Revista Brasileira de Computação Aplicada, November, 2019

DOI: 10.5335/rbca.v11i3.9047
Vol. 11, Nº 3, pp. 1–11
Homepage: seer.upf.br/index.php/rbca/index

ISSN 2176-6649

TUTORIAL

# Unconstrained numerical optimization using real-coded genetic algorithms: a study case using benchmark functions in R from Scratch

Omar Andres Carmona Cortes[1], and Josenildo Costa da Silva[1]

[1]Departamento de Computação (DComp) – Instituto Federal do Maranhão (IFMA)

*omar@ifma.edu.br; jcsilva@ifma.edu.br

**Received:** 2019-01-17. **Revised:** 2019-06-14. **Accepted:** 2019-08-14.

## Abstract

Unconstrained numerical problems are common in solving practical applications that, due to its nature, are usually devised by several design variables, narrowing the kind of technique or algorithm that can deal with them. An interesting way of tackling this kind of issue is to use an evolutionary algorithm named Genetic Algorithm. In this context, this work is a tutorial on using real-coded genetic algorithms for solving unconstrained numerical optimization problems. We present the theory and the implementation in R language. Five benchmarks functions (Rosenbrock, Griewank, Ackley, Schwefel, and Alpine) are used as a study case. Further, four different crossover operators (simple, arithmetical, non-uniform arithmetical, and Linear), two selection mechanisms (roulette wheel and tournament), and two mutation operators (uniform and non-uniform) are shown. Results indicate that non-uniform mutation and tournament selection tend to present better outcomes.

**Keywords**: Benchmark Functions; Genetic Algorithms; Numerical Optimization; Real-Coded; Unconstrained.

## Resumo

Problemas de otimização sem restrições são comuns em aplicações práticas e sendo estes formados normalmente por várias variáveis, limita-se o tipo de técnica ou algoritmo que pode ser utilizado para sua solução. Uma forma interessante de lidar com esse tipo de problema é através do uso de um algoritmo evolutivo chamado Algoritmo Genético. Nesse contexto, este trabalho é um tutorial sobre algoritmos genéticos em código real para solucionar problemas de otimização sem restrições, apresentando tanto a teoria quanto sua implementação em linguagem R. Cinco funções de benchmark ((Rosenbrock, Griewank, Ackley, Schwefel, and Alpine) são utilizadas como estudo de caso. Além disso, são também usados quatro diferentes operadores de cruzamento (simples, aritmético, aritmético não uniforme e linear), dois mecanismos de seleção (roleta e torneio) e dois operadores de mutação (uniforme e não uniforme). Os resultados indicam que a mutação não uniforme e o operador torneio de mutação apresentam os melhores resultados.

**Palavras-Chave**: Funções de Benchmark; Algoritmos Genéticos; Otimização Numérica; Código Real; Sem Restrições.

## 1 Introduction

Numerical Optimization problems exist widely in different areas of science research and engineering practice (Zang et al., 2018), *i.e.*, it is an essential tool in decision science and the analysis of physical systems (Nocedal and Wright, 2006). In other words, it is a tool for solving practical problems devised by many variables and no constraints, also known as unconstrained problems. Its primary purpose is to

discover the best values for design variables and/or objective functions that are not known precisely (Muc and Sanetra, 2017). In general, unconstrained problems can be classified into two categories: test functions and real-world problems. Test functions are artificial problems and can be used to evaluate the behavior of an algorithm in sometimes diverse and challenging situations (Jamil and Yang, 2013). On the other hand, real-world problems originate from different fields such as physics, chemistry, engineering, mathematics, etc. In this work, we will focus on five test functions, also known as benchmark functions. A set of real-world problems can be seen in Averick et al. (1992). Those functions have been used to test different kind of algorithms as we can see in Borges et al. (2018), Maucec and Brest (2018), Karaboğa and Kaya (2018), Cavalca and Fernandes (2018), etc.

