Journal of Statistical Software

April 2013, Volume 53, Issue 4. *http://www.jstatsoft.org/*

GA: A Package for Genetic Algorithms in R

Luca Scrucca
Università degli Studi di Perugia

Abstract

Genetic algorithms (GAs) are stochastic search algorithms inspired by the basic principles of biological evolution and natural selection. GAs simulate the evolution of living organisms, where the fittest individuals dominate over the weaker ones, by mimicking the biological mechanisms of evolution, such as selection, crossover and mutation. GAs have been successfully applied to solve optimization problems, both for continuous (whether differentiable or not) and discrete functions.

This paper describes the R package **GA**, a collection of general purpose functions that provide a flexible set of tools for applying a wide range of genetic algorithm methods. Several examples are discussed, ranging from mathematical functions in one and two dimensions known to be hard to optimize with standard derivative-based methods, to some selected statistical problems which require the optimization of user defined objective functions. (This paper contains animations that can be viewed using the Adobe Acrobat PDF viewer.)

Keywords: optimization, evolutionary algorithms, R.

1. Introduction

Genetic algorithms (GAs) are a class of evolutionary algorithms made popular by John Holland and his colleagues during the 1970s (Holland 1975), and which have been applied to find exact or approximate solutions to optimization and search problems (Goldberg 1989; Sivanandam and Deepa 2007). Compared with other evolutionary algorithms, the distinguishing features in the original proposal were: (i) bit strings representation; (ii) proportional selection; and (iii) crossover as the main genetic operator. Since then, several other representations have been formulated in addition to binary strings. Further methods have been proposed for crossover, while mutation has been introduced as a fundamental genetic operator. Therefore, nowadays GAs belong to the larger family of evolutionary algorithms (EAs), and the two terms are often used interchangeably.

Following Spall (2004) the problem of maximizing a scalar-valued objective function $f : \mathcal{S} \rightarrow \mathbb{R}$ can be formally represented as finding the set

$$\Theta^* \equiv \arg\max_{\theta \in \Theta} f(\theta) = \{\theta^* \in \Theta : f(\theta^*) \geq f(\theta), \quad \forall\, \theta \in \Theta\}, \tag{1}$$

where $\Theta \subseteq \mathcal{S}$. The set $\mathcal{S} \subseteq \mathbb{R}^p$ defines the search space, i.e., the domain of the parameters $\theta = (\theta_1, \ldots, \theta_p)$ where each θ_i varies between the corresponding lower and upper bounds. The set Θ indicates the feasible search space, which may be defined as the intersection of $\mathcal{S}$ and a set of $m \geq 0$ additional constraints:

$$g_j(\theta) \leq 0 \quad \text{for } j = 1, \ldots, q, \qquad h_j(\theta) = 0 \quad \text{for } j = q+1, \ldots, m.$$

The solution set Θ^* in (1) may be a unique point, a countable (finite or infinite) collection of points, or a set containing an uncountable number of points.

While the formal problem representation in (1) refers to maximization of an objective function, minimizing a loss function can be trivially converted to a maximization problem by changing the sign of the objective function.

Typically, for differentiable continuous functions f the optimization problem is solved by root-finding, i.e., by looking for θ^* such that $\partial f(\theta^*)/\partial\theta_i = 0$ for every $i = 1, \ldots, p$. However, care is needed because such a root may not correspond to a global optimum of the objective function. Different techniques are required if we constrain θ to lie in a connected subset of $\mathbb{R}^p$ (constrained optimisation) or if we constrain θ to lie in a discrete set (discrete optimisation). In the latter case, also known as combinatorial optimization, the set of feasible solutions is discrete or can be reduced to discrete.

R (R Core Team 2012) includes some built-in optimization algorithms. The function `optim` provides implementations of three deterministic methods: the Nelder-Mead algorithm, a quasi-Newton algorithm (also called a variable metric algorithm), and the conjugate gradient method. Box-constrained optimization is also available. A stochastic search is provided by `optim` using simulated annealing. The function `nlm` performs minimization of a given function using a Newton-type algorithm. The golden section search for one dimensional continuous functions is available through the `optimize` function. Many other packages deal with different aspects of function optimization. A comprehensive listing of available packages is contained in the CRAN task view on "Optimization and Mathematical Programming" (Theussl 2013).

Packages **gafit** (Tendys 2002), **galts** (Satman 2012a) and **mcga** (Satman 2012b) offer some limited options for using optimization routines based on genetic algorithms. The package **rgenoud** (Mebane Jr. and Sekhon 2011) combines evolutionary algorithm methods with a derivative-based (quasi-Newton) method to solve optimization problems. **genalg** (Willighagen 2005) attempts to provide a genetic algorithm framework for both binary and floating points problems, but it is limited in scope and flexibility. **DEoptim** (Mullen, Ardia, Gil, Windover, and Cline 2011) implements the differential evolution algorithm for global optimization of a real-valued function.

The aim in writing the **GA** package was to provide a flexible, general-purpose R package for implementing genetic algorithms search in both the continuous and discrete case, whether constrained or not. Users can easily define their own objective function depending on the problem at hand. Several genetic operators are available and can be combined to explore the best settings for the current task. Furthermore, users can define new genetic operators and easily evaluate their performances. The package is available from the Comprehensive R Archive Network (CRAN) at `http://CRAN.R-project.org/package=GA`.

In the next section, we briefly review the basic ideas behind GAs. Then, we present the **GA** package in Section 3, followed by several examples on its usage in Section 4. Such examples range from mathematical functions in one and two dimensions known to be hard to optimize with standard derivative-based methods, to some selected statistical problems which require the optimization of user defined objective functions.

2. Genetic algorithms

Genetic algorithms are stochastic search algorithms which are able to solve optimization problems of the type described in Equation 1, both for continuous (whether differentiable or not) and discrete functions (Affenzeller and Winkler 2009; Back, Fogel, and Michalewicz 2000a,b; Coley 1999; Eiben and Smith 2003; Haupt and Haupt 2004; Spall 2003). Constraints on the parameters space can also be included (Yu and Gen 2010).

GAs use evolutionary strategies inspired by the basic principles of biological evolution. At a certain stage of evolution a population is composed of a number of individuals, also called strings or chromosomes. These are made of units (genes, features, characters) which control the inheritance of one or several characters. Genes of certain characters are located along the chromosome, and the corresponding string positions are called loci. Each genotype would represent a potential solution to a problem.

The decision variables, or phenotypes, in a GA are obtained by applying some mapping from the chromosome representation into the decision variable space, which represent potential solutions to an optimization problem. A suitable decoding function may be required for mapping chromosomes onto phenotypes.

The fitness of each individual is evaluated and only the fittest individuals reproduce, passing their genetic information to their offspring. Thus, with the selection operator, GAs mimic the behavior of natural organisms in a competitive environment, in which only the most qualified and their offspring survive. Two important issues in the evolution process of GAs search are exploration and exploitation. Exploration is the creation of population diversity by exploring the search space, and is obtained by genetic operators, such as mutation and crossover. Crossover forms new offsprings from two parent chromosomes by combining part of the genetic information from each. On the contrary, mutation is a genetic operator that randomly alters the values of genes in a parent chromosome. Exploitation aims at reducing the diversity in the population by selecting at each stage the individuals with higher fitness. Often an elitist strategy is also employed, by allowing the best fitted individuals to persist in the next generation in case they did not survive.

The evolution process is terminated on the basis of some convergence criteria. Usually a maximum number of generations is defined. Alternatively, a GA is stopped when a sufficiently large number of generations have passed without any improvement in the best fitness value, or when a population statistic achieves a pre-defined bound.

Figure 1 shows the flow-chart of a typical genetic algorithm. A user must first define the type of variables and their encoding for the problem at hand. Then the fitness function is defined, which is often simply the objective function to be optimized. More generally, it can be any function which assigns a value of relative merit to an individual. Genetic operators, such as

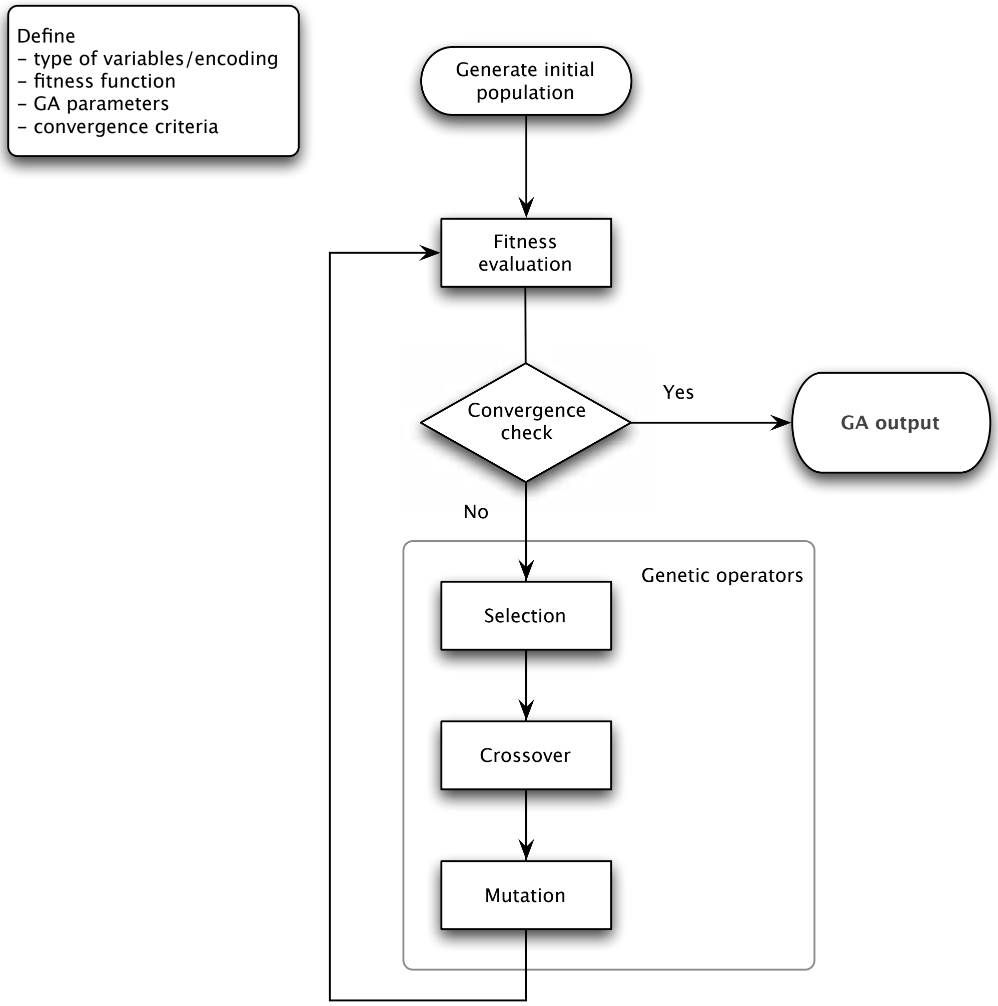


Figure 1: Flow-chart of a genetic algorithm.

crossover and mutation, are applied stochastically at each step of the evolution process, so their probabilities of occurrence must be set. Finally, convergence criteria must be supplied.

