathbb{R}^n$, is the objective function to be minimized, S is the number of particles of the swarm (set of feasible points), each particle having a location vector $x_i \in \mathcal{R}$ in the search space and a velocity vector defined by $v_i \in \mathcal{R}$. Let p_i be the best known position of the particle i and g the best position of all particles. The small modifications are highlighted in the algorithm below. The default optimization does not address the optimization problem restricted to a region $\mathcal{R}$. In the course of the iterations, it is common for several particles to fall outside the search region $\mathcal{R}$. The strategy of eliminating these particles and randomly relocating them in the search region increases the variability of the algorithm by preventing all particles from converging to a local minimum.
1. For each particle $i = 1, \ldots, S$ do:

- Initialize the particle's position with a uniformly distributed random vector: $x_i \sim U(b_{lo}, b_{up})$, where b_{lo} and b_{up} are the lower and upper boundaries of the search-space.
- Initialize the particle's best known position to its initial position: $p_i \leftarrowtail x_i$.
- If $f(p_i) < f(g)$ update the swarm's best known position: $g \leftarrowtail p_i$.
- Initialize the particle's velocity: $v_i \sim U(-|b_{up} - b_{lo}|, |b_{up} - b_{lo}|)$.

2. Until a termination criterion is met (e.g. number of iterations performed, or a solution with adequate objective function value is found), repeat:

- For each particle $i = 1, \ldots, S$ do:
  - Pick random numbers: $r_p, r_g \sim U(0, 1)$.
  - For each dimension $d = 1, \ldots, n$ do:
    - Update the particle's velocity: $v_{i,d} \leftarrowtail \omega\, v_{i,d} + \varphi_p r_p (p_{i,d} - x_{i,d}) + \varphi_g r_g (g_d - x_{i,d})$.
  - Update the particle's position: $x_i \leftarrowtail x_i + v_i$

- If $x_i \notin \mathcal{R}$
  - Eliminate x_i.
  - Generate new values $x_i \in \mathcal{R}$ (random values).
- If $f(x_i) < f(p_i)$ do:
  - Update the particle's best known position: $p_i \leftarrowtail x_i$
  - If $f(p_i) < f(g)$ update the swarm's best known position: $g \leftarrowtail p_i$.

3. Now g holds the best found solution.
The parameter ω is called inertia coefficient and, as the name implies, controls the inertia of each particle arranged in the search space. The quantities φ_p and φ_g control the acceleration of each particle and are called acceleration coefficients. The PSO algorithm described above implemented in R programming language is given in the next section. A conditional stopping criterion will be discussed.
The choices of constants ω, φ_p and φ_g can dramatically affect the performance of the algorithm in the optimization process. Discussions about appropriate parameter choices have been the subject of some researches, see [**?**] and [**?**].
One possible method is not assess the fitness of the particles outside the search region and expect that these particles return to the search region according some social interaction with other particles, as we can see in [**?**]. However, many problems involving likelihood-based inference require numerical constrained optimization. For example, the log-likelihood function is maximized subject to the constraint that the parameter of interest takes on the null-hypothesized value in the likelihood ratio test. In such problems, replacing the particles outside the feasible search region is a way to keep the initial variability of the algorithm.

3 The AdequacyModel package

3.1 Multi-parameter global optimization

The above algorithm is implemented in the **AdequacyModel** package available in R website. It is quite general and can be applied to maximize or minimize any objective function involving or not a database taking into account restriction vectors delimiting the search space. We want to make clear that the pso function is constructed to minimize an objective function. However, to maximize f is equivalent to minimize $-f$. A brief description of the **AdequacyModel** package is listed below:

- `func`: an objective function to be minimized;
- `S`: number of considered particles. By default, the number of particles is equal to 150;
- `lim_inf` e `lim_sup`: define the inferior and superior boundaries of the search space, respectively;
- `e`: current error. The algorithm stops if the variance in the last iterations is less than or equal to `e`;
- `data`: by default `data = NULL`. However, when the `func` is a log-likelihood function, `data` is a data vector;
- `N`: minimum number of iterations (default `N = 500`);

- `prop`: Proportion of last optimal values whose variance is calculated and used as a stop criterion. That is, if the number of iterations is greater than or equal to the minimum number of iterations `N`, then calculate the variance of the last optimal values, where $0 \leq$ `prop` ≤ 1.

One advantage of the PSO method is that we do not need to be concerned with initial values. Problems with initial values are frequent in iterative methods such as the BFGS when the objective function involves flat or nearly flat regions. We can obtain totally different results depending on the chosen initial values. This kind of issue is not usual in heuristic-based methods, where the updated steps include randomness (generation of pseudo-random numbers). The following example presents issues related to the initial guesses for the algorithm and shows the use of the `pso` function, especially the argument `func` to specify the objective function. In order to provide a greater control of the optimization process, we define N as the stop criterion that states the minimum number of iterations. The number of optimal values considered in the variance calculation is given by the proportion of optimal values stated by the argument prop, which is equal to 0.2 by default. In other words, if the 20% last optimal values show variance less than or equal to `e`, the algorithm will stop the global search, thus indicating convergence according to the fixed criteria. These stop criteria indicate that there is no significant improvements in the global search for this proportion of iterations. Thus, if the variance is less than or equal to $\varepsilon > 0$ assigned to the argument `e` of the `pso()` function, the algorithm will stop the iterations and return the best point that minimizes the objective function.

3.2 Examples

3.2.1 Trigonometric function

Initially, we consider the case of a global search in a univariate function to estimate a one-dimensional vector. Consider the objective function $f(\theta) = 6 + \theta^2 \sin(14\theta)$. This function has some local minima such that $\theta = 2.3605$ which globally minimizes $f(\theta)$ and $f(2.3605) = -11.5618$. In Figure 1, we plot $f(\theta)$ for $\theta \in [-2.5, 2.5]$. The blue square symbol indicates the global minimum obtained by the BFGS, SANN and Nelder-Mead methods. The red bullet in turn represents the global minimum obtained by the PSO method.

```
R> f <- function(x){
+     -(6 + x ^ 2 * sin(14 * x))
+  }
R> f_pso <- function(x, par){
+     theta <- par[1]
+     -(6 + theta ^ 2 * sin(14 * theta))
+  }
R> set.seed(9)
R> result_pso_f <- pso(func = f_pso, S = 500, lim_inf = -2.5,
+                       lim_sup = 2.5, e = 0.0001)
R> set.seed(9)
R> result_sann_f <- optim(par = 0, fn = f, lower = -2.5, upper = 2.5,
+                          method = "SANN")
R> result_bfgs_f <- optim(par = 0, fn = f, lower = -2.5, upper = 2.5,
+                          method = "BFGS")
R> result_nelder_f <- optim(par = 0, fn = f, lower = -2.5, upper = 2.5,
+                            method = "Nelder-Mead")
```

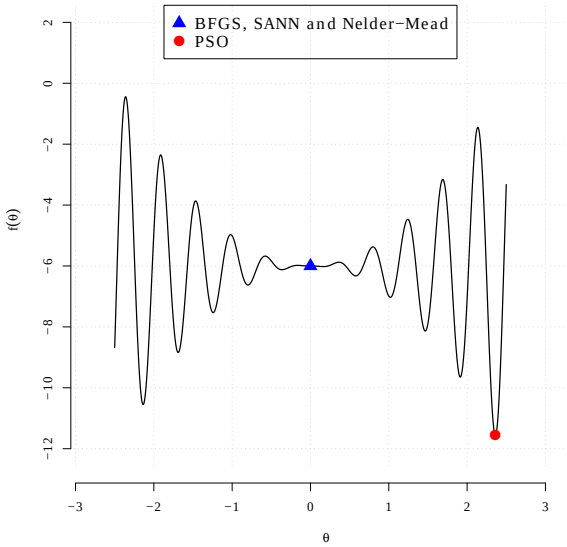

Fig 1. Function $f(\theta) = 6 + \theta^2 \sin(14\theta)$ with global minimum estimates.