There are several optimization techniques for solving unconstrained numerical problems. The traditional ones aim to discover the optimum solutions of continuous and differentiable functions, *i.e.*, they use analytical methods and calculus to locate the best solutions. In fact, the classical methods are fast; however, they are limited because they can only deal with unconstrained function and a small number of variables. Moreover, practical applications usually deal with non-differentiable functions. Thus, evolutionary algorithms (Eiben and Smith, 2007) appear as a viable solution for optimizing constrained and non-differentiable functions.

In this context, the idea of this work is to present a tutorial on Genetic Algorithms in Numerical Optimization using benchmark function with a study case in R language (RStudio Team, 2018). Then a question can appear: why not using an R package? The main reason is that when we use a package, such as genalg (Willighagen and Ballings, 2015), we are limited to those features offered by the package. Particularly in the genalg package, the user has no control on genetic operators whatsoever. The only control provided by the referred package is mostly concerning parameters, *i.e.*, the user cannot choose different crossover or mutation operators. Regarding the GA package (Scrucca, 2017), which is a more generic and flexible package includes other evolutionary algorithms such as Differential Evolution, present an advantage of using different genetic operators and parallel algorithms. However, having the code made from scratch, the user can quickly implement different operators or even create hybrid algorithms that are fitter to the problem being solved.

Thus, this tutorial is divided as follows: Section 2 presents some basics on numerical optimization and benchmark functions; Section 3 shows the theory of real-coded genetic algorithms and their operators; Section 4 explains important concepts in R that are essential to understand the code; Section 5 implements all GA concepts in R; Section 6 illustrates how the GA works in five benchmark functions; finally, Section 7 draws the conclusions of this tutorial.

## 2 Numerical Optimization and Benchmarks

The unconstrained optimization aims to minimize or maximize an objective function that depends on real variables, with no restrictions at all on the values of these variables (Nocedal and Wright, 2006). Mathematically, it is min or max$f(x)$, where $x \in \mathbf{R}^n$ and $n \geq 1$. Thus, a solution $x_*$ is a global solution of a minimization problem if $f(x_*) < f(x)\ \forall\ x$; analogously, it is a solution of a maximization problem if $f(x_*) > f(x)\ \forall\ x$.

Regardless of the kind of optimization, if we want to use a GA for this kind of problem, it is mandatory $n > 1$. Actually, $n$ regards to the dimensionality of the search space, which is an important factor in the problem complexity, since the higher the dimension, the higher the probability of getting trapped in a local optima (Cortes et al., 2012). A study of the dimensionality problem and its features was carried out by Friedman (1994).

Two other properties are essential in numerical optimization: separability and multi-modality. The separability concerns the possibility of dividing $f(x)$ into two or more functions. Consequently, non-separable functions are harder to optimize then separable ones. Multi-modality regards to the existence of many local optima. In this context, non-separable and multi-modal functions are harder to solve than the other ones.

We will test our code using four unconstrained continuous numerical benchmarks functions: Rosenbrock (1960), Griewank (1981), Ackley (1987), Schwefel (1981), and Rahnamyan et al. (2007) as presented in Table 1.

Table 2 presents the benchmarks properties (Separability, Modality, and Differentiability), the domain, and the global optima. The domain is a constraint for each gene, *i.e.*, the lower and upper bounds. The optimal solution is the minimum value that the benchmark can reach. The separability represents if the function is separable, *i.e.*, if the function can be split into two or more functions. In other words, a function of $p$ variables is called separable, if it can be written as a sum of $p$ functions of just one variable (Boyer et al., 2005). Finally, the modality regards to the existence of many local optima. In this context, non-separable and multi-modal functions are harder to solve than the other ones.