The evolution process starts with the generation of an initial random population of size n, so for step $k = 0$ we may write $\{\theta_1^{(0)}, \theta_2^{(0)}, \dots, \theta_n^{(0)}\}$. The *fitness* of each member of the population at any step k, $f(\theta_i^{(k)})$, is computed and probabilities $p_i^{(k)}$ are assigned to each individual in the population, usually proportional to their fitness. The reproducing population is formed (*selection*) by drawing with replacement a sample where each individual has probability of surviving equal to $p_i^{(k)}$. A new population $\{\theta_1^{(k+1)}, \theta_2^{(k+1)}, \dots, \theta_n^{(k+1)}\}$ is formed from the reproducing population using *crossover* and *mutation* operators. Then, set $k = k + 1$ and the algorithm returns to the fitness evaluation step. When convergence criteria are met the evolution stops, and the algorithm deliver $\theta^* \equiv \arg\max_{\theta_i^{(k)}} f(\theta_i^{(k)})$ as the optimum.

3. Overview of the GA package

The **GA** package implements genetic algorithms using S4 object-oriented programming (OOP). For an introduction to OOP in the S language see Venables and Ripley (2000), while for a more thorough treatment of the subject specifically for R see Chambers (2008) and Gentleman (2009). The proponents of OOP argue that it allows for easier design, writing and maintaining of software code. However, the actual internal implementation should be transparent to the end user, and in the following we describe the use of the package from a user perspective.

The main function in the package is called `ga`, which has the following arguments:

```
ga(type = c("binary", "real-valued", "permutation"),
  fitness, ...,
  min, max, nBits,
  population = gaControl(type)@population,
  selection = gaControl(type)@selection,
  crossover = gaControl(type)@crossover,
  mutation = gaControl(type)@mutation,
  popSize = 50, pcrossover = 0.8, pmutation = 0.1,
  elitism = max(1, round(popSize * 0.05)),
  monitor = gaMonitor, maxiter = 100, run = maxiter,
  maxfitness = -Inf, names = NULL, suggestions, seed)
```

The available arguments are:

`type` The type of genetic algorithm to be run depending on the nature of decision variables. Possible values are: `"binary"` for binary representations of decision variables; `"real-valued"` for optimization problems where the decision variables are floating-point representations of real numbers; `"permutation"` for problems that involves reordering of a list.

`fitness` The fitness function, any allowable R function which takes as input an individual `string` representing a potential solution, and returns a numerical value describing its "fitness".

`...` Additional arguments to be passed to the fitness function. This allows one to write fitness functions that keep some variables fixed during the search.

`min` A vector of length equal to the decision variables providing the minimum of the search space in case of real-valued or permutation encoded optimizations.

`max` A vector of length equal to the decision variables providing the maximum of the search space in case of real-valued or permutation encoded optimizations.

`nBits` A value specifying the number of bits to be used in binary encoded optimizations.

`population` The string name or an R function for randomly generating an initial population.

`selection` The string name or an R function performing selection, i.e., a function which generates a new population of individuals from the current population probabilistically according to individual fitness.

`crossover` The string name or an R function performing crossover, i.e., a function which forms offsprings by combining part of the genetic information from their parents.

`mutation` The string name or an R function performing mutation, i.e., a function which randomly alters the values of some genes in a parent chromosome.

`popSize` The population size.

`pcrossover` The probability of crossover between pairs of chromosomes. Typically this is a large value and by default is set to 0.8.

`pmutation` The probability of mutation in a parent chromosome. Usually mutation occurs with a small probability, and by default is set to 0.1.

`elitism` The number of best fitness individuals to survive at each generation. By default the top 5% individuals will survive at each iteration.

`monitor` An R function which takes as input the current state of the `ga` object and show the evolution of the search. By default, the function `gaMonitor` prints the average and best fitness values at each iteration. If set to `plot` these information are plotted on a graphical device. Other functions can be written by the user and supplied as argument.

`maxiter` The maximum number of iterations to run before the GA search is halted.

`run` The number of consecutive generations without any improvement in the best fitness value before the GA is stopped.

`maxfitness` The upper bound on the fitness function after that the GA search is interrupted.

`names` A vector of character strings providing the names of decision variables.

`suggestions` A matrix of solutions string to be included in the initial population.

`seed` An integer vector containing the random number generator state. This argument can be used to replicate the results of a GA search.

A call to the `ga` function should at least contain the arguments `type` and `fitness`. Furthermore, for binary search the argument `nBits` is required, whereas `min` and `max` are needed for real-valued or permutation encoding.

Default settings for genetic operators are given by the R function `gaControl`, which is described in detail in Section 3.1. Users can choose different operators among those already available and discussed in Section 3.1, or define their own genetic operators as illustrated with an example in Section 4.9.

The function `ga` returns an S4 object of class `"ga"`. This object contains slots that report most of the arguments provided in the function call, as well as the following slots:

`iter` A numerical value for the current iteration of the search.

`population` A matrix of dimension `object@popSize` times the number of decision variables.

`fitness` The evaluated fitness function for the current population of individuals.

`best` The "best" fitness value at each iteration of the GA search.

`mean` The average fitness value at each iteration of the GA search.

`fitnessValue` The "best" fitness value found by the GA search. At convergence of the algorithm this is the fitness evaluated at the solution string(s).

`solution` A matrix of solution strings, with as many rows as the number of solutions found, and as many columns as the number of decision variables.

The **GA** package is byte-compiled, as are all of standard (base and recommended) R packages. For a simple vectorized fitness function, byte-compiling may marginally improve the computational time required. However, if the fitness function is not vectorized and it must perform complex calculations, byte-compiling should significantly reduce the computational time.

3.1. Functions and genetic operators

Several R functions for generating the initial population and for applying genetic operators are contained in the **GA** package. The naming of these functions follow the scheme `ga`<*type*>`_`<*operator*> where

<*type*> can be one of `bin`, `real` or `perm`, according to the type of GA problem, and

<*operator*> identifies the genetic operator to be employed.

Note that this naming scheme is just a convention we thought was useful to adopt, but, in principle, any name could be used.

Hereafter, we briefly introduce the available operators for each GA type. Interested readers may find detailed descriptions of such operators in, for instance, Back *et al.* (2000a,b), Yu and Gen (2010) and Eiben and Smith (2003).

Population

For generating the initial population, the available R functions are:

`gabin_Population` Generate a random population of `object@nBits` binary values.

`gareal_Population` Generate a random (uniform) population of real values in the range [`object@min`, `object@max`].

`gaperm_Population` Generate a random (uniform) population of integer values in the range [`object@min`, `object@max`].

All these functions take as input an object of class `"ga"` and return a matrix of dimension `object@popSize` times the number of decision variables.

Selection

The following R functions are available for the selection genetic operator:

`gabin_lrSelection`, `gareal_lrSelection`, `gaperm_lrSelection` Linear-rank selection.

`gabin_nlrSelection`, `gareal_nlrSelection`, `gaperm_nlrSelection` Nonlinear-rank selection.

`gabin_rwSelection`, `gareal_rwSelection`, `gaperm_rwSelection` Proportional (roulette wheel) selection.

`gabin_tourSelection`, `gareal_tourSelection`, `gaperm_tourSelection` (Unbiased) tournament selection.

`gareal_lsSelection` Fitness proportional selection with fitness linear scaling.

`gareal_sigmaSelection` Fitness proportional selection with Goldberg's sigma truncation scaling.

The above functions take as arguments an object of class `"ga"` and, possibly, other parameters controlling the genetic operator. They all returns a list with two elements:

`population` A matrix of dimension `object@popSize` times the number of decision variables containing the selected individuals or strings.

`fitness` A vector of length `object@popSize` containing the fitness values for the selected individuals.

Crossover

Available R functions for the crossover genetic operator are:

`gabin_spCrossover`, `gareal_spCrossover` Single-point crossover.

`gabin_uCrossover`, `gareal_uCrossover` Uniform crossover.

`gareal_waCrossover` Whole arithmetic crossover.

`gareal_laCrossover` Local arithmetic crossover.

`gareal_blxCrossover` Blend crossover.

`gaperm_cxCrossover` Cycle crossover.

`gaperm_pmxCrossover` Partially matched crossover.

`gaperm_oxCrossover` Order crossover.

`gaperm_pbxCrossover` Position-based crossover.

These functions take as arguments an object of class `"ga"` and a two-rows matrix of values indexing the `parents` from the current population. They all return a list with two elements:

`children` A matrix of dimension 2 times the number of decision variables containing the generated offsprings.

`fitness` A vector of length 2 containing the fitness values for the offsprings. A value `NA` is returned if an offspring is different (which is usually the case) from the two parents.

Mutation

Available R functions for the mutation genetic operator are:

`gabin_raMutation`, `gareal_raMutation` Uniform random mutation.

`gareal_nraMutation` Nonuniform random mutation.

`gareal_rsMutation` Random mutation around the solution.

`gaperm_simMutation` Simple inversion mutation.

`gaperm_ismMutation` Insertion mutation.

`gaperm_swMutation` Exchange mutation or swap mutation.