Note that the global minimum estimates obtained by the BFGS, SANN and Nelder-Mead methods through the `optim()` function (for more details, execute `?optim`) are heavily influenced by initial values. It is quite clear from Figure 1 that there is a $\varepsilon > 0$ such that f has derivative close to 0 around $(-\varepsilon, \varepsilon)$. On the other hand, the `pso` function from the **AdequacyModel** script provides the true global minimum, which obviously coincides with the analytic solution. Note that all evaluated methods converge according to their associated stop criteria. For the BFGS, SANN and Nelder-Mead methods, we set the same initial value 0. For the SANN method and `pso` function, which involve randomization, we set a seed equal to 9, i.e. `set.seed(9)`. The global minimum values obtained by the BFGS, Nelder-Mead and SANN methods are identical and influenced by the starting values. Unlike these methodologies, the PSO method implemented by the `pso()` function does not require initial values. These results can be replicated using the **AdequacyModel** package and and the examples that follow. Note in the examples that there is no need for initial kicking information for optimizations through the `pso()` function.

3.2.2 Easom function

We now consider the Easom function $f(x, y) = -\cos(x)\cos(y)\exp\{-[(x-\pi)^2 + (y-\pi)^2]\}$ for $-10 \leq x, y \leq 10$. Some plots are displayed at different angles in Figures 2(a) and 2(b). The Easom function is minimized at $x = y = \pi$, and $f(\pi, \pi) = -1$. The `pso()` function to minimize $f(x, y)$ is

```
R> easom <- function(x, par){
+   x1 <- par[1]
+   x2 <- par[2]
+   -cos(x1) * cos(x2) * exp(-((x1 - pi) ^ 2 + (x2 - pi) ^ 2))
+ }
R> set.seed(9)
R> results_pso <- pso(func = easom, S = 500, lim_inf = c(-10, -10),
+                     lim_sup = c(10, 10), e = 0.0001)
```

Before execution of the `pso` function, we set `set.seed(9)`, for which the same results can be replicated. The estimated minimum points by the `pso` function are $\hat{x} = 3.139752$ and

$\hat{y} = 3.141564$, which are very close to $x = y = \pi$. The convergence of the algorithm for very close values to the global optimum can be noted in Easom level curves displayed in Figure 3.

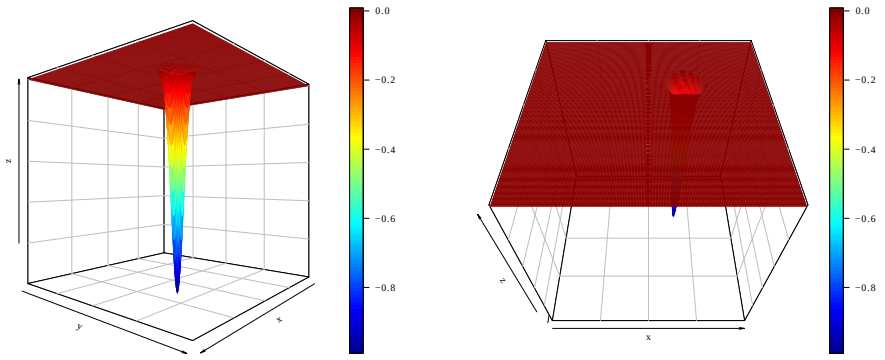

Fig 2. Easom function at two different angles.

We use the BFGS method through the `optim()` function and take as initial values $x = -9$ and $y = 9$. Note that the convergence is achieved in the BFGS method and the estimated minimum points coincide with the fixed initial values ($\hat{x} = -9$ and $\hat{y} = 9$), which is quite different from the minimum true point $x = y = \pi$, thus supporting that this method is very sensitive to initial values. The reader can observe this fact from the code below.

```
R> easom1 <- function(x){
+   x1 <- x[1]
+   x2 <- x[2]
+   -cos(x1) * cos(x2) * exp(-((x1 - pi) ^ 2 + (x2 - pi) ^ 2))
+}
R> result_bfgs_easom <- optim(par = c(9, 9), fn = easom1, method = "BFGS")
```

Notice that `result_bfgs_easom$convergence == 0` is equal to `TRUE`, which indicates convergence. Execute `help(optim)` for more details about the convergence criterion of the BFGS method implemented in the `optim` function. For the Easom function, the convergence is harmed by the existence of infinite candidates to the minimum point distributed on a flat region. The output stored in the object `result_bfgs_easom` is presented below:

```
R> result_bfgs_easom
$par
[1] -9  9

$value
[1] -1.283436e-30

$counts
function gradient
1        1

$convergence
[1] 0
```

```
$message
NULL
```

Setting `result_nelder_easom <- optim(par = c(-9, 9), fn = easom1, method = "Nelder-Mead")`, we also obtain a distant estimated point from the true global minimum point, where $\hat{x} = -8.1$ and $\hat{y} = 9$ give a minimum value approximately equal to zero. The results stored in `result_nelder_easom` are given below:

```
R> result_nelder_easom
$par
[1] -8.1  9.0

$value
[1] -3.609875e-71

$counts
function gradient
3       NA

$convergence
[1] 0

$message
NULL
```

A similar fact based on the simulated method where the estimates can be found with the script below:

```
R> set.seed(9)
R> result_sann_easom <- optim(par = c(-9, 9), fn = easom1,
+                             method = "SANN")
```

As in the previous cases, it is noted that `result_sann_easom$convergence == 0` is `TRUE` (there is convergence) and the estimated minimum point has coordinates distant from the coordinates of the true minimum point, where the estimated coordinates are $\hat{x} = 1.110688$ and $\hat{y} = 13.934928$ with the seed fixed at 9, i.e. `set.seed(9)`.

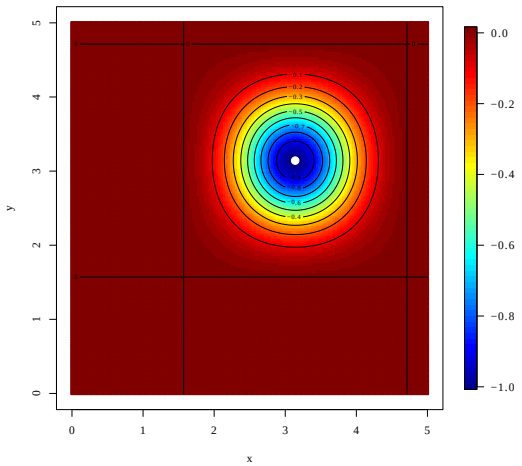

Fig 3. Curves of levels of the Easom function. The white point is the minimum value obtained by the `pso()` function.

3.2.3 Cross-in-tray function

Now, we use the `pso` function to minimize the Cross-in-tray function. This is a difficult function to be minimized for different reasons from those presented in the previous examples. The Cross-in-tray function has many local minima as they can be seen in Figures 4(a) and 4(b). This fact can certainly harm the convergence of various algorithms that search for a global optimum. The Cross-in-tray function is

$$f(x,y) = -0.0001 \left(\left| \sin(x)\sin(y)\exp\left(\left| 100 - \frac{\sqrt{x^2+y^2}}{\pi} \right| \right) \right| + 1 \right)^{0.1},$$

where $-10 \leq x, y \leq 10$ and

$$\text{Min} = \begin{cases} f(1.34941, -1.34941) &= -2.06261 \\ f(1.34941, 1.34941) &= -2.06261 \\ f(-1.34941, 1.34941) &= -2.06261 \\ f(-1.34941, -1.34941) &= -2.06261. \end{cases}$$

This function has four points of global minimum. Any estimates of the minimum points $(\hat{x}, \hat{y})$ that applied in $f(\cdot)$ present minimum value close to -2.0626 which is a good solution.