## 3 Real-Coded Genetic Algorithms

Fig. 1 shows the pseudo code for a GA. Firstly, the GA creates a random population, then evaluates it to select individuals undergoing genetic operators. Usually, methods such as the roulette wheel or tournament, for example, select the new population. Afterward, the crossover exchanges information (genes) between two parents based on the probability of mutation ($p_c$), creating one or more offspring. Then, the mutation operator can change zero or more genes on

**Table 1:** Unconstrained Benchmark Functions

| Name | Function |
|---|---|
| Rosenbrock | $f_1(x) = \sum_{i=1}^{n} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$ |
| Griewank | $f_2(x) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}})$ |
| Ackley | $f_3(x) = -20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - \exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)) + 20 + exp(1)$ |
| Schwefel | $f_4(x) = \sum_{i=1}^{n} -x_i \sin\sqrt{\lvert x_i \rvert}$ |
| Alpine | $f_5(x) = \sum_{i=1}^{n} \lvert x_i \sin(x_i) + 0.1x_i \rvert$ |

**Table 2:** Benchmark functions properties

| Name | Domain | Min | Separable | Multimodal | Differentiable |
|---|---|---|---|---|---|
| Rosenbrock | [-5,10] | 0 | No | No | Yes |
| Griewank | [-600,600] | 0 | No | Yes | Yes |
| Ackley | [-32,32] | 0 | No | Yes | Yes |
| Schwefel | [-500,500] | -12569.49 | Yes | Yes | Yes |
| Alpine | [0,10] | 0 | No | Yes | No |

each chromosome. Finally, if elitism is TRUE, the algorithm guarantees that the best individual remains in the population. Following subsections detail how to represent individuals, also called chromosomes, and how these operators work.

```
1  Pop = Initialize Population(LB,UB)
2  Fit = Evaluate(Pop)
3  While !(Stop Criterion)
4   Pop' = Selection(Pop)
5   Pop' = Crossover(Pop')
6   Pop' = Mutation(Pop')
7   Fit' = Evaluate(Pop')
8   If (Elitism == FALSE)
9     Pop = Pop'
10  Else
11    If (best(fit) > best(fit'))
12          Swap()
13    End-If
14  End-If
15 End-While
```

**Figure 1:** The Genetic Algorithm Pseudo Code

## 3.1 Representation

There are two main representations of genetic algorithms: binary-coded and real-coded. In the binary-code representation, an individual or chromosome, which is a possible solution to the problem being solved, is represented by a vector of $\{0,1\}$. On the other hand, as expected, real numbers devise a real-coded chromosome as presented in Fig. 2. Because we are dealing whit real numbers, each gene requires a domain constraint represented by a lower and an upper bound, respectively (*LB* and *UB*). In other words, assuming that $c_i$ is a gene within a chromosome $i$, we have $LB_i \leq c_i \leq UB_i$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 3.5 | 1.0 | 2.4 | 7.2 | 8.0 | 12.5 | 7.8 |

**Figure 2:** Example of a Real Chromosome

## 3.2 Crossover

The main purpose of the crossover operator is to exchange information (genes) between parents, creating one or more offspring. In real-coded representation, there are several ways of doing that. A thorough list of operators can be seen in Herrera et al. (1998). In this tutorial, we describe four of them: simple, arithmetical, non-uniform arithmetical, and Linear. In the simple crossover, a cut point is randomly chosen, then the offspring are formed, making a combination of parts. Considering that $p1 = c_1^1, c_2^1, \ldots, c_n^1$ and $p2 = c_1^2, c_2^2, \ldots, c_n^2$ are two parents, and $j$ is the cutting point, the first offspring is $o_1 = c_1^1, \ldots c_j^1, c_{j+1}^2, \ldots, c_n^2$, and the second one is $o_2 = c_1^2, \ldots c_j^2, c_{j+1}^1, \ldots, c_n^1$.