`gaperm_dmMutation` Displacement mutation.

`gaperm_scrMutation` Scramble mutation.

These functions take as arguments an object of class `"ga"` and a vector of values for the `parent` from the current population where mutation should occur. They all return a vector of values containing the mutated string.

Default settings

The function `ga` uses a set of default settings for genetic operators. These can be retrieved or set with the function `gaControl`. Its usage depends on the arguments provided. A call with no arguments returns a list containing the current values, which by defaults are:

```
R> gaControl()

$binary
$binary$population
[1] "gabin_Population"
$binary$selection
[1] "gabin_lrSelection"
$binary$crossover
[1] "gabin_spCrossover"
$binary$mutation
[1] "gabin_raMutation"

$`real-valued`
$`real-valued`$population
[1] "gareal_Population"
$`real-valued`$selection
[1] "gareal_lsSelection"
$`real-valued`$crossover
[1] "gareal_laCrossover"
$`real-valued`$mutation
```

```
[1] "gareal_raMutation"

$permutation
$permutation$population
[1] "gaperm_Population"
$permutation$selection
[1] "gaperm_lrSelection"
$permutation$crossover
[1] "gaperm_oxCrossover"
$permutation$mutation
[1] "gaperm_simMutation"

$eps
[1] 1.490116e-08
```

A call to `gaControl` with a single string specifying the name of the component returns the current value(s):

```
R> gaControl("binary")

$population
[1] "gabin_Population"

$selection
[1] "gabin_lrSelection"

$crossover
[1] "gabin_spCrossover"

$mutation
[1] "gabin_raMutation"
```

In this case the function returns the current genetic operators used by the `"binary"` GAs search.

To change the default values, a named component must be followed by a single value (in case of `"eps"`) or a list of component(s) specifying the name of the function for a genetic operator. For instance, the following code saves the current default values, and then set the tournament selection as the new default for binary GAs:

```
R> defaultControl <- gaControl()
R> gaControl("binary" = list(selection = "gabin_tourSelection"))
```

When any value is set by `gaControl`, this will remain in effect for the rest of the session. To restore the previously saved package defaults:

```
R> gaControl(defaultControl)
```

4. Examples

Many examples concerning optimization tasks are provided in this Section. In particular, we will present the optimization of well-known benchmark mathematical functions, but also applications of genetic algorithms in a variety of statistical problems.

Hereafter, we assume that the **GA** package is already installed and loaded in the current R session, for example by entering the following command:

```
R> library("GA")
```

4.1. Function optimization on one dimension

We start by a simple one-dimensional function minimization by considering the function $f(x) = |x| + \cos(x)$, which has $\min f(0) = 1$ for $-\infty < x < +\infty$ (see test function F1 in Haupt and Haupt 2004). Here we restrict our attention to $x \in [-20, 20]$, so this function can be defined and plotted in R as follows:

```
R> f <- function(x) abs(x) + cos(x)
R> min <- -20
R> max <- +20
R> curve(f, min, max)
```

We can define the fitness function, which in this case is simply minus the function to minimize, and run the genetic algorithm with the code:

```
R> fitness <- function(x) -f(x)
R> GA <- ga(type = "real-valued", fitness = fitness, min = min, max = max)
```

Here we specified `type = "real-valued"` for a real-valued function optimization using the R function `fitness` as the objective function to be maximized over the range provided by the arguments `min` and `max`. By default the `ga` function monitors the search by printing the mean and the best fitness values at each iteration:

```
Iter = 1   | Mean = -10.30292   | Best = -1.106484
Iter = 2   | Mean = -5.740801   | Best = -1.106484
[...]
Iter = 100   | Mean = -2.103191   | Best = -1.000008
```

At the end of the search an S4 object of class `"ga"` is returned, which can be printed and plotted as follows:

```
R> GA

An object of class "ga"

Call:
ga(type = "real-valued", fitness = fitness, min = min, max = max)
```

```
Available slots:
 [1] "call"         "type"         "min"          "max"
 [5] "nBits"        "names"        "popSize"      "iter"
 [9] "run"          "maxiter"      "suggestions"  "population"
[13] "elitism"      "pcrossover"   "pmutation"    "fitness"
[17] "best"         "mean"         "fitnessValue" "solution"
```

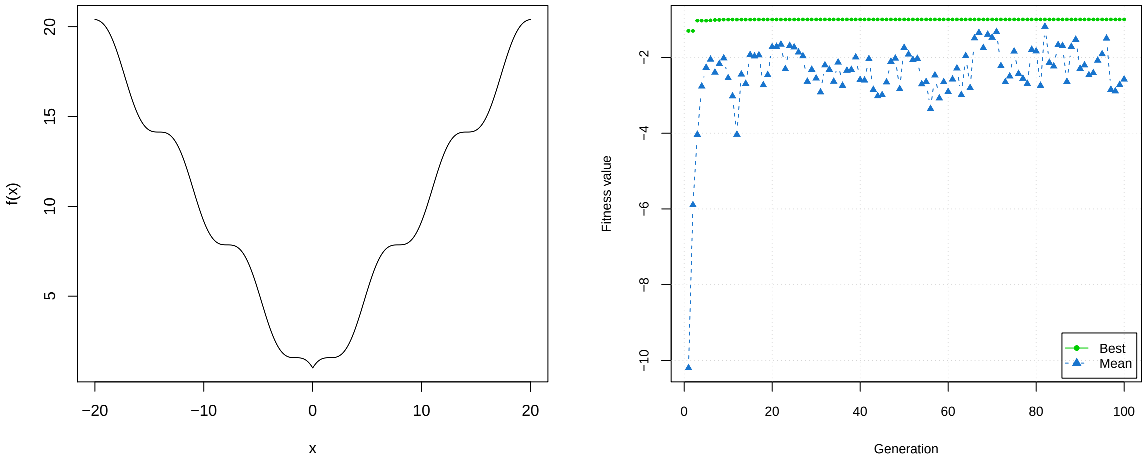
```
R> plot(GA)
R> summary(GA)
```

```
+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  50
Number of generations =  100
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
     x1
Min -20
Max  20

GA results:
Iterations             = 100
Fitness function value = -1.000008
Solution               =
                x1
[1,] -7.697129e-06
```

The `plot` method produces the graph in Figure 2b, where the best and average fitness values along the iterations are shown.

Figure 2a contains an animation of the GA search, which shows the evolution of the population units and the corresponding functions values at each generation. This has been obtained by defining a new `monitor` function and then passing this function as an optional argument to `ga`:

```
R> monitor <- function(obj) {
+    curve(f, min, max, main = paste("iteration =", obj@iter), font.main = 1)
+    points(obj@population, -obj@fitness, pch = 20, col = 2)
+    rug(obj@population, col = 2)
+    Sys.sleep(0.2)
+  }
```



(a) (b)

Figure 2: One-dimensional test function: $f(x) = |x| + \cos(x)$. Panel (a) shows the function; if you are viewing this in Acrobat, click on the image to see an animation of fitness evaluation at each iteration. Panel (b) shows best and average fitness valuew at each GA generation step.

```
R> GA <- ga(type = "real-valued", fitness, min = min, max = max,
+    monitor = monitor)
```

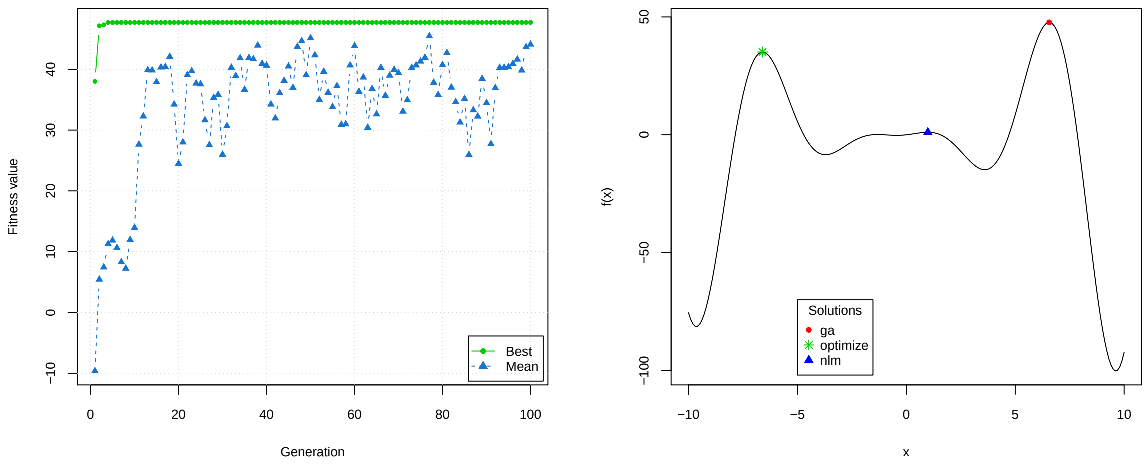
Consider now the function $f(x) = (x^2 + x)\cos(x)$ defined over the range $-10 \leq x \leq 10$ (see test function F6 in Haupt and Haupt 2004). This function can be defined and plotted in R as follows:

```
R> f <- function(x) (x^2 + x) * cos(x)
R> min <- -10
R> max <- 10
R> curve(f, min, max)
```

For the maximization of this function we may use `f` directly as the fitness function:

```
R> GA <- ga(type = "real-valued", fitness = f, min = min, max = max)
R> plot(GA)
R> summary(GA)
```

```
+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  50
Number of generations =  100
```



(a)

(b)

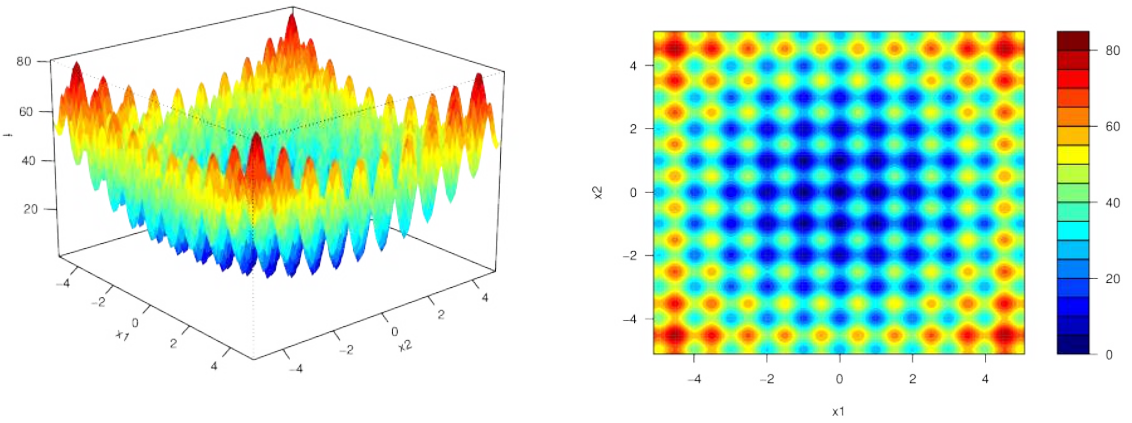
Figure 3: One-dimensional test function: $f(x) = (x^2 + x)\cos(x)$. Panel (a) shows best and average fitness values at each GA generation step. Panel (b) shows the solutions found by GA and two other numerical optimization algorithms available in R.

```
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
     x1
Min -10
Max  10

GA results:
Iterations             = 100
Fitness function value = 47.70562
Solution               =
           x1
[1,] 6.560539
```

Figure 3a shows the evolution of GA search, which quickly identifies the global maximum of the function (see Figure 3b).