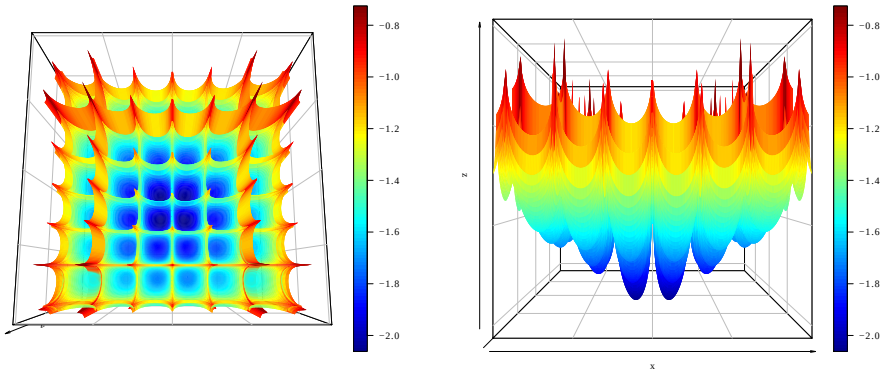

Fig 4. Cross-in-tray function at two different angles.

By means of the `optim` function, we note the convergence of the BFGS, SANN and Nelder-Mead methods with initial values at $x = 0$ and $y = 0$ and estimated values of x and y equal to $\hat{x} = \hat{y} = 0$ for the three approaches, and $f(\hat{x}, \hat{y}) = -0.0001$. The minimization of the Cross-in-tray function adopting the PSO algorithm achieves a satisfactory outcome as shown in Figure 5. The estimated minimum point is $(1.3490, 1.3490)$ yielding the minimum value $f(1.3490, 1.3490) = -2.0626$. These same results can be obtained with the script below:

```
R> cross <- function(x, par){
+    x1 <- par[1]
+    x2 <- par[2]
+    -0.0001 * (abs(sin(x1) * sin(x2) *
+                exp(abs(100 - sqrt(x1 ^ 2 + x2 ^ 2) / pi))) + 1) ^ 0.1
+ }
R> set.seed(9)
```

```
R> result_pso_cross <- pso(func = cross, S = 500, lim_inf = c(-10, -10),
+                           lim_sup = c(10, 10), e = 0.0001)
```

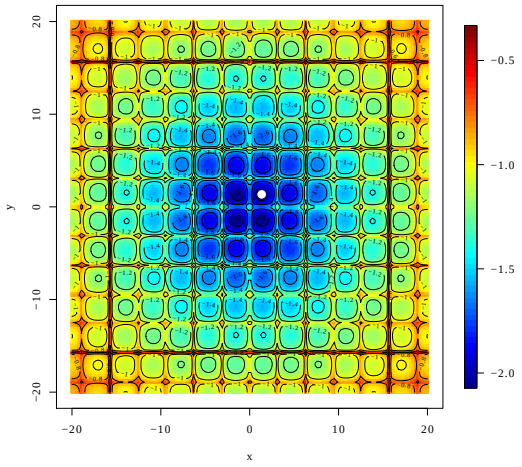

Fig 5. Curves of levels of Cross-in-tray function. The white point is the minimum value obtained by the `pso()` function.

Note: The results of the optimization using the `optim()` function and the Nelder-Mead, BFGS and simulated annealing methods can be determined from the code below such that, for all these methodologies, the initial shot is given at the point $(0, 0)$.

```
R> cross1 <- function(x){
+   x1 <- x[1]
+   x2 <- x[2]
+   -0.0001 * (abs(sin(x1) * sin(x2) *
+              exp(abs(100 - sqrt(x1 ^ 2 + x2 ^ 2) / pi))) + 1) ^ 0.1
+  }

R> result_bfgs_cross <- optim(par = c(0, 0), fn = cross1, lower = -10,
+                              upper = 10, method = "BFGS")

R> result_nelder_cross <- optim(par = c(0, 0), fn = cross1, lower = -10,
+                                upper = 10, method = "Nelder-Mead")

R> set.seed(9)
R> result_sann_cross <- optim(par = c(0, 0), fn = cross1, lower = -10,
+                              upper = 10, method = "SANN")
```

3.2.4 Hölder function

We consider the Hölder function, very peculiar and difficult to be optimized, defined by

$$f(x,y) = -\left|\sin(x)\cos(y)\exp\left(\left|1 - \frac{\sqrt{x^2+y^2}}{\pi}\right|\right)\right|,$$

where

$$\text{Min} = \begin{cases} f(8.05502,\ 9.66459) & = & -19.2085 \\ f(-8.05502,\ 9.66459) & = & -19.2085 \\ f(8.05502,\ -9.66459) & = & -19.2085 \\ f(-8.05502,\ -9.66459) & = & -19.2085, \end{cases}$$

and $-10 \leq x,\ y \leq 10$. Figure 6 displays the plots of the Hölder function defined above.

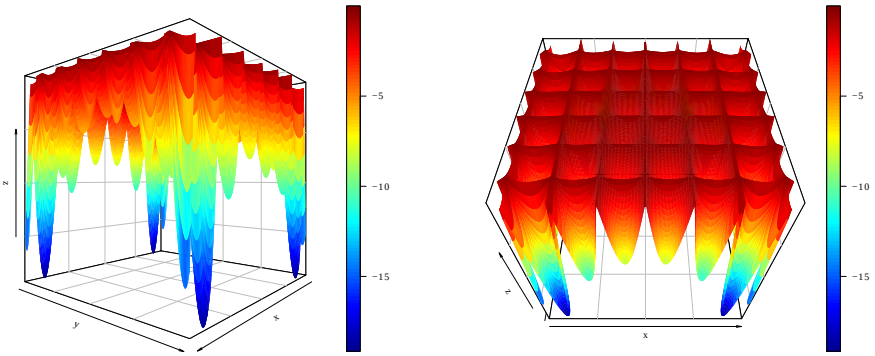

Fig 6. Hölder function at two different angles.

For the Hölder function, the results obtained from the BFGS, SANN and Nelder-Mead methods, as in the previous examples, are not good. However, in all cases, there is a convergence following these methodologies implemented in the `optim()` function. For initial values at the point $(0, 0)$, the convergence leads to this point, i.e., the three methodologies estimate the minimum point at $\hat{x} = 0$ and $\hat{y} = 0$. For the SANN method, we set `set.seed(9)`. However, the problem is easily circumvented by increasing the number of iterations. Figure 7 displays plots of the levels of the Hölder function with the point of convergence of the PSO algorithm. This result is determined using the following script:

```
R> holder <- function(x, par){
+     x1 <- par[1]
+     x2 <- par[2]
+     -abs(sin(x1) * cos(x2) * exp(abs(1 - sqrt(x1 ^ 2 + x2 ^ 2) / pi)))
+  }

R> set.seed(9)
R> result_pso_holder <- pso(func = holder, S = 500,
+                           lim_inf = c(-10, -10),
+                           lim_sup = c(10, 10), e = 0.0001)
```

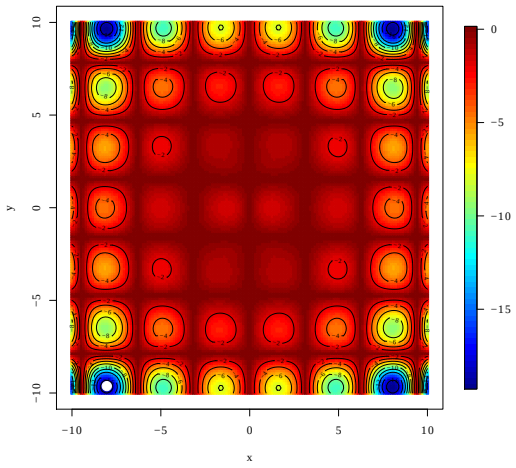

Fig 7. Curves of levels of Hölder function. The white point is the minimum value obtained by the `pso()` function.