Eqs. (1) and (2) illustrates how to perform the arithmetical crossover for two descendants, in which $r$ is a random number in the range $[0,1]$.

$$o_i^1 = r \times c_i^1 + (1 - r)c_i^2 \tag{1}$$

$$o_i^2 = (1 - r) \times c_i^1 + (r)c_i^2 \tag{2}$$

The difference between arithmetical and non-uniform arithmetical crossover relies on the fact that the variable $r$ is not random anymore but computed by dividing the current generation $t$ by the maximum number of generations ($T_{max}$) as shown in Eq. (3).

$$r = \frac{t}{T_{max}} \tag{3}$$

Finally, the Linear Crossover creates three offspring as presented in Eqs. (4) to (6). The first one is similar to the arithmetical crossover with $r = 0.5$. The other ones explore the outer limits of $c_i^1$ and $c_i^2$, respectively.

$$o_i^1 = 0.5 * c_i^1 + 0.5 * c_i^2 \tag{4}$$

$$o_i^2 = 1.5 * c_i^1 - 0.5 * c_i^2 \quad (5)$$

$$o_i^3 = -0.5 * c_i^1 + 1.5 * c_i^2 \quad (6)$$

It is essential to observe that not every individual from the selected population undergoes mutation. Only those whose probability is less than the probability of crossover, $p_c$, participate in the operation.

### 3.3 Mutation

The mutation operator changes genes from a chromosome. Concerning real-coded individuals, the uniform mutation, also known as a random mutation, randomly replaces a gene inside its domain based on a parameter known as probability of mutation $p_m$.

Another well-known mutation in the literature is the non-uniform mutation operator, in which the selected genes are mutated according to Eqs. (7) and (8), in which $s_i$ is the gene being mutated, $t$ represents the current generation, LB and UB are the lower and the upper bound of the variable $i$, respectively, $r$ is a random number between 0 and 1, $T$ is the maximal number of generations, and $b$ is the degree of dependency (usually $b = 5$).

$$s_i^{'} = \begin{cases} s_i + \Delta(t, UB_i - s_i), & \text{if } \theta = 0 \\ s_i - \Delta(t, s_i - LB_i), & \text{if } \theta = 1 \end{cases} \quad (7)$$

$$\Delta(t, y) = y \times (1 - r^{(1-\frac{t}{T})^b}) \quad (8)$$

The non-uniform mutation is one of the operators responsible for the fine-tuning capabilities of the system (Michalewicz, 1999).

## 4 Important Concepts in R

The purpose of this chapter is not to provide a thorough vision of R programming but gives concepts that are essential to understanding the code in the next sections. Details of how to program in R can be found in Lander (2015) and Crawley (2012) books.

The first essential concept is the notion of indirect indexing. Programming languages, such as C and Java, access elements in a matrix using a specific index devised by row and column. If the programmer wants to access a set of elements, it is necessary to use a for loop and work on them element-by-element. In R, the indexes of a matrix can also be a matrix. For example, suppose that we have to replace some elements obeys a condition by a random number in a matrix *Mat*, Fig. 3 illustrates how to perform this task. The *which()* function returns a matrix containing two columns (row and column) of the elements that satisfy the *condition*. Then, a vector operation (line 2) replaces the corresponding elements. We have to note that the number of elements created by the *runif()* function has to be the same number of rows in *idx*, therefore, we have to use the *nrow()* function. Fig. 4 illustrates how the operation works.

```
1 idx <- which(condition, arr.ind = TRUE)
2 Mat[idx] <- runif(nrow(idx))
```

**Figure 3:** Indirect indexing example

The other concept we have to know how to deal with is the logical indexing. The concept is quite similar to the indirect indexing; however, in this case, all indexes are logical values. Let us suppose that we have a matrix *Mat* of integer numbers randomly created as presented in Fig. 5. Then we want to replace all values less than 10 with the lower bound 10. The instruction $idx < -Mat < 10$ returns a matrix in which all positions obeys the condition are true. Afterward, all values are replaced. This operation makes things easier when the domain is the same for all genes, which is common in numerical optimization. On the other hand, if the domain is different for each gene, then we have to use the instruction $which(idx, arr.ind = TRUE)$ to locate the positions that fulfill the condition.