The final GA result can be compared with the solutions provided by two other optmization algorithms available in R: `optimize`, which uses a combination of golden section search and successive parabolic interpolation, and `nlm`, which uses a Newton-type algorithm. The results shown in Figure 3b makes clear that the latter two optimization algorithms are both trapped in local maxima, while the GA is able to identify the right global maximum. The code used to obtain this graph is the following:

```
R> opt.sol <- optimize(f, lower = min, upper = max, maximum = TRUE)
R> nlm.sol <- nlm(function(...) -f(...), 0, typsize = 0.1)
R> curve(f, min, max)
```



(a) (b)

Figure 4: Panel (a) shows a perspective plot of the Rastrigin test function, while panel (b) shows the corresponding contours. If you are viewing this in Acrobat, click on the panel (b) image to see an animation of fitness evaluation at each GA iteration.

```
R> points(GA@solution, GA@fitnessValue, col = 2, pch = 20)
R> points(opt.sol$maximum, opt.sol$objective, col = 3, pch = 8)
R> points(nlm.sol$estimate, -nlm.sol$minimum, col = 4, pch = 17)
R> legend(x = -5, y = -70, legend = c("ga", "optimize", "nlm"),
+    title = "Solutions", pch = c(20,8,17), col = 2:4)
```

4.2. Function optimization on two dimensions

The *Rastrigin function* is a non-convex function often used as a test problem for optimization algorithms because it is a difficult problem due to its large number of local minima. In two dimensions it is defined as

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10(\cos(2\pi x_1) + \cos(2\pi x_2)),$$

with $x_i \in [-5.12, 5.12]$ for $i = 1, 2$. It has a global minimum at $(0, 0)$ where $f(0, 0) = 0$. Figure 4 shows a perspective plot[1] and a contour plot of the Rastrigin function obtained as follows:

```
R> x1 <- x2 <- seq(-5.12, 5.12, by = 0.1)
R> f <- outer(x1, x2, Rastrigin)
R> persp3D(x1, x2, f, theta = 50, phi = 20)
R> filled.contour(x1, x2, f, color.palette = jet.colors)
```

The optimization of this function with the monitoring of the space searched at each GA iteration (see Figure 4b) can be obtained through the following code:

[1] The function `persp3D`, included in the **GA** package, is an enhanced version of the base `persp` function.

```
R> monitor <- function(obj) {
+    contour(x1, x2, f, drawlabels = FALSE, col = gray(0.5))
+    title(paste("iteration =", obj@iter), font.main = 1)
+    points(obj@population, pch = 20, col = 2)
+    Sys.sleep(0.2)
+  }
R> GA <- ga(type = "real-valued",
+    fitness = function(x) -Rastrigin(x[1], x[2]),
+    min = c(-5.12, -5.12), max = c(5.12, 5.12), popSize = 50,
+    maxiter = 100, monitor = monitor)
R> summary(GA)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  50
Number of generations =  100
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
       x1    x2
Min -5.12 -5.12
Max  5.12  5.12

GA results:
Iterations             = 100
Fitness function value = -3.462895e-06
Solution               =
                x1           x2
[1,] -0.0001164603 6.238426e-05
```

Looking at the final result, we may conclude that GA did a pretty good job in recovering the area where the minimum of the Rastrigin function lies. To refine the search, the final solution of GA can be input to a derivative-based optimization algorithm, for instance:

```
R> NLM <- nlm(function(x) Rastrigin(x[1], x[2]), GA@solution)
R> NLM[c("minimum", "estimate")]

$minimum
[1] 0
$estimate
[1] 2.518049e-14 4.492133e-14
```

4.3. A robust estimator: Andrews Sine function

Chatterjee, Laudato, and Lynch (1996) discussed an application of GAs to obtain robust estimates of coefficients of a linear model. For the usual model $y_i = \beta^\top \boldsymbol{x}_i + \epsilon_i$, where $\boldsymbol{x} = (1, x_1, \ldots, x_p)$ is a vector of predictors independent from the error component having $\mathsf{E}(\epsilon) = 0$, the coefficients β are obtained by minimizing

$$\sum_{i=1}^{n} \rho\left(\frac{y_i - \beta^\top \boldsymbol{x}_i}{s}\right)$$

where s is a scaling factor which can be set at 1. Using $\rho(x) = x^2$, we revert to the usual OLS estimator. A robust estimator can be obtained by using the Andrews Sine function (Andrews 1974) defined as follows:

$$\rho(x) = \begin{cases} a^2(1 - \cos(x/a)) & \text{if } |x| \leq \pi a \\ 2a^2 & \text{if } |x| > \pi a \end{cases},$$

where $a = 1.5$ in the original proposal by Andrews.

The Andrews Sine function and the fitness function to be used in the GA (recall that we need to maximize the fitness) are defined as:

```
R> AndrewsSineFunction <- function(x, a = 1.5)
+    ifelse(abs(x) > pi * a, 2 * a^2, a^2 * (1 - cos(x/a)))
R> rob <- function(b, s = 1)
+    -sum(AndrewsSineFunction((y - X %*% b)/s))
```

We apply the robust fitting procedure to the well-known `stackloss` dataset available in the **datasets** package.

```
R> data("stackloss", package = "datasets")
```

The range of the search space can be obtained from a preliminary OLS estimation of the coefficients and their standard errors:

```
R> OLS <- lm(stack.loss ~ ., data = stackloss)
R> y <- model.response(model.frame(OLS))
R> X <- model.matrix(OLS)
R> se.coef <- sqrt(diag(vcov(OLS)))
R> min <- coef(OLS) - 3 * se.coef
R> max <- coef(OLS) + 3 * se.coef
```

We can now run the GA search, this time by using a large number of possible iterations and increasing the probability of mutation to ensure that vast portion of the parameter space is explored.

```
R> GA <- ga(type = "real-valued", fitness = rob, min = min, max = max,
+    popSize = 100, pmutation = 0.2, maxiter = 5000, run = 200)
R> summary(GA)
```

```
+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  100
Number of generations =  5000
Elitism               =  5
Crossover probability =  0.8
Mutation probability  =  0.2
Search domain
    (Intercept)  Air.Flow Water.Temp Acid.Conc.
Min  -75.607665 0.3110656  0.1912133 -0.6210046
Max   -4.231684 1.1202148  2.3993589  0.3167596

GA results:
Iterations             = 3738
Fitness function value = -26.96648
Solution               =
     (Intercept) Air.Flow Water.Temp  Acid.Conc.
[1,]   -37.17334  0.81665  0.5240304 -0.07210479
```

The final robust coefficients estimated are equal, up to the second significant digit, to those provided by Chatterjee *et al.* (1996).

4.4. Curve fitting

Jones, Maillardet, and Robinson (2009, p. 219–220) presented a curve fitting application using data on the growth of trees. The relationship bewteen the volume of the trunk (in m^3) of a spruce tree as function of age (years since the trunk reached a height of 1.3 m) is modeled using a popular ecological model known as Richards curve:

$$f(x) = a\left(1 - e^{-bx}\right)^c.$$

This is a nonlinear regression model with parameters $\theta = (a, b, c)^\top$. Using a quadratic loss function (i.e., nonlinear least squares), the Richards curve can be fitted using genetic algorithms as follows:

```
R> data("trees", package = "spuRs")
R> tree <- trees[trees$ID == "1.3.11", 2:3]
R> richards <- function(x, theta)
+    theta[1] * (1 - exp(-theta[2] * x))^theta[3]
R> fitnessL2 <- function(theta, x, y) -sum((y - richards(x, theta))^2)
R> GA2 <- ga(type = "real-valued", fitness = fitnessL2,
+    x = tree$Age, y = tree$Vol, min = c(3000, 0, 2), max = c(4000, 1, 4),
+    popSize = 500, crossover  = gareal_blxCrossover, maxiter = 5000,
+    run = 200, names = c("a", "b", "c"))
R> summary(GA2)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  500
Number of generations =  5000
Elitism               =  25
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
       a b c
Min 3000 0 2
Max 4000 1 4

GA results:
Iterations             = 585
Fitness function value = -2773.837
Solution               =
            a          b        c
[1,] 3592.376 0.01544009 2.783055
```

Note that here the fitness function `fitnessL2` needs to be maximized with respect to the parameters in `theta`, given the observed data in `x` and `y`. The latter are supplied as a further argument in the call to the function `ga` and are kept fixed during the search.

Furthermore, we increased population size to 500, and we decided to stop the algorithm after at most 5000 iterations or 200 generations without improving the fitness. Finally, we adopted a blend crossover for improving the search over the parameter space.