4 Fitting distributions with the AdequacyModel

The problem of deciding on the suitability of an unknown cumulative distribution function (cdf) F_θ from a sample $x_1, \ldots, x_n$ is equivalent to the decision problem on an unknown parameter θ. Let $\mathcal{F} = \{F_\theta;\, \theta \in \Theta\}$ be a family of distributions, where Θ is the parameter space of θ. The best element F_θ in $\mathcal{F}$ can be determined from the MLE $\hat{\theta}_n$ of θ. Suppose that in $\mathcal{F}$ exists a F_θ for F evaluated at $\hat{\theta}_n$.

Some statistics are commonly used to verify the adequacy of the cdf F_θ to fit the observations. Alternatives to the likelihood ratio test were proposed by [**?**] by correcting the Carmér-von Mises (W^2) and Anderson-Darling (A^2) statistics. Let $F_n(x)$ be the empirical distribution function and $F(x;\widehat{\theta}_n)$ be the postulated cdf evaluated at $\widehat{\theta}_n$. According to [**?**], the usual Cramér-von Mises (W^2) and Anderson-Darling (A^2) statistics can be expressed as

$$W^2 = \sum_{i=1}^{n} [u_i - \{(2i-1)/(2n)\}]^2 + 1/(12n) \tag{1}$$

and

$$A^2 = -n - n^{-1} \sum_{i=1}^{n} \{(2i-1)\log(u_i) + (2n+1-2i)\log(1-u_i)\}, \tag{2}$$

where $u_i = \Phi((y_i - \overline{y})/s_y)$ (Φ is the standard normal cdf), $v_i = F(x_i;\widehat{\theta}_n)$, $y_i = \Phi^{-1}(v_i)$ and s_y is the sample standard deviation of the y_i's for $i = 1, \ldots, n$.

The corrected statistics W^* and A^* are given by

$$W^* = \left\{ n \int_{-\infty}^{+\infty} \{F_n(x) - F(x;\widehat{\theta}_n)\}^2 dF(x;\widehat{\theta}_n) \right\} \left(1 + \frac{0.5}{n}\right) = W^2 \left(1 + \frac{0.5}{n}\right), \tag{3}$$

$$\begin{aligned} A^* &= \left\{ n \int_{-\infty}^{+\infty} \frac{\{F_n(x) - F(x;\widehat{\theta}_n)\}^2}{\{F(x;\widehat{\theta})(1 - F(x;\widehat{\theta}_n))\}} dF(x;\widehat{\theta}_n) \right\} \left(1 + \frac{0.75}{n} + \frac{2.25}{n^2}\right) \\ &= A^2 \left(1 + \frac{0.75}{n} + \frac{2.25}{n^2}\right). \end{aligned} \tag{4}$$

The statistics W^* and A^* are measured by the difference between $F_n(x)$ and $F(x;\widehat{\theta}_n)$. Lower values of them provide further evidence that $F(x;\widehat{\theta}_n)$ generate the data. The null hypothesis tested using equations (3) and (4) is that the random sample has cdf $F(x;\theta)$. The algorithm below can be adopted to obtain W^* and A^*:

1. Estimate θ by $\widehat{\theta}_n$, order the observations in crescent values to calculate $v_i = F(x_i;\widehat{\theta}_n)$;
2. Calculate $y_i = \Phi^{-1}(v_i)$, where Φ^{-1} is the standard normal quantile function;
3. Calculate $u_i = \Phi\{(y_i - \overline{y})/s_y\}$, where $\overline{y} = n^{-1}\sum_{i=1}^{n} y_i$ and $s_y^2 = (n-1)^{-1}\sum_{i=1}^{n}(y_i-\overline{y})^2$;
4. Calculate W^2 e A^2 using equations (1) and (2), respectively;
5. Obtain $W^* = W^2(1 + 0.5/n)$ and $A^* = A^2(1 + 0.75/n + 2.25/n^2)$, where n is the sample size;
6. We reject $\mathcal{H}_0$ at the significance level α if the test statistics exceed the critical values presented by [**?**].

In practice, we can use W^* and A^* to compare two or more continuous distributions. The distribution that gives the lowest values of these statistics is the best suited to explain the random sample. The `goodness.fit()` function provides some useful statistics to assess the quality of fit of probabilistic models by including W^* and A^*. The function can also determine other measures such as the Akaike Information Criterion (AIC), Consistent Akaike Information Criterion (CAIC), Bayesian Information Criterion (BIC), Hannan-Quinn Information Criterion (HQIC) and Kolmogorov-Smirnov Test (KST). The general form for the function is given below with the descriptions of each one of its arguments:

```
goodness.fit(pdf, cdf, starts = NULL, data, method = "PSO",
             domain = c(0, Inf), mle = NULL)
```

where

- `pdf`: probability density function;
- `cdf`: cumulative distribution function;
- `starts`: initial parameters to maximize the likelihood function;
- `data`: data vector;
- `method`: method used for minimization of the -log-likelihood function. The methods supported are: PSO (default), BFGS, Nelder-Mead, SANN, CG (conjugate gradient). We can also provide only the first letter of the methodology, i.e., P, B, N, S or C, respectively;
- `domain`: domain of the pdf. By default the domain of the pdf is the open interval $(0, \infty)$. This option must be a vector with two components;
- `mle`: vector with the MLEs. This option should be used if one already has knowledge of the MLEs. The default is NULL, i.e., user the function will try to obtain the MLEs;
- `...`: If `method = "PSO"`, then all arguments of the `pso()` function could be passed to the `goodness.fit()` function.

It is not necessary to define the likelihood function or log-likelihood but only the pdf and cdf. The function will self-criticism to the arguments passed to the `goodness.fit()`. For example, if the supplied functions to the arguments `pdf` or `cdf` are not genuine pdfs and cdfs, a message will be given so that the user can check the arguments passed. We provide below two examples of the use of the `goodness.fit()` function.

4.1 Carbon fiber data

Consider a data set of stress (until fracture) of carbon fibres (in Gba). The data can be obtained by [**?**]. The data and some details can be accessed with the command `data(carbone)` in the **AdequacyModel** package. Suppose also that we are interested in choosing the best model in $\mathcal{F} = \{F_\theta; \theta \in \Theta\}$ that can represent the distribution of $X_1, \ldots, X_n$, whose observations are in `carbone`. Here, we consider that $\mathcal{F}$ is the exponentiated Weibull (Exp-Weibull) distribution, whose cdf is

$$F(x; \alpha, \beta, a) = \left\{1 - \exp\left[-(\alpha x)^\beta\right]\right\}^a, \ x > 0,$$

where α, β and a are positive parameters. Thus, each element in $\mathcal{F}$ is of the form $F(x; \alpha, \beta, a)$. We initially implement the density $f(x; \alpha, \beta, a)$ and cdf $F(x; \alpha, \beta, a)$. They will serve as arguments for the `pdf` and `cdf`, respectively. We present below the implementation of these functions that will be given to the `goodness.fit()` function.

```
R> # Probability density function.
R> pdf_expweibull <- function(par, x) {
+     alpha <- par[1]
+     beta <- par[2]
+     a <- par[3]
+     alpha * beta * a * exp(-(alpha * x) ^ beta) * (alpha * x) ^ (beta
+     - 1) * (1 - exp(-(alpha * x) ^ beta)) ^ (a - 1)
+ }

R> # Cumulative distribution function.
R> cdf_expweibull <- function(par, x) {
+     alpha <- par[1]
+     beta <- par[2]
+     a <- par[3]
+     (1 - exp(-(alpha * x) ^ beta)) ^ a
+ }

R> data(carbone)
R> results <- goodness.fit(pdf = pdf_expweibull, cdf = cdf_expweibull,
+                           starts = c(1, 1, 1), data = carbone,
+                           method = "BFGS", domain = c(0, Inf),
+                           mle = NULL)
```