The third important concept we have to tackle is called *group operations* or *group functions*. This kind of instruction, as the name suggests, executes in a group of data. Moreover, we preferably execute group operations instead of for loops because the first one is usually faster than loops. The first set of group functions is: *sum*(), *mean*(), and *sd*(). These functions receive a vector or a matrix as a parameter and return the sum, the mean, and the standard deviation of the entrance data, respectively. If the parameter is a matrix and ones wants to perform row or column based operations, we have to use, for example, the functions *rowSum*() or *colSum*().

On the other hand, if we want to perform row or column-based operations using a pre-existed function, we have to use the function *apply*(), which the syntax is $apply(obj, margin, function, parameters)$, where *obj* is a data structure, usually, a matrix, *margin* sets the kind of operation (1 - row-based, 2 - column based), *function* is an implemented function, and *parameters* are the parameters required by the implemented function. An indispensable extension of this function is the *lapply*() function in which *obj* has to be a list. Next, we present some general remarks about R that are important to the correct implementation of GAs.

**Remarks**

- Vectors and matrices start with 1 instead of 0;
- Operations between matrices are element-wise. If a traditional multiplication matrix is required, we have to use the symbol %*%;
- Avoid for loops;

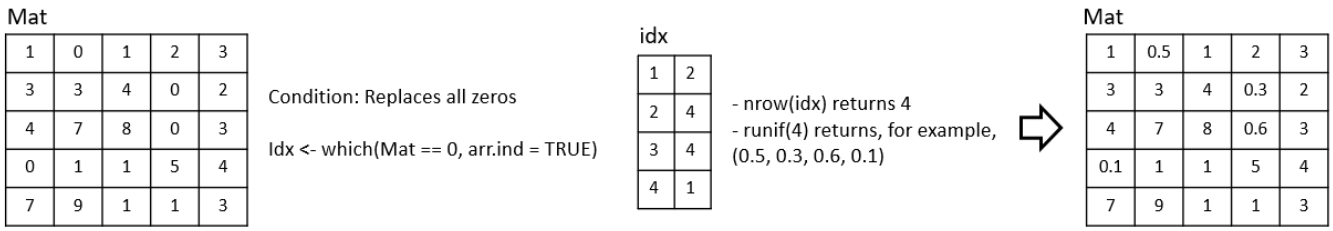


**Figure 4:** Indirect indexing

```
1 Mat <- matrix(sample(25), nrow = 5)
2 idx <- Mat < 10
3 Mat[idx] <- 10
```

**Figure 5:** Logical indexing example

- The "not" operator, represented by "!", also works with logical indexing;
- If indexes are numbers, *Mat*[!*idx*] for instance, does not work. Instead, we have to use *Mat*[−*idx*];
- Lists are data structures formed by different kind of objects. A list can contain, for example, a matrix, a vector, a function, and an integer value at the same time in the same data structure;
- If you do not name the elements of a list, you have to use double brackets [[*element*]] to access them;
- The symbol # precedes a comment;
- If you receive a warning after the execution of your code, probably you are assigning vectors with different sizes. It is essential to correct all warning to avoid wrong results.