If a different loss function is required, for instance a L1-norm, we simply need to change the fitness function as follows:

```
R> fitnessL1 <- function(theta, x, y) -sum(abs(y - richards(x, theta)))
R> GA1 <- ga(type = "real-valued", fitness = fitnessL1,
+    x = tree$Age, y = tree$Vol, min = c(3000, 0, 2), max = c(4000, 1, 4),
+    popSize = 500, crossover  = gareal_blxCrossover, maxiter = 5000,
+    run = 200,  names = c("a", "b", "c"))
R> summary(GA1)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+
GA settings:
Type                  =  real-valued
Population size       =  500
Number of generations =  5000
Elitism               =  25
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
       a b c
Min 3000 0 2
Max 4000 1 4

GA results:
Iterations             = 937
Fitness function value = -134.017
Solution               =
            a          b        c
[1,] 3534.741 0.01575454 2.800797
```

4.5. Subset selection

A typical application of binary GAs in statistical modeling is subset selection (see e.g., the R package **glmulti**, Calcagno and de Mazancourt 2010). Given a set of p predictors, subset selection aims at identifying those predictors which are most relevant for explaining the variation of a response variable. This allows one to achieve parsimony of unknown parameters, yielding both better estimation and clearer interpretation of regression coefficients. The problem of subset selection can be naturally treated by GAs using a binary string, with 1 indicating the presence of a predictor and 0 its absence from a given candidate subset. The fitness of a candidate subset can be measured by one of the several model selection criteria, such as AIC, BIC, etc.

Bozdogan (2004) discussed the use of GAs for subset selection in linear regression models, and in the following we present an application closely following his analysis, but with the use of Akaike's information criterion (AIC; Akaike 1973). We start by loading the dataset from the **UsingR** package (Verzani 2005) and then we fit a linear regression model by OLS:

```
R> data("fat", package = "UsingR")
R> mod <- lm(body.fat.siri ~ age + weight + height + neck + chest + abdomen +
+    hip + thigh + knee + ankle + bicep + forearm + wrist, data = fat)
R> summary(mod)

[...]
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -18.18849   17.34857  -1.048  0.29551
age           0.06208    0.03235   1.919  0.05618 .
weight       -0.08844    0.05353  -1.652  0.09978 .
```

```
height       -0.06959    0.09601  -0.725  0.46925
neck         -0.47060    0.23247  -2.024  0.04405 *
chest        -0.02386    0.09915  -0.241  0.81000
abdomen       0.95477    0.08645  11.044  < 2e-16 ***
hip          -0.20754    0.14591  -1.422  0.15622
thigh         0.23610    0.14436   1.636  0.10326
knee          0.01528    0.24198   0.063  0.94970
ankle         0.17400    0.22147   0.786  0.43285
bicep         0.18160    0.17113   1.061  0.28966
forearm       0.45202    0.19913   2.270  0.02410 *
wrist        -1.62064    0.53495  -3.030  0.00272 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.305 on 238 degrees of freedom
Multiple R-squared: 0.749, Adjusted R-squared: 0.7353
F-statistic: 54.65 on 13 and 238 DF,  p-value: < 2.2e-16
```

The design matrix (without the intercept) and the response variable are extracted from the fitted model object using:

```
R> x <- model.matrix(mod)[, -1]
R> y <- model.response(model.frame(mod))
```

Then, the fitness function to be maximized can be defined as follows:

```
R> fitness <- function(string) {
+    inc <- which(string == 1)
+    X <- cbind(1, x[,inc])
+    mod <- lm.fit(X, y)
+    class(mod) <- "lm"
+    -AIC(mod)
+  }
```

which simply estimates the regression model using the predictors identified by a 1 in the corresponding position of `string`, and returns the negative of the chosen criterion. Note that an intercept term is always included, and that we employ the basic `lm.fit` function to speed up calculations. The following R code runs the GA:

```
R> GA <- ga("binary", fitness = fitness, nBits = ncol(x),
+    names = colnames(x), monitor = plot)
R> plot(GA)
R> summary(GA)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+
```

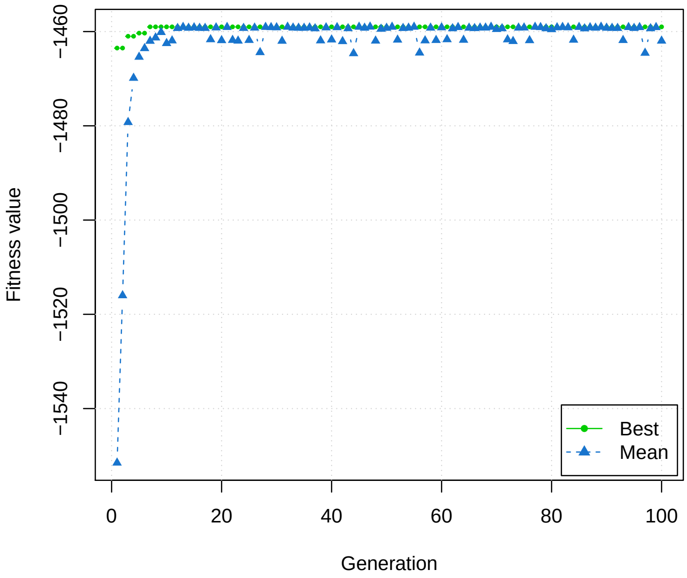


Figure 5: Plot of best and average fitness values at each step of the GA search. If you are viewing this in Acrobat, click on the image to see an animation of fitness evaluation during the GA iterations.

```
GA settings:
Type                  =  binary
Population size       =  50
Number of generations =  100
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1

GA results:
Iterations             = 100
Fitness function value = -1458.996
Solution               =
     age weight height neck chest abdomen hip thigh knee ankle bicep
[1,]   1      1      0    1     0       1   1     1    0     0     0
     forearm wrist
[1,]       1     1
```

A graphical summary of the GA search is shown in Figure 5.

The linear regression model fit obtained using the best subset found by GA is the following:

```
R> mod2 <- lm(body.fat.siri ~ .,
+    data = data.frame(body.fat.siri = y, x[,GA@solution == 1]))
R> summary(mod2)

[...]
```

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -22.65637   11.71385  -1.934  0.05426 .
age           0.06578    0.03078   2.137  0.03356 *
weight       -0.08985    0.03991  -2.252  0.02524 *
neck         -0.46656    0.22462  -2.077  0.03884 *
abdomen       0.94482    0.07193  13.134  < 2e-16 ***
hip          -0.19543    0.13847  -1.411  0.15940
thigh         0.30239    0.12904   2.343  0.01992 *
forearm       0.51572    0.18631   2.768  0.00607 **
wrist        -1.53665    0.50939  -3.017  0.00283 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.282 on 243 degrees of freedom
Multiple R-squared: 0.7466, Adjusted R-squared: 0.7382
F-statistic: 89.47 on 8 and 243 DF,  p-value: < 2.2e-16
```

Compared to Bozdogan (2004) solution, which used the ICOMP(IFIM) criterion for evaluating the subsets, the GA solution with fitness based on AIC selects one more predictor, namely `weight`. Because of its strong collinearity with `hip` ($r = 0.94$), the latter predictor is not statistically significant (p value $= 0.1594$). This result is not surprising, since it is known that AIC tends to overestimate the number of predictors required, while ICOMP(IFIM) is able to protect against multicollinearity.

4.6. Acceptance sampling

Acceptance sampling is an area of applied statistics where sampling is used to determine whether to accept or reject a production lot of material (raw materials, semifinished products, or finished products). An introduction to acceptance sampling is contained in the textbook by Montgomery (2009), while a monograph devoted to the argument is Schilling and Neubauer (2009).

In acceptance sampling for attributes, only the presence or absence of a characteristic in the inspected item is recorded. Among the available sampling plans, the single-sampling plan involves taking a random sample of size n from a lot of size N. The number d of defective items found is compared to an acceptance number c, and the lot is accepted if $d \leq c$. The probability of acceptance P_a can be computed by assuming a Binomial distribution for the number of defectives in a lot (the so-called type B sampling). Thus, such a probability is given by

$$P_a = \sum_{d=0}^{c} \binom{n}{d} p^d (1-p)^{n-d}.$$

A plot of P_a vs p is called operating characteristic (OC) curve, and expresses the probability of acceptance as a function of lot quality.

In practical applications, a single-sampling plan needs the specification of the sample size n and the acceptance number c. This is usually pursued by specifying two points on the OC curve and solving the resulting system of equations:

$$\begin{cases} 1-\alpha & = \displaystyle\sum_{d=0}^{c} \binom{n}{d} p_1^d (1-p_1)^{n-d} \\ \beta & = \displaystyle\sum_{d=0}^{c} \binom{n}{d} p_2^d (1-p_2)^{n-d} \end{cases}, \tag{2}$$

where p_1 is typically set at the average quality limit (AQL), and p_2 at the lot tolerance percent defective (LTPD). The system of equations in (2) is nonlinear and no direct solution is available. Traditionally, a graph called *nomogram* is consulted for obtaining the pair (n, c) that solves (2), at least approximately due to discreteness of parameters. Below we present a simple solution to this problem using GAs.

We may define and plot the two selected points on the OC curve as follows:

```
R> AQL <- 0.01
R> alpha <- 0.05
R> LTPD <- 0.06
R> beta <- 0.10
R> plot(0, 0, type = "n", xlim = c(0, 0.2), ylim = c(0, 1), bty = "l",
+    xaxs = "i", yaxs = "i", ylab = expression(P[a]), xlab = expression(p))
R> lines(c(0,AQL), rep(1 - alpha, 2), lty = 2, col = "gray")
R> lines(rep(AQL,2), c(1 - alpha, 0), lty = 2, col = "gray")
R> lines(c(0,LTPD), rep(beta,2), lty = 2, col = "gray")
R> lines(rep(LTPD, 2), c(beta,0), lty = 2, col = "gray")
R> points(c(AQL, LTPD), c(1 - alpha, beta), pch = 16)
R> text(AQL, 1 - alpha,
+    labels = expression(paste("(", AQL, ", ", 1 - alpha, ")")), pos = 4)
R> text(LTPD, beta,
+    labels = expression(paste("(", LTPD, ", ", beta, ")")), pos = 4)
```

with the resulting graph shown in Figure 6a.

Given that both n and c should be positive integer values, we may use binary GAs with Gray encoding. This eliminates the well-known Hamming cliff problem associated with binary coding. As an example, consider a five-bit encoding using the standard binary coding. Two consecutive integers, for instance 15 and 16, are encoded as:

```
R> decimal2binary(15, 5)

[1] 0 1 1 1 1

R> decimal2binary(16, 5)

[1] 1 0 0 0 0
```

then moving from 15 to 16 (or vice versa) all five bits need to be changed. On the other hand, using Gray encoding:

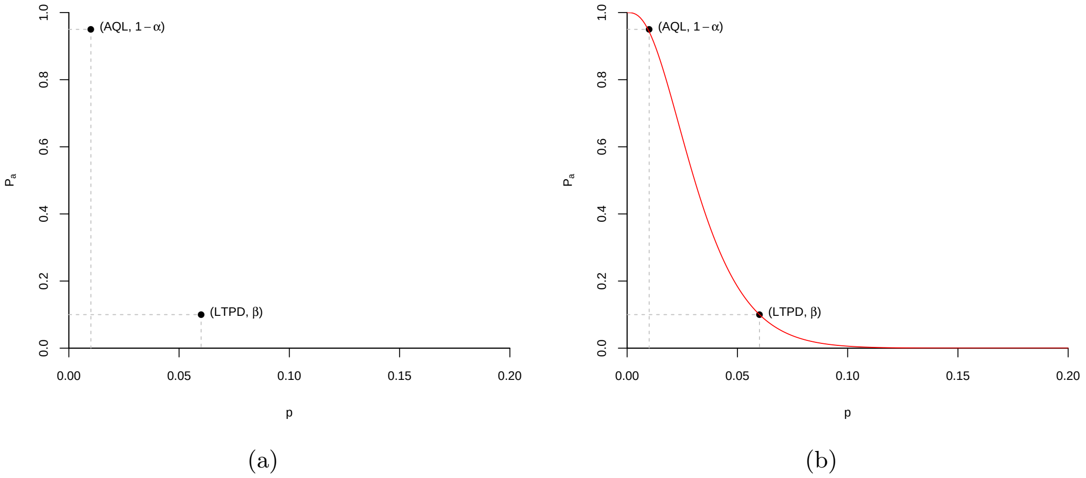


Figure 6: OC curve for single acceptance sampling plan. Panel (a) shows the two fixed points for which a solution is sought. Panel (b) shows the OC curve for the solution ($n = 87, c = 2$) found by GAs.

```
R> binary2gray(decimal2binary(15, 5))

[1] 0 1 0 0 0

R> binary2gray(decimal2binary(16, 5))

[1] 1 1 0 0 0
```

the two binary strings differ by one bit only. Thus, in Gray encoding the number of bit differences between any two consecutive strings is one, whereas in binary strings this is not always true. The R functions `binary2decimal` and `gray2binary` are also available to move from one type of encoding to another.

Returning to our problem, a decoding function which takes as input a solution string of binary values in Gray representation, and then transform it to a decimal representation for the pair (n, c) can be defined in R as:

```
R> decode <- function(string) {
+    string <- gray2binary(string)
+    n <- binary2decimal(string[1:l1])
+    c <- min(n, binary2decimal(string[(l1 + 1):(l1 + l2)]))
+    return(c(n, c))
+  }
```

where `l1` and `l2` are the number of bits required to separately encode the two parameters.

The fundamental step for solving (2) via GAs is to define a loss (quadratic) function to evaluate a proposal solution pair:

```
R> fitness <- function(string) {
+    par <- decode(string)
+    n <- par[1]
+    c <- par[2]
+    Pa1 <- pbinom(c, n, AQL)
+    Pa2 <- pbinom(c, n, LTPD)
+    Loss <- (Pa1 - (1 - alpha))^2 + (Pa2 - beta)^2
+    -Loss
+  }
```

Then, a GA search is run over the user-defined search space $[2, 200] \times [0, 20]$:

```
R> n <- 2:200
R> c <- 0:20
R> b1 <- decimal2binary(max(n))
R> l1 <- length(b1)
R> b2 <- decimal2binary(max(c))
R> l2 <- length(b2)
R> GA <- ga(type = "binary", nBits = l1+l2, fitness = fitness,
+    popSize = 200, maxiter = 200, run = 100)
R> summary(GA)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  binary
Population size       =  200
Number of generations =  200
Elitism               =  10
Crossover probability =  0.8
Mutation probability  =  0.1

GA results:
Iterations             = 121
Fitness function value = -5.049435e-05
Solution               =
     x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13
[1,]  0  1  1  1  1  1  0  0  1   0   0   1   1

R> decode(GA@solution)

[1] 87  2
```

The final solution provided is thus decoded to obtain the solution pair ($n = 87, c = 2$). The corresponding OC curve is shown in Figure 6b. This is obtained from Figure 6a by adding the solution OC curve as follows:

```
R> n <- 87
R> c <- 2
R> p <- seq(0, 0.2, by = 0.001)
R> Pa <- pbinom(2, 87, p)
R> lines(p, Pa, col = 2)
```

An advantage of using GAs for solving the problem of identifying the parameters of an acceptance sampling plan is that this approach can be easily extended to more complicated plans, for instance double sampling plan, by suitably modifying the functions `fitness` and `decode`.

4.7. Constrained optimization

The *knapsack problem* considers the maximization of the weighted profit subject to the constraint of the knapsack's capacity. Formally, given a set of weights w_i, profits p_i, and knapsack's capacity W, find a binary vector $x = (x_1, \ldots, x_n)$ such that $\sum_{i=1}^n x_i p_i$ is maximized under the constraint that $\sum_{i=1}^n x_i w_i \leq W$. The solution to this problem is a binary string of length n where $x_i = 1$ if the i-th item is selected for the knapsack, and $x_i = 0$ otherwise.

Consider the following data (Yu and Gen 2010, p. 271) with profit (`p`), weight (`w`), and capacity (`W`):

```
R> p <- c(6, 5, 8, 9, 6, 7, 3)
R> w <- c(2, 3, 6, 7, 5, 9, 4)
R> W <- 9
```

A binary GA can be used to solve the knapsack problem, but not all the possible solutions are feasible due to the inequality constraint. We may take the constraint into account by penalizing unfeasible solutions. Thus, the fitness function can be defined as follows:

```
R> knapsack <- function(x) {
+    f <- sum(x * p)
+    penalty <- sum(w) * abs(sum(x * w)-W)
+    f - penalty
+  }
```

where the objective function `f` is penalized with `penalty`, a quantity that depends on the distance between the capacity of the proposed solution to the knapsack's capacity. Then:

```
R> GA <- ga(type = "binary", fitness = knapsack, nBits = length(w),
+    maxiter = 1000, run = 200, popSize = 20)
R> summary(GA)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  binary
Population size       =  20
```

```
Number of generations =  1000
Elitism               =  1
Crossover probability =  0.8
Mutation probability  =  0.1

GA results:
Iterations             = 202
Fitness function value = 15
Solution               =
     x1 x2 x3 x4 x5 x6 x7
[1,]  1  0  0  1  0  0  0
```

```
R> sum(p * GA@solution)
```

```
[1] 15
```

```
R> sum(w * GA@solution)
```

```
[1] 9
```

The inclusion of an effective and efficient penalty term in the fitness function can be difficult, and often requires some tuning depending on the specific problem. Furthermore, this approach is suitable in the presence of inequality constraints. In the case of equality constraints, different approaches should be adopted. A procedure for the inclusion of equality constraints is provided by repair algorithms, which simply repair an infeasible solution before fitness evaluation. Although this is a straightforward procedure, it is unlikely to be computationally very efficient. A better procedure may be found if the constrained optimization could be re-expressed as an unconstrained problem.

Consider the allele frequency estimation problem in Lange (2004, p. 123–125). For the three alleles A, B, and O, there are four observable phenotypes A, B, AB, and O. This is because each individual inherits two alleles from the parents, and alleles A and B are genetically dominant to allele O. Lange (2004) considered a sample of $n = 521$ duodenal ulcer patients whose frequency distribution of the observed phenotype is: $n_A = 186, n_B = 38, n_{AB} = 13, n_O = 284$. Assuming a multinomial distribution in conjunction with the Hardy-Weinberg law of population genetics, Lange (2004) maximized the log-likehood using a MM algorithm. The solution found is $p = (p_A, p_B, p_O) = (0.2136, 0.0501, 0.7363)$ with corresponding log-likelihood (except for a constant term involving the multinomial coefficient) equal to -511.5715.

We start by applying a constrained GA with repairing:

```
R> n.A <- 186
R> n.B <- 38
R> n.AB <- 13
R> n.O <- 284
R> loglik <- function(p) {
+    n.A * log(p[1]^2 + 2 * p[1] * p[3]) +
+    n.B * log(p[2]^2 + 2 * p[2] * p[3]) +
+    n.AB * log(2 * p[1] * p[2]) +
+    n.O * log(p[3]^2)
+  }
R> fitness <- function(p) {
+    p <- p/sum(p)
+    loglik(p)
+  }
R> GA <- ga(type = "real-valued", fitness = fitness,
+    min = c(0, 0, 0), max = c(1, 1, 1), popSize = 50, maxiter = 1000,
+    run = 100, names = c("A", "B", "O"))
R> p <- GA@solution/sum(GA@solution)
R> print(p)

             A          B         O
[1,] 0.2134997 0.05004689 0.7364534

R> loglik(p)

[1] -511.5716

R> GA@iter

[1] 425
```

This approach yields a solution very close to that provided by Lange (2004). An unconstrained optimization can be pursued via parameter transformation using the inverse multinomial logit transformation:

```
R> invmlogit <- function(theta) {
+    p <- rep(0, length(theta) + 1)
+    p[1] <- 1/(1 + sum(exp(theta)))
+    p[-1] <- exp(theta) * p[1]
+    return(p)
+  }
R> fitness <- function(theta) {
+    p <- invmlogit(theta)
+    loglik(p)
+  }
R> GA <- ga(type = "real-valued", fitness = fitness, min = rep(-3, 2),
+    max = rep(3, 2), popSize = 50, maxiter = 1000, run = 100)
R> p <- invmlogit(GA@solution)
R> names(p) <- c("A", "B", "O")
R> print(p)