The object `results` feature all goodness-of-fit statistics cited previously as well as the MLEs in case of `mle = NULL` (default). The standard errors of the MLEs if the argument `method` receives `PSO`, `BFGS`, `Nelder-Mead`, `SANN` and `CG`. Thus,

- `R> results$W` provides the statistic W^*;
- `R> results$A` provides the statistic A^*;
- `R> results$KS` provides the Kolmogorov-Smirnov statistic;
- `R> results$mle` provides a vector with the MLEs of the model parameters given as arguments for the `pdf`;
- `R> results$AIC`: provides the AIC statistic;
- `R> results$CAIC`: provides the CAIC statistic;
- `R> results$BIC`: provides the BIC statistic;

- `R> results$HQIC`: provides the HQIC statistic;
- `R> result$KS`: returns an object of class `htest` with information on the Kolmogorov-Smirnov test;
- `R> results$Erro`: provides the standard errors of the MLEs of the parameters, which index the model parameters given as arguments for the `pdf` and `cdf`;
- `R> results$value`: displays the minimum value of the function `-log(likelihood)`;
- `R> result$Convergence`: provides information on the convergence of the method passed as an argument for `method`. If `result$Convergence` $== 0$ for `TRUE`, there was convergence.

In case of the `method = "PSO"` (default), the standard errors will not be provided. The researcher may obtain these standard errors using bootstrap, see [**?**]. We provide below the results stored in the object `results` (output of the `goodness.fit()` function) and a plot with the fitted Exp-Weibull density.

```
R> results
$W
[1] 0.07047089

$A
[1] 0.4133608

$KS

One-sample Kolmogorov-Smirnov test

data:  data
D = 0.064568, p-value = 0.7987
alternative hypothesis: two-sided

$mle
[1] 0.3731249 2.4058010 1.3198053

$AIC
[1] 288.6641

$CAIC
[1] 288.9141

$BIC
[1] 296.4796

$HQIC
[1] 291.8272

$Erro
[1] 0.06265212 0.60467076 0.59835491

$Value
[1] 141.332
```

```
$Convergence
[1] 0
```

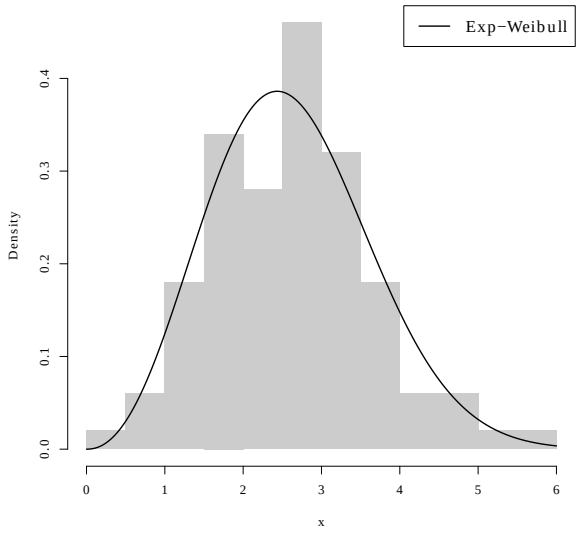

Fig 8. Fitted Exp-Weibull density to stress data (until fracture) of carbon fibers in Gba.

Notes: (*i*) The Kolmogorov-Smirnov statistic may return `NA` with a certain frequency which informs that this statistic is not reliable for the current data. More details about this issue can be obtained with `help(ks.test)`. In situations where `results$Convergence==0` is `TRUE`, there is convergence for the method passed as an argument to the `method` that minimizes the log-likelihood function multiplied by -1, that is, it minimizes `-log(likelihood)`. (*ii*) The convergence criterion as well as other details about possible values returned by `results$Convergence` can be obtained with `help(optim)` if the argument `method` of the `goodness.fit()` function receives the strings `"BFGS"`, `"Nelder-Mead"`, `"SANN"` or `"CG"` (or such those initial letters `"B"`, `"N"`, `"S"` or `"C"`). For the PSO methodology of minimization of the -log(likelihood) function (default `method = "PSO"`), the convergence criterion is displayed as discussed in Section 2, which normally is satisfied. (*iii*) The script for Figure 8 is:

```
R> pdf(file = "plot_adjustment.pdf", width = 9, height = 9, paper = "special",
+      family = "Bookman", pointsize = 14)

x = seq(0, 6, length.out = 250)

hist(carbone, probability = TRUE, xlab = "x", main = "")

lines(x, pdf_expweibull(par = results$mle, x), lwd = 2)

legend("topright", legend = c(expression(paste("Exp-Weibull"))), lwd = c(2.5),
+      inset = 0.03, lty = c(1), cex = 1.1, col = c("black"))

dev.off()
```

4.2 Flood level data

As a second example, we shall analyse a data set from [?] which refers to 20 observations of the maximum ood level (in millions of cubic feet per second) for Susquehanna River at Harrisburg, Pennsylvania. The data are: 0.26, 0.27, 0.30, 0.32, 0.32, 0.34, 0.38, 0.38, 0.39, 0.40, 0.41, 0.42, 0.42, 0.42, 0.45, 0.48, 0.49, 0.61, 0.65, 0.74. These data are fitted by using the Kumaraswamy beta (Kw-beta) distribution. Obviously, due to the genesis of the Kw-beta distribution, the flood level is by excellence ideally modelled by this distribution. Thus, the use of the Kw-beta distribution for fitting this data set is well justified.
A random variable X follows a Kw-beta distribution with shape parameters $a, b, \alpha, \beta > 0$, if its cdf and pdf are given by

$$F(x;\alpha,\beta,a,b) = 1 - \{1 - G(x;\alpha,\beta)^a\}^b$$

and

$$f(x;\alpha,\beta,a,b) = a\,b\,g(x;\alpha,\beta)G(x;\alpha,\beta)^{a-1}\{1 - G(x;\alpha,\beta)^a\}^{b-1},$$