# 5 GA Implementation

## 5.1 Initializing

Let us start implementing each function separately in the following order: initialize population, selection, crossover, and mutation. Thus, Fig. 6 shows the code that initializes the population, in which each gene is within the domain $[lb, ub]$. Then, we evaluate the population using the *apply()* function that receives the objective function as a parameter. Finally, the initialization function returns a list containing the first population and the fitness for each chromosome, corresponding to lines 1 and 2 from Fig. 1.

```
1 init.population <- function(func,lb,ub,pop.size,dimension){
2  pop <- matrix(runif(pop.size*dimension),nrow=pop.size)
3  fitness <- rep(NA,pop.size)
4  pop <- lb + pop*(ub-lb)
5  fitness <- apply(pop,1,func)
6  return(list(pop = pop, fit = fitness))
7 }
```

**Figure 6:** Function for initializing population

## 5.2 Selection

The next step is the selection method that chooses which chromosomes will try to participate in the crossover stage. Two selection mechanisms, roulette wheel and tournament, are presented in Figs. 7 and 8, respectively.

```
1 roulette.wheel <- function(pop, fitness, pop.size, dim){
2  new.pop <- matrix(rep(NA, pop.size*dim), nrow=pop.size)
3  new.fit <- rep(NA,pop.size)
4  F <- sum(fitness)
5  p  <- -fitness/F
6  q <- cumsum(p)
7  r <- runif(pop.size)
8  for (i in 1:pop.size){
9   if (r[i] < q[1]){
10   new.pop[i,] <- pop[1,]
11  }
12  else{
13   idx <- which.min(q < r[i])
14   new.pop[i,] <- pop[idx,]
15  }
16 }
17 return(new.pop)
18 }
```

**Figure 7:** Roulette wheel function

In the roulette function, the variables *new.pop* and *new.fit* will contain the new population and its fitness, respectively. Then *q* will contain the cumulative probability matrix that is the wheel. Afterward, the vector *r* will contain one random number in the domain $[0, 1]$ for each chromosome, and the *for* loop will check which one will form the temporary population undergoes the crossover process.

```
1 tournament <- function(pop,fitness,pop.size,dim,t.size = 4){
2  new.pop <- matrix(rep(NA,pop.size*dim), nrow=pop.size)
3  for(i in 1:pop.size){
4   idx <- sample(1:pop.size, t.size)
5   pos <- which.min(fitness[idx])
6   new.pop[i,] <- pop[idx[pos],]
7  }
8  return(new.pop)
9 }
```

**Figure 8:** Tournament function

In the selection by tournament method, the variable *t.size* is responsible for setting the tournament size, *i.e.*,

how many individuals will participate in each round. Then the winner of each round is chosen to undergo mutation. It is a simple but effective form of selection.

## 5.3 Crossover

The next step in the GA algorithm is to apply the crossover operation on the new population. Fig. 9 shows the code for this task. The main parameters of this function are the selected population and *pc* (probability of crossover). Why is *pc* so important? Because not every single chromosome will undergo crossover, only those ones which a random number *r* is less than *pc*, *i.e.*, *r* < *pc*. Lines 2 to 5 perform this selection. Then the *for* loop produces two offspring using the cross point as a divisor, returning only the population because the evaluation is necessary only after mutation.

```
1  simple.cross <- function(lb,ub,pop.size,dimension,pop,pc){
2   new.pop <- matrix(rep(NA,pop.size*dimension),
3                     nrow = pop.size)
4   r <- runif(pop.size)
5   idx <- (r < pc)
6   pop <- pop[idx,]
7   tmp.pop.size <- nrow(pop)
8   for(i in seq(1,pop.size, by = 2)){
9    indivs <- sample(1:tmp.pop.size,2,replace = FALSE)
10   cross.point  <- sample(2:(dimension-1),1)
11   offspring1   <- c(pop[indivs[1],1:cross.point],
12   pop[indivs[2],(cross.point+1):dimension])
13   offspring2   <- c(pop[indivs[2],1:cross.point],
14   pop[indivs[1],(cross.point+1):dimension])
15   new.pop[i,]   <- offspring1
16   new.pop[i+1,] <- offspring2
17  }
18 return(pop = new.pop)
19 }
```

**Figure 9:** Simple crossover

The arithmetical crossover, also known as uniform arithmetical crossover, is an interesting crossover method because it explores the search space between two genes and does not create invalid individuals, *i.e.*, all genes are within their domain. Fig. 10 presents the implementation in which we can see that *l* controls how further from a gene the crossover goes, i.e., closer to the gene of the first or second parent.