         A          B          O
0.21350767 0.05015133 0.73634100
```

```
R> loglik(p)

[1] -511.5715

R> GA@iter

[1] 169
```

The GA search using unconstrained maximization yields an improved solution, and it also requires much fewer iterations.

4.8. Traveling salesperson problem

The traveling salesperson problem (TSP) is one of the most widely discussed problems in combinatorial optimization. In its simplest form, consider a set of n cities with known symmetric intra-distances, the TSP involves finding an optimal route for visiting all the cities and return to the starting point such that the distance traveled is minimized. The set of feasible solutions is given by the total number of possible routes, which is equal to $(n-1)!/2$, a value which quickly can become enormous. Several algorithms for solving the TSP have been proposed in the literature, and some of them are available in the R package **TSP** (Hahsler and Hornik 2007).

Several different representations and genetic operators for solving the TSP with GAs are available (for a review see Larranaga, Kuijpers, Murga, Inza, and Dizdarevic 1999). The most natural representation is denominated path representation. In this representation, the n cities are put in order according to a list of n elements, so that if the city i is the j-th element of the list, city i is the j-th city to be visited. For example, given 5 cities the list (B, D, A, C, E) corresponds to the tour that visits first city B, then D, etc., ending with city E.

Consider a simple example using the data on road distances (in km) between 21 cities in Europe:

```
R> data("eurodist", package = "datasets")
R> D <- as.matrix(eurodist)
```

The fitness function to be maximized can be defined as the reciprocal of the tour length. The following R code can be used to define the tour length, the fitness function, and run the GA search:

```
R> tourLength <- function(tour, distMatrix) {
+   tour <- c(tour, tour[1])
+   route <- embed(tour, 2)[,2:1]
+   sum(distMatrix[route])
+ }
R> tspFitness <- function(tour, ...) 1/tourLength(tour, ...)
R> GA <- ga(type = "permutation", fitness = tspFitness, distMatrix = D,
+    min = 1, max = attr(eurodist, "Size"), popSize = 50, maxiter = 5000,
+    run = 500, pmutation = 0.2)
R> summary(GA)
```

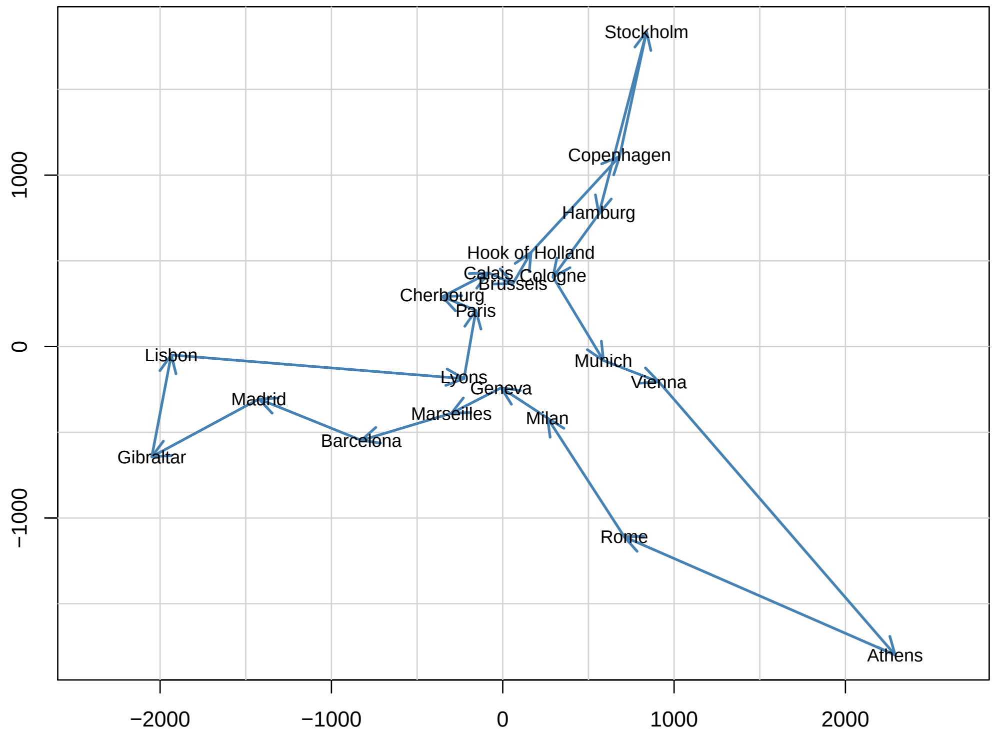


Figure 7: Map of European cities with optimal TSP tour found by GA.

```
+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  permutation
Population size       =  50
Number of generations =  5000
Elitism               =
Crossover probability =  0.8
Mutation probability  =  0.2

GA results:
Iterations             = 811
Fitness function value = 0.00007786949
Solutions              =
     x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18
[1,] 17 21  1 19 16  8 15  2 14   9  12  13  18   5   4   3  11   7
[2,]  7 11  3  4  5 18 13 12  9  14   2  15   8  16  19   1  21  17
     x19 x20 x21
[1,]  20  10   6
[2,]   6  10  20
```

The solutions found correspond to a unique path, with tour length equal to:

```
apply(GA@solution, 1, tourLength, D)
```

```
[1] 12842 12842
```

Figure 7 shows a map of cities, where the coordinates are computed from intra-distances using multidimensional scaling, and the solution path found by GA. The code to draw the graph is the following:

```
R> mds <- cmdscale(eurodist)
R> x <- mds[, 1]
R> y <- -mds[, 2]
R> plot(x, y, type = "n", asp = 1, xlab = "", ylab = "")
R> abline(h = pretty(range(x), 10), v = pretty(range(y), 10),
+    col = "light gray")
R> tour <- GA@solution[1, ]
R> tour <- c(tour, tour[1])
R> n <- length(tour)
R> arrows(x[tour[-n]], y[tour[-n]], x[tour[-1]], y[tour[-1]],
+    length = 0.15, angle = 25, col = "steelblue", lwd = 2)
R> text(x, y, labels(eurodist), cex=0.8)
```

4.9. User defined genetic operators

Several R functions are included in the **GA** package for obtaining an initial population and for applying genetic operators depending on the encoding of decision variables as described in Section 3.1. However, user defined functions can be also provided as arguments to `ga`. Suppose we would like to implement *Boltzmann selection*, an operator in which the strength of selection increases over the iterations in a manner similar to the "temperature" variable in simulated annealing. For two randomly selected individuals with fitness $f_1 > f_2$, the probability of selecting the first individual is computed as

$$p = \exp\{-(f_1 - f_2)/T\}$$

where T is the temperature, a parameter which controls the rate of selection. The temperature is high at the beginning, which means the selection pressure is low. The temperature is gradually lowered, hence the selection pressure gradually increases, thereby allowing the GA to narrow in more closely to the best part of the search space while maintaining a certain degree of diversity. Following Sivanandam and Deepa (2007, p. 49) we may define $T = T_0(1-\alpha)^k$, where $k = 1 + 100 \times \text{iter}/\text{maxiter}$. The parameter $\alpha \in [0, 1]$ controls the amount of pressure applied, with smaller values allowing for increased chance of exploring the search space for many iterations, and larger values which force the search to concentrate on the part of the search space where the current solution is located.

An R function implementing the Boltzmann selection is given by the following code:

```
R> BoltzmannSelection <- function(object, alpha = 0.2,
+    eps = gaControl(object@type)@eps, ...)
+  {
+    f <- object@fitness
+    T0 <- max(f)-min(f)
+    k <- 1 + 100 * object@iter/object@maxiter
+    T <- max(T0 * (1 - alpha)^k, eps)
+    sel <- rep(NA, object@popSize)
+    for(i in 1:object@popSize) {
+      s <- sample(1:object@popSize, size = 2)
+      p <- exp(-abs(f[s[1]]-f[s[2]])/T)
+      if(f[s[1]] > f[s[2]]) {
+        sel[i] <- if(p > runif(1)) s[2] else s[1]
+      } else {
+        sel[i] <- if(p > runif(1)) s[1] else s[2]
+      }
+    }
+    out <- list(population = object@population[sel, , drop = FALSE],
+      fitness = f[sel])
+    return(out)
+  }
```

We may now run a GA search for the Rastrigin function discussed in Section 4.2 as follows:

```
R> GA <- ga(type = "real-valued",
+    fitness =  function(x) -Rastrigin(x[1], x[2]),
+    min = c(-5.12, -5.12), max = c(5.12, 5.12),
+    popSize = 50, maxiter = 200, selection = BoltzmannSelection)
R> summary(GA)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  50
Number of generations =  200
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain
       x1    x2
Min -5.12 -5.12
Max  5.12  5.12