whit $G(x;\alpha,\beta) = \mathrm{I}_x(\alpha,\beta)$ and $g(x;\alpha,\beta) = x^{\alpha-1}(1-x)^{b-1}/\,\mathrm{B}(a,b)$, $\mathrm{I}_x(a,b)$ is the incomplete beta function ratio $\mathrm{I}_y(a,b) = \frac{1}{\mathrm{B}(a,b)}\int_0^y \omega^{a-1}(1-\omega)^{b-1}\mathrm{d}\omega$ and $\mathrm{B}(\cdot,\cdot)$ denotes the beta function. We present below the implementation of the functions that will be given to the `goodness.fit()` function.

```
R> # Kumaraswamy Beta - Probability density function.
R> pdf_kwbeta <- function(par, x){
+     beta <- par[1]
+     a <- par[2]
+     alpha <- par[3]
+     b <- par[4]
+     (a * b * x ^ (alpha - 1) * (1 - x) ^ (beta- 1 )*
+     (pbeta(x,alpha,beta)) ^ (a - 1) *
+     (1 - pbeta(x, alpha, beta) ^ a) ^ (b - 1)) / beta(alpha, beta)
+ }
R>
R> # Kumaraswamy Beta - Cumulative distribution function.
R> cdf_kwbeta <- function(par, x){
+     beta <- par[1]
+     a <- par[2]
+     alpha <- par[3]
+     b <- par[4]
+     1 - (1 - pbeta(x, alpha, beta) ^ a) ^ b
+ }
R>
R> # Data set
R> data_unit <- c(0.26, 0.27, 0.30, 0.32, 0.32, 0.34, 0.38, 0.38, 0.39,
+                 0.40, 0.41, 0.42, 0.42, 0.42, 0.45, 0.48, 0.49, 0.61,
+                 0.65, 0.74)
R>
R> results <- goodness.fit(pdf = pdf_kwbeta, cdf = cdf_kwbeta,
+                           starts = c(1, 1, 1, 1), data = data_unit,
+                           method = "BFGS", domain=c(0,1),
+                           lim_inf = c(0, 0, 0, 0),
+                           lim_sup = c(10, 10, 10, 10), S = 200,
+                           prop = 0.1, N = 40)
```

```
R> results
$`W`
[1] 0.06228039

$A
[1] 0.3483813

$KS

        One-sample Kolmogorov-Smirnov test

data:  data
D = 0.14992, p-value = 0.7596
alternative hypothesis: two-sided


$mle
[1] 28.3805432 29.0062276  5.2899143  0.1774844

$AIC
[1] -24.71882

$`CAIC `
[1] -22.05215

$BIC
[1] -20.73589

$HQIC
[1] -23.94131

$Erro
[1]  1.93409776 30.74704316  1.92556208  0.04377468

$Value
[1] -16.35941

$Convergence
[1] 0
```

The estimates of the parameters are $(\widehat{a}, \widehat{b}, \widehat{\alpha}, \widehat{\beta}) = (29.0062, 0.1775, 5.2899, 28.3805)$, and the standard errors for the estimates of the parameters $\widehat{a}, \widehat{b}, \widehat{\alpha}$ and $\widehat{\beta}$ are, respectively, 30.747, 0.0438, 1.9256 and 1.9341.

4.3 TTT plot

Several aspects of an absolutely continuous distribution can be seen more clearly from the hazard rate function (hrf) than from either the cdf and pdf. The hrf is an important quantity characterizing life phenomena. Let X be a random variable with pdf $f(x)$ and cdf $F(x)$. The hrf of X is defined by

$$h(x) = \frac{f(x)}{1 - F(x)},$$

where $1 - F(x)$ is the survival function.
The hrf may be increase, decrease, constant, upside-down bathtub, bathtub-shaped or indicate a more complicated process. In many applications there is a qualitative information about the hazard rate shape, which can help in selecting a specified model. In this context, a device called the *total time on test* (TTT) or its scaled TTT transform proposed by [**?**] may be used for obtaining the empirical behavior of the hrf. The scaled TTT transform if defined by $(0 < u < 1)$

$$\phi_X(u) = \frac{H_X^{-1}(u)}{H_X^{-1}(1)},$$

where $H_X^{-1}(u) = \int_0^{Q(u)}[1 - F(x)]dx$ and $Q(u)$ is the quantile function of X. The quantity $\phi_X(\cdot)$ can be empirically approximated by

$$T(i/n) = \frac{\sum_{k=1}^{i} X_{k:n} + (n-i)X_{i:n}}{\sum_{k=1}^{n} X_k},$$

where $i = 1, \ldots, n$ and $X_{k:n}$, $k = 1, \ldots, n$, are the order statistics of the sample. Thus, the TTT plot is obtained by plotting $T(i/n)$ against i/n. We can detect the type of the hazard rate of the data. It is a straight diagonal for constant failure rates, it is convex for decreasing failure rates and concave for increasing failure rates. It is first convex and then concave if the failure rate is bathtub-shaped. It is first concave and then convex if the failure rate is upside-down bathtub. For more details, see [**?**].
The computation of the TTT plot is addressed in the **AdequacyModel** package. The real data set named `carbone` is used to illustrate the TTT plot function of this package. It refers to breaking stress of carbon fibres (in Gba) from [**?**]. The `TTT()` function developed to obtain the TTT curve follows the instructions:

```
R> library(AdequacyModel)
R> data(carbone)
R> TTT(carbone, col = "red", lwd = 2.5, grid = TRUE, lty = 2)
```

The TTT plot for the carbone data [**?**] is displayed in Figure 9, which reveals an increasing hrf. This plot reveals that distributions with increasing hrf could be good candidates for modeling the carbone data, see the theoretical plot in Figure 1 in [**?**].

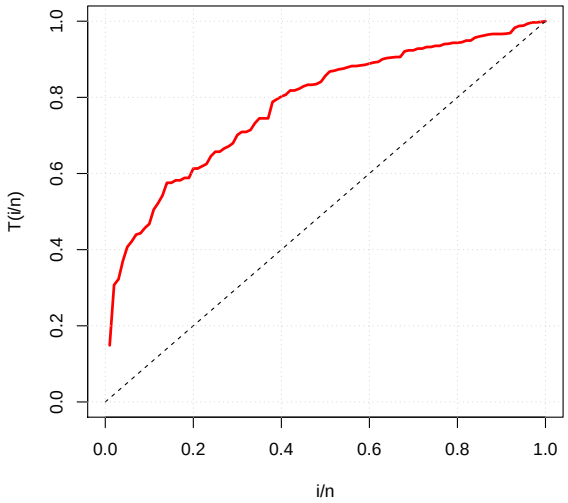

Fig 9. TTT-plot for carbon data.

5 Simulations

Visando estudar a consistência do método PSO, implementado na função `pso()` do pacote **AdequacyModel**, foram realizadas duas simulações de Monte Carlo (MC), considerando as funções Rastrigin e Himmelblau's, respectivamente, em que todos os resultados poderão ser reproduzidos com o código no Anexo A. As funções são bastante peculiares, sendo ambas multimodais. Funções multimodais pertence à classe de funções que impõe grandes desafios à métodos de otimização.
A função Rastringin considerada é definida por $5.12 \leq x_i \leq 5.12, \forall\, x_i \in \boldsymbol{x}$, tal que

$$f(\boldsymbol{x}) = An + \sum_{i=1}^{n} [x_i^2 - A\cos(2\pi x_i)], \tag{5}$$

em que foi considerado $A = 10$. Além disso, considerou-se $n = 2$, para que fosse possível a construção de um gráfico, em três dimensões. Essa função possui o valor mínimo $f(0, 0) = 0$. Por sua vez, a função de Himmelblau's é definida em $-5 \leq x, y \leq 5$, tal que

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2. \tag{6}$$

Essa função apresenta quatro pontos de mínimos globais, com 0 (zero) sendo o valor mínimo global. Os quatro pontos de mínimos globais são:

$$\text{Min} = \begin{cases} f(3.0, 2.0) &= 0.0 \\ f(-2.805118, 3.131312) &= 0.0 \\ f(-3.779310, -3.283186) &= 0.0 \\ f(3.584428, -1.848126) &= 0.0 \end{cases}$$

As peculiaridades das superfícies das funções de Rastringin e Himmelblau's podem ser observados nas Figuras 10(a) e 10(b), respectivamente. A Figura 10(a) exemplifica situações em que temos uma função objetivo com múltiplos mínimos locais e que poderá confundir muitos algoritmos de otimização. Já a Figura 10(b) tenta submeter a função `pso()` à um cenário com quatro ótimos globais que possuem regiões de descidas menos acentuadas.