The arithmetical crossover, also known as a uniform arithmetical crossover, is an interesting crossover method because it explores the search space between two genes and does not create invalid individuals, *i.e.*, all genes are within their domain. Fig. 10 presents the implementation in which we can see that *l* controls how further from a gene the crossover goes, i.e., closer to the gene of the first or second parent.

Finally, linear crossover generates three individuals as previously explained; however, only the best two will form the next population. Also, because of the Eqs. (5) and (6), we have to control the boundaries of the new individuals. Thus, the instruction *nrow*(*idx*) > 0 means that a gene is out of its limits. After correcting the chromosome boundaries, the concern is how to select the best two chromosomes elegantly. The answer is

```
1  arith.crossover <- function(lb,ub,pop.size,dimension,
2                              pop,pc){
3   new.pop <- matrix(rep(NA,pop.size*dimension),
4                     nrow = pop.size)
5   r <- runif(pop.size)
6   idx <- (r < pc)
7   pop <- pop[idx,]
8   tmp.pop.size <- nrow(pop)
9   for(i in seq(1,pop.size, by = 2)){
10   l = runif(1)
11   indivs <- sample(1:tmp.pop.size,2,replace = FALSE)
12   new.pop[i,] <- l*pop[indivs[1],] + (1-l)*pop[indivs[2],]
13   new.pop[i+1,] <- (l-1)*pop[indivs[1],] + l*pop[indivs[2],]
14  }
15  return(pop = new.pop)
16 }
```

**Figure 10:** Arithmetical crossover

```
1  n.arith.crossover <- function(lb,ub,pop.size,dimension
2                                pop,pc,max.it,t){
3   new.pop <- matrix(rep(NA,pop.size*dimension),
4                     nrow = pop.size)
5   r <- runif(pop.size)
6   idx <- (r < pc)
7   pop <- pop[idx,]
8   tmp.pop.size <- nrow(pop)
9   l = t/max.it
10  for(i in seq(1,pop.size, by = 2)){
11   r = runif(1)
12   indivs <- sample(1:tmp.pop.size,2,replace = FALSE)
13   new.pop[i,] <- l*pop[indivs[1],] + (1-l)*pop[indivs[2],]
14   new.pop[i+1,] <- (l-1)*pop[indivs[1],] + l*pop[indivs[2],]
15  }
16  return(pop = new.pop)
17 }
```

**Figure 11:** Non-Uniform Arithmetical crossover

to sort out together the new fitness and the respective chromosomes, choosing then the two smaller ones.

```
1  linear.crossover <- function(lb,ub,pop.size,dimension,
2                               pop,pc,func){
3   new.indiv <- matrix(rep(NA, 3*dimension), nrow = 3)
4   new.fit <- rep(NA,3)
5   new.pop <- matrix(rep(NA,pop.size*dimension), nrow =
6                     pop.size)
7   r <- runif(pop.size)
8   idx <- (r < pc)
9   pop <- pop[idx,]
10  tmp.pop.size <- nrow(pop)
11  for(i in seq(1,pop.size, by = 2)){
12   indivs <- sample(1:tmp.pop.size,2,replace = FALSE)
13   new.indiv[1,] <- 0.5*pop[indivs[1],] + 0.5*pop[indivs[2],]
14   new.indiv[2,] <- 1.5*pop[indivs[1],] - 0.5*pop[indivs[2],]
15   new.indiv[3,] <- -0.5*pop[indivs[1],] + 1.5*pop[indivs[2],]
16   idx <- which(new.indiv < lb, arr.ind = TRUE)
17   if (nrow(idx) > 0)
18    new.indiv[idx] <- lb[idx[,2]]
19   idx <- which(new.indiv > ub, arr.ind = TRUE)
20   if (nrow(idx) > 0)
21    new.indiv[idx] <- ub[idx[,2]]
22   new.fit <- apply(new.indiv,1,func)
23   tmp <- cbind(new.fit,new.indiv)
24   tmp <- tmp[order(tmp[,1]),]
25   new.pop[i,] <- tmp[1,2:(dimension+1)]
26   new.pop[i+1,] <- tmp[2,2:(dimension+1)]
27  }
28  return(pop = new.pop)
29 }
```