GA results:
```

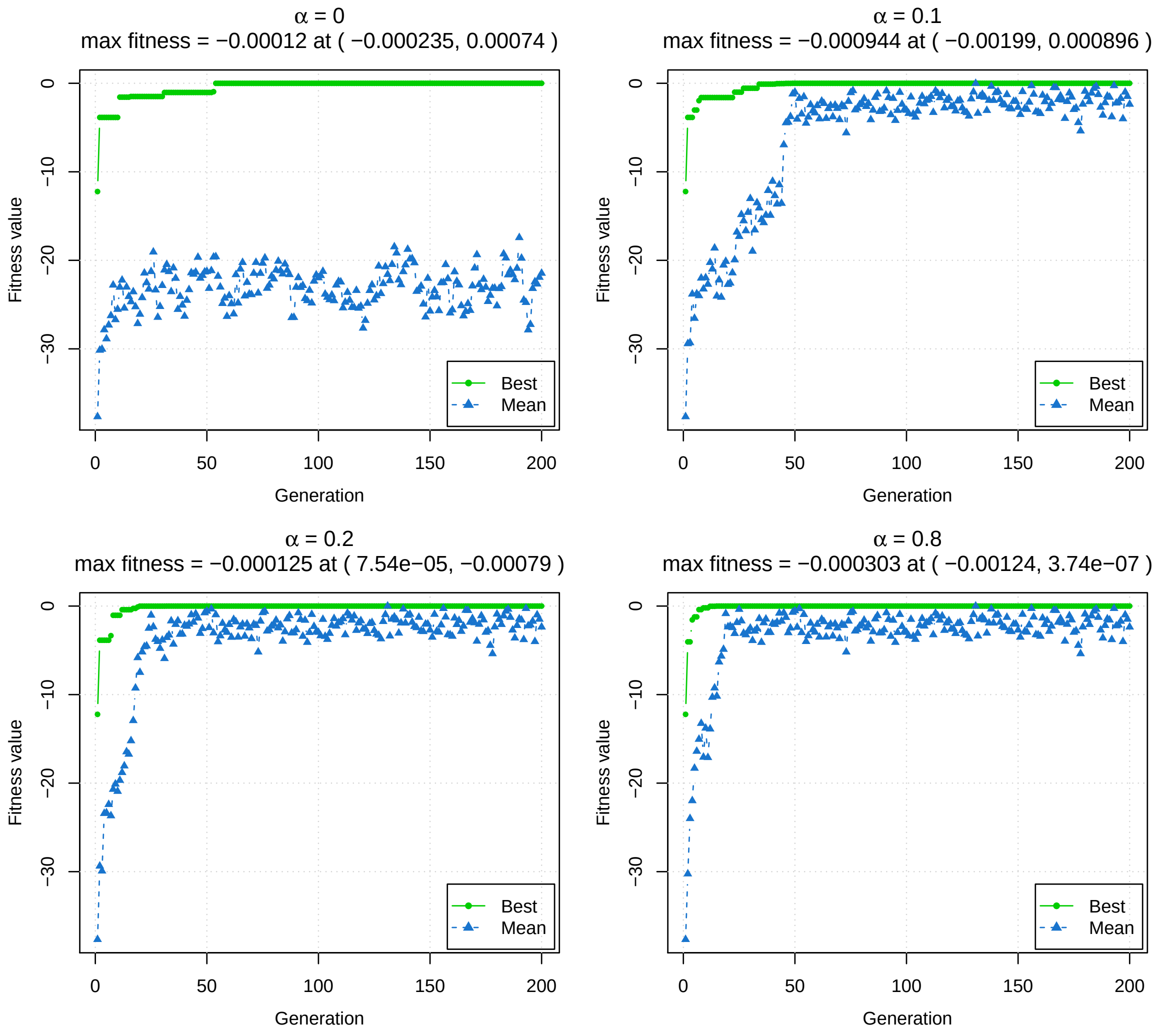


Figure 8: GA search paths using Boltzmann selection at different values of α parameter.

```
Iterations             = 200
Fitness function value = -0.0001248783
Solution               =
               x1            x2
[1,] 7.540733e-05 -0.0007897893
```

The results shown above are for the default $\alpha = 0.2$. Other values of α, say 0.8, can be obtained by appropriately specifying the `selection` argument in the `ga` function call as follows:

```
R> ga(type = "real-valued", fitness =  function(x) -Rastrigin(x[1], x[2]),
+     min = c(-5.12, -5.12), max = c(5.12, 5.12), popSize = 50, maxiter = 200,
+     selection = function(...) BoltzmannSelection(..., alpha = 0.8))
```

Figure 8 shows the search path for $\alpha = (0, 0.1, 0.2, 0.8)$. When $\alpha = 0$ there is no selection pressure and the search is basically a random walk through the search space. As α increases the selection pressure also increases with the iterations; furthermore, the pressure is higher for larger values of α. In general, for small and positive values of α the iterations required to achieve an accurate solution can increase substantially.

5. Conclusion

In this paper we discussed the R package **GA** for applying genetic algorithm methods in optimization problems. The package is flexible enough to allow users to define their own objective function to be optimized, either using built-in standard genetic operators, or by defining and exploring new operators.

According to the no-free-lunch theorem (Wolpert and Macready 1997), which roughly speaking states that there is no optimization algorithm which is uniformly better than other algorithms on average, genetic algorithms are not the panacea for all types of optimization searches. In general, GAs are slower than derivative-based algorithm. However, the latter may be unable to find any optimum at all. On the contrary, GAs can be successful when the fitness function is not smooth or there are local optima. Furthermore, their use in practical problems may serve to highlight a set of candidate solutions which, albeit not the optimal ones, could be at least worthwile to consider.

Finally, we think that the **GA** package may serve the community in providing a simple, accurate, and extensible tool for exploring the potentiality of genetic algorithms in statistical applications.

References

Affenzeller M, Winkler S (2009). *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications.* Chapman & Hall/CRC, Boca Raton.

Akaike H (1973). "Information Theory and an Extension of the Maximum Likelihood Principle." In *Second International Symposium on Information Theory*, volume 1, pp. 267–281. Académiai Kiadò, Budapest.

Andrews DF (1974). "A Robust Method for Multiple Linear Regression." *Technometrics*, **16**(4), 523–531.

Back T, Fogel DB, Michalewicz Z (2000a). *Evolutionary Computation 1: Basic Algorithms and Operators.* IOP Publishing, Bristol and Philadelphia.

Back T, Fogel DB, Michalewicz Z (2000b). *Evolutionary Computation 2: Advanced Algorithms and Operators.* IOP Publishing, Bristol and Philadelphia.

Bozdogan H (2004). "Intelligent Statistical Data Mining with Information Complexity and Genetic Algorithms." In H Bozdogan (ed.), *Statistical Data Mining and Knowledge Discovery*, pp. 15–56. Chapman & Hall/CRC, Boca Raton.

Calcagno V, de Mazancourt C (2010). "**glmulti**: An R Package for Easy Automated Model Selection with (Generalized) Linear Models." *Journal of Statistical Software*, **34**(12), 1–29. URL http://www.jstatsoft.org/v34/i12/.

Chambers JM (2008). *Software for Data Analysis: Programming with R*. Springer-Verlag, New York.

Chatterjee S, Laudato M, Lynch LA (1996). "Genetic Algorithms and Their Statistical Applications: An Introduction." *Computational Statistics & Data Analysis*, **22**, 633–651.

Coley DA (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific, Singapore.

Eiben AE, Smith JE (2003). *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin Heidelberg.

Gentleman R (2009). *R Programming for Bioinformatics*. Chapman & Hall/CRC, Boca Raton.

Goldberg D (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Boston.

Hahsler M, Hornik K (2007). "**TSP** – Infrastructure for the Traveling Salesperson Problem." *Journal of Statistical Software*, **23**(2), 1–21. URL http://www.jstatsoft.org/v23/i02/.

Haupt RL, Haupt SE (2004). *Practical Genetic Algorithms*. 2nd edition. John Wiley & Sons, New York.

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

Jones O, Maillardet R, Robinson A (2009). *Introduction to Scientific Programming and Simulation Using R*. Chapman & Hall/CRC, Boca Raton.

Lange K (2004). *Optimization*. Springer-Verlag, New York.

Larranaga P, Kuijpers CMH, Murga RH, Inza I, Dizdarevic S (1999). "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators." *Artificial Intelligence Review*, **13**(2), 129–170.

Mebane Jr WR, Sekhon JS (2011). "Genetic Optimization Using Derivatives: The **rgenoud** Package for R." *Journal of Statistical Software*, **42**(11), 1–26. URL http://www.jstatsoft.org/v42/i11/.

Montgomery DC (2009). *Introduction to Statistical Quality Control*. 6th edition. John Wiley & Sons, New York.

Mullen K, Ardia D, Gil D, Windover D, Cline J (2011). "**DEoptim**: An R Package for Global Optimization by Differential Evolution." *Journal of Statistical Software*, **40**(6), 1–26. URL http://www.jstatsoft.org/v40/i06/.

R Core Team (2012). *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/.

Satman MH (2012a). ***galts**: Genetic Algorithms and C-Steps Based LTS (Least Trimmed Squares) Estimation.* R package version 1.2, URL http://CRAN.R-project.org/package=galts.

Satman MH (2012b). ***mcga**: Machine Coded Genetic Algorithms for Real-Valued Optimization Problems.* R package version 2.0.6, URL http://CRAN.R-project.org/package=mcga.

Schilling EG, Neubauer DV (2009). *Acceptance Sampling in Quality Control.* 2nd edition. Chapman & Hall/CRC, Boca Raton.

Sivanandam SN, Deepa SN (2007). *Introduction to Genetic Algorithms.* Springer-Verlag, Berlin.

Spall JC (2003). *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control.* John Wiley & Sons, Hoboken.

Spall JC (2004). "Stochastic Optimization." In JE Gentle, W Härdle, Y Mori (eds.), *Handbook of Computational Statistics*, pp. 169–197. Springer-Verlag, Berlin.

Tendys T (2002). ***gafit**: Genetic Algorithm for Curve Fitting.* R package version 0.4, URL http://CRAN.R-project.org/src/contrib/Archive/gafit/.

Theussl S (2013). "CRAN Task View: Optimization and Mathematical Programming." Version 2013-02-14, URL http://CRAN.R-project.org/view=Optimization.

Venables WN, Ripley BD (2000). *S Programming.* Springer-Verlag, New York.

Verzani J (2005). *Using R for Introductory Statistics.* Chapman & Hall/CRC, Boca Raton.

Willighagen E (2005). ***genalg**: R-Based Genetic Algorithm.* R package version 0.1.1, URL http://CRAN.R-project.org/package=genalg.

Wolpert DH, Macready WG (1997). "No Free Lunch Theorems for Optimization." *IEEE Transactions on Evolutionary Computation*, **1**(1), 67–82.

Yu X, Gen M (2010). *Introduction to Evolutionary Algorithms.* Springer-Verlag, Berlin.

Affiliation:

Luca Scrucca
Dipartimento di Economia, Finanza e Statistica
Università degli Studi di Perugia
Via A. Pascoli, 20
06123 Perugia, Italy
E-mail: luca@stat.unipg.it
URL: http://www.stat.unipg.it/luca/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume 53, Issue 4 *Submitted:* 2011-11-29
April 2013 *Accepted:* 2012-11-16