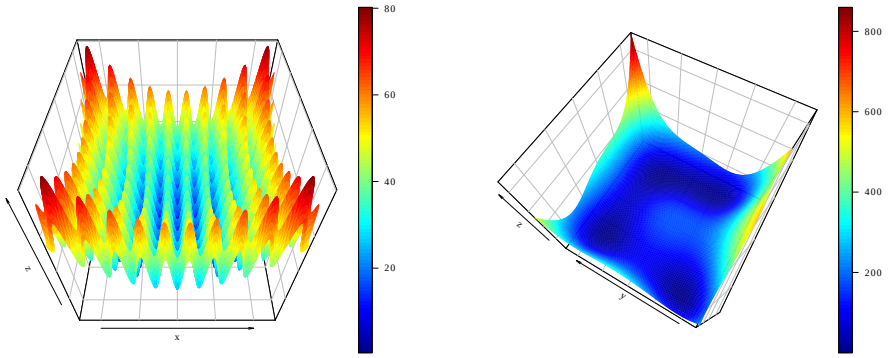

Fig 10. Superfícies das funções Rastringin e Hemmelblau's, respectivamente, utilizadas para avaliação da função `pso()` do pacote **AdequacyModel**.

É importante deixar claro que toda metaheurística tentará, se bem construída e implementada, fornecer uma boa solução para um problema. Dessa forma, é impossível sempre garantir a melhor solução. Sendo assim, ficaremos satisfeitos na obtenção de soluções razoáveis.

Aqui, tentaremos mostrar que a função `pso()` poderá ter grande utilidade, por exemplo, na detecção de padrões de vales (cavidades) em superfícies, uma vez que o comportamento das partículas do método `pso()`, implementado no pacote **AdequacyModel**, dificulta a atração de todas as partículas para um mesmo vale, ou seja, para uma mesma solução candidata de mínimo global. Isso, muito provavelmente deve-se à reposição aleatória de partículas que saem da região de busca, dando assim uma maior variabilidade, sem retirar a precisão do método.

Visando reduzir bastante o tempo das simulações, o código foi implementado para fazer uso de paralelismo de memória compartilhada (multicore), utilizando o pacote **parallel** disponível em qualquer instalação de R. Nessa classe de paralelismo, temos que os núcleos estão distribuídos em um mesmo chip. Dessa forma, mesmo que seja executado em um computador com mais de um processador, apenas os núcleos do processador que estará executando o ambiente do R será considerado. Isso é suficiente para as simulações em questão e facilitará a checagem dos resultados obtidos nessa seção, uma vez que reduzirá significativamente o tempo da execução de ambas as simulações quando realizadas em processadores multicores que são comuns nos dias atuais.

Como as iterações de MC são matematicamente independentes, a ideia é escrever os laços das simulações de MC por meio do funcional `mclaplly()`, do pacote **parallel**. O funcional **mclapply()** é bastante semelhante ao `lapply()` do pacote **base**, em que o prefixo "`mc`" no nome do funcional faz referência ao termo "multicore" . Ao contrário do `lapply()`, o funcional `mcapply()` irá disparar, simultaneamente, cada iteração do loop de MC (thread) sobre cada um dos cores de um mesmo processador. Em linguagens de programação interpretadas, como é o caso da linguagem R, substituir a tradicional estrutura de repetição `for` por um funcional poderá melhorar a eficiência computacional do código. O uso de funcionais para repetição de código é algo comum em linguagens de programação com paradigma funcional, que também é um dos paradigmas disponíveis na linguagem R.

As simulações de MC foram realizadas em um computador com processador Intel(R) Core(TM) i7-4710MQ trabalhando entre 2.50 GHz (frequência mínima) à 3.50 GHz (frequência máxima), 6 MB de cache, 8 threads e memória RAM de 32 GB DDR3. Para cada uma das funções, foram consideradas 20 mil simulações de MC. Por tratar-se de uma metodologia com passos aleatórios, foi fixado uma semente, `set.seed(1L)`, para que os resultados sejam reproduzíveis.

Para facilitar ainda mais a reprodução das simulações para as funções objetivo consideradas, as simulações MC podem ser reproduzidas ao chamar a função `simulation_mc()` que possui oito argumentos. São eles:

- `mc`: número de simulações de MC, com padrão em `mc = 20e3L`;
- `FUN`: poderá assumir as strings `"rastrigin"` (padrão) ou `"himmelblaus"` e refere-se à função objetivo considerada;
- `seed`: semente a ser considerada, em que o padrão é `seed = 1L`;
- `plot.curve`: se TRUE (padrão) irá construir o gráfico com curvas de níveis e sobrepor os valores ótimos estimados em cada iteração de MC (pontos brancos);
- `S`: quantidade de partículas consideradas, em que `S = 150` é o padrão;
- `e`: variância mínima admitida dos valores ótimos das últimas iterações, com `e = 1e-4` por padrão;
- `N`: número mínimo de iterações do método PSO, com `N = 50` definido por padrão;
- `prop`: proporção dos últimos valores de mínimo que será considerada para o calculado da variância utilizada como critério de parada.

Com o código em anexo (see A), as simulações de MC para a Rastrigin (Eq. 5) e função de Himmelblau's (Eq. 6) poderão ser reproduzidas fazendo `simulation_mc(mc = 2e4, FUN = "rastrigin")` e `simulation_mc(mc = 2e4, FUN = "himmelblaus")`, respectivamente. A simulação de MC que submeteu a função `pso()` para otimização da função de Rastringin levou aproximadamente 3.77 horas com os hardwares considerados. Um tempo muito inferior foi necessário para a realização das simulações de MC para a função de Himmelblau's (Eq. 6) devido a sua simplicidade em comparação com a função Rastrigin (Eq. 5), levando aproximadamente 4.58 minutos.

A primeira simulação de MC considerou a função de Rastringin (Eq. 5). Os resultados de cada iteração de MC estão representados como pontos brancos no gráfico de curvas de níveis da superfície, como mostra a Figura 11(a). De forma análoga, cada uma das 20 mil simulações de MC que submeteu a função `pso()` ao processo de otimização da função de Himmelblau's (Eq.6) estão apresentadas no gráfico das curvas de níveis da função de Himmelblau's, como pode-se obsevar na Figura 11(b).

O algoritmo PSO apresentado na Seção 2.2 e codificado na função `pso()` do pacote **AdequacyModel** apresentou resultados satisfatórios na obtenção de mínimos globais em ambos os casos. Para o caso da função de Rastringin (Eq. 5), nem sempre a melhor solução foi obtida, porém boas soluções foram alcançadas. A nuvem de pontos, como mostra a Figura 11(a), concentrou-se em regiões com bons candidatos à ponto de mínimo global.

A função `pso()` também forneceu bons resultados quando submetida à otimização da função de Himmelblau's (Eq.6). Nesse caso, todas as partículas foram atraídas para os vales que contém os quatro pontos de ótimos globais. Além disso, percebeu-se que a função `pso()` poderá ser útil na detecção de cavidades de uma superfície (detecção de vales). Isso muito provavelmente deve-se ao fato da recolocação aleatória de partículas no espaço de busca, o que permite que as partículas possam se dividir em grupos que não necessariamente serão atraídos para a mesma cavidade. Desde que se tenha uma representação matemática da imagem de uma superfície, a função `pso()` do pacote **AdequacyModel** poderá ser útil na detecção de cavidades na superfície. Ademais, a função `pso()` retorna um histórico dos valores de ótimos globais que podem ser úteis na detecção de regiões de cavidade.