**Figure 12:** Linear crossover

## 5.4 Mutation

Fig. 13 shows the listing for the mutation operator that receives two essential parameters: the new population and the probability of mutation *pm*. Then, line 4 identifies which genes, row, and column, will undergo mutation in the whole population. Finally, line 7 evaluates the new population.

```
1  uniform.mut <- function(func,lb,ub,pop,pop.size,
2                          dimension,pm){
3   r <- matrix(runif(pop.size*dimension),nrow=pop.size)
4   fitness <- rep(NA,pop.size)
5   idx <- which(r < pm,arr.ind=TRUE)
6   pop[idx] <- lb[idx[,2]] + runif(nrow(idx)) *
7   (ub[idx[,2]] - lb[idx[,2]])
8   fitness <- apply(pop,1,func)
9   return(list(pop = pop, fit = fitness))
10 }
```

**Figure 13:** Uniform mutation

The next mutation operator is the non-uniform mutation, which is based on the number of iterations or generations since as iterations go on the mutation size goes down. Fig. 14 shows the mutation operator in R, in which we have to expand the boundaries using the variables *tmp.lb* and *tmp.ub*, because these vectors can be larger than the domain vectors *lb* and *ub*. Then we use the logical indexing to decide which equation to use ($\theta = 0$ or $\theta = 1$ from Eq. (7)) in order to perform the mutation.

```
1  n.uniform.mutation <- function(func,lb,ub,pop,pop.size,dimension,
2            pm,max.it,it){
3   r <- matrix(runif(pop.size*dimension),nrow=pop.size)
4   fitness <- rep(NA,pop.size)
5   idx <- which(r < pm,arr.ind=TRUE)
6   values <- pop[idx]
7   r <- sample(c(0,1), nrow(idx), replace = TRUE)
8   tmp.idx <- r == 0
9   tmp.lb <- lb[idx[,2]]
10  tmp.ub <- ub[idx[,2]]
11  values[tmp.idx] <-    values[tmp.idx] + delta(it,max.it,
12                        tmp.ub[tmp.idx]-values[tmp.idx])
13  values[!tmp.idx] <- values[!tmp.idx] - delta(it,max.it,
14                      values[!tmp.idx]-tmp.lb[!tmp.idx])
15  pop[idx] <- values
16  fitness <- apply(pop,1,func)
17  return(list(pop = pop, fit = fitness))
18 }
19
20 delta <- function(it, max.it,y, b=5){
21  r <- runif(length(y))
22  y <- y * (1 - r^((1-it/max.it)^b))
23  return(y)
24 }
```

**Figure 14:** Non-Uniform mutation

## 5.5 Main Function and Elitism

Now, it is time to get all functions together as presented in Fig. 15. As we can see, the code is pretty similar to the algorithm presented previously in Fig. 1. To test other operators, we only need to use the respective functions replacing the symbol "#". Moreover, if someone wants to test everything at once, it is trivial to add control structures to execute all operators.

```
1  GA <- function(func, lb, ub, pop.size = 10,
2                 dimension = 10, max.it = 100,
3                 pc = 0.6, pm = 0.005, sel, t.size = 4,
4                 elitism = TRUE){
5   tmp.pop <- matrix(rep(NA,pop.size*dimension),
6                     nrow = pop.size)
7   init <- init.population(func,lb,ub,pop.size,dimension)
8   pop <- init$