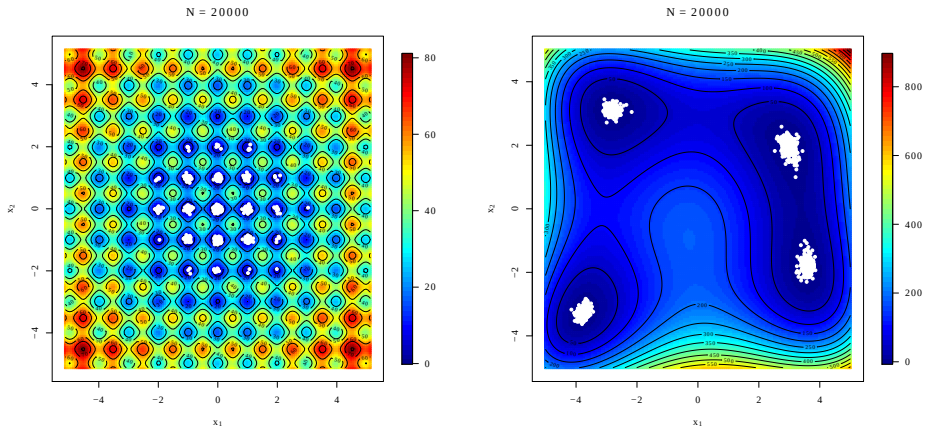

Fig 11. Curvas de níveis das funções Rastringin e Himmelblaus, respectivamente, com pontos ótimos (pontos brancos) obtidos pelas 20 mil simulações de MC ($N = 20000$).

6 Conclusions

In this paper we provided the **AdequacyModel** package for the R statistical environment with an easy-to-use set of statistical measures to assess the adequacy of lifetime models for a given data set using the PSO as the underlying optimization method. Our contribution to the PSO is to give more control over some aspects of the algorithm, such as the number of particles and iterations and a stop criterion based on the minimum number of iterations and the variance of a given proportion of optimal values. Simulation studies showed that the results obtained by the PSO implemented for the proposed package are not affected by perturbation in initial points. Regarding data analysis, our proposed package allows to easily enter with a data set for which the objective function makes use. Further, the `goodness.fit()` function provides measures that allow to compare non-nested models models using the classic AIC, CAIC, BIC statistics. Two empirical applications were presented in order to illustrate the importance and usefulness of the proposed package.
Em versão futura do pacote, será reescrita a função `pso()` utilizando a linguagem C/C++. Reescrever a função em C/C++ trará benefícios referente ao desempenho computacional da função, visto que a depender dos valores dos parâmetros `N`, `e` e `prop` da função `pso()`, poderemos facilmente nos deparar com situações computacionalmente intensivas.

A Monte Carlo simulation code

```
#!/usr/bin/env Rscript
# install.packages("purrr")
# install.packages("AdequacyModel")
# install.packages("plot3D")
# install.packages("fields")
# install.packages("parallel")

# Monte Carlo Simulation ----------------------------------------------------

simulation_mc <- function(mc = 20e3L, FUN = "rastrigin",
                    seed = 1L, plot.curve = TRUE,
                    S = 150, e = 1e-4, N = 50L,
                    prop = 0.1){

   if (FUN != "rastrigin" && FUN != "himmelblaus")
   stop("The argument ", FUN, " It is not valid.\n
         Choice \"rastrigin\" or \"himmelblaus\"")

   if (FUN == "rastrigin"){
   # Rastrigin Function ------------------------------------------------------
   obj <- function(par, x, A = 10L) {
      expr_to_eval <-
      purrr::map(.x = 1:length(par),
      .f = ~ parse(text = paste("x", .x, " <- par[", .x, "]",
                  sep = "")))
      x_vector <- NULL
      for (i in 1:length(par)) {
         eval(expr_to_eval[[i]])
         x_vector[i] <- eval(rlang::parse_expr(paste("x", i, sep = "")))
      }
      return(A * length(x_vector) +
```

```
   sum(x_vector ^ 2 - A * cos(2 * pi * x_vector)))
}
args <- list(
          func = obj,
          S = S,
          lim_inf = rep(-5.12, 2),
          lim_sup = rep(5.12, 2),
          e = e,
          N = N,
          prop = prop
       )
} else {
# Hummelblaus Function: ------------------------------------------------
obj <- function(par, x) {
   x <- par[1]
   y <- par[2]
   (x ^ 2 + y - 11) ^ 2 + (x + y ^ 2 - 7) ^ 2
}
args <- list(
          func = obj,
          S = S,
          lim_inf = rep(-5, 2),
          lim_sup = rep(5, 2),
          e = e,
          N = N,
          prop = prop
        )
 }

# One step (Monte Carlo)
onestep <- function(x, list_args) {
result <-
   do.call(getExportedValue("AdequacyModel", "pso"),
           args = list_args)
list(par = result$par, value = result$f[length(result$f)])
}

# A combined multiple-recursive generator' from L'Ecuyer (1999),
# each element of which is a feedback multiplicative generator with
# three integer elements: thus the seed is a (signed) integer vector of
# length 6. The period is around 2^191.
set.seed(seed = seed, kind = "L'Ecuyer-CMRG")

time <- system.time(
   results_mc <-
   parallel::mclapply(
   X = 1:mc,
   FUN = onestep,
   mc.cores = parallel::detectCores(),
   list_args = args
   )
) # End system.time().
```

```
   results <- unlist(results_mc)
   par_1 <- results[names(results) == "par1"]
   par_2 <- results[names(results) == "par2"]
   value <- results[names(results) == "value"]

   if (plot.curve && FUN == "rastrigin"){

   rastrigin_plot <- function(x,y){
      20  + (x ^ 2 - 10 * cos(2 * pi * x)) +
      (y ^ 2 - 10 * cos(2 * pi * y))
   }
   M  <- plot3D::mesh(seq(-5.12,  5.12, length.out = 500),
                      seq(-5.12,  5.12, length.out = 500))
   x  <- M$x ; y <- M$y

   pdf(file = "monte_carlo_rastrigin.pdf", width = 9,
       height = 9, paper = "special",
       family = "Bookman", pointsize = 14)
       z <- rastrigin_plot(x, y)
       fields::image.plot(x, y, z, xlab = bquote(x[1]),
                          ylab = bquote(x[2]),
                          main = paste0("N = ", length(par_1)))
       contour(seq(-5.12, 5.12, length.out = nrow(z)),
               seq(-5.12, 5.12, length.out = nrow(z)), z, add = TRUE)
       points(par_1, par_2, pch = 20, col = rgb(1, 1, 1))
   dev.off()
   }
   else if (plot.curve && FUN == "himmelblaus"){
      himmelblaus_plot <- function(x, y){
         (x ^ 2 + y - 11) ^ 2 + (x + y ^ 2 - 7) ^ 2
      }
   M  <- plot3D::mesh(seq(-5,  5, length.out = 500),
                      seq(-5,  5, length.out = 500))
   x  <- M$x ; y <- M$y

   pdf(file = "monte_carlo_himmelblaus.pdf", width = 9,
       height = 9, paper = "special", family = "Bookman",
       pointsize = 14)
      z <- himmelblaus_plot(x, y)
      fields::image.plot(x, y, z, xlab = bquote(x[1]),
                         ylab = bquote(x[2]),
                         main = paste0("N = ", length(par_1)))
      contour(seq(-5, 5, length.out = nrow(z)),
              seq(-5, 5, length.out = nrow(z)),
              z, add = TRUE,  nlevels = 30)
      points(par_1, par_2, pch = 20, col = rgb(1, 1, 1))
   dev.off()
   }
   list(x = par_1, y = par_2, value = value, time = time)
}
```

```
# Saving Results ------------------------------------------------------------
result_rastrigin <- simulation_mc(mc = 2e4, FUN = "rastrigin")
save(file = "simulation_rastrigin.RData",  result_rastrigin)

result_himmelblaus <- simulation_mc(mc = 2e4, FUN = "himmelblaus")
save(file = "simulation_himmelblaus.RData", result_himmelblaus)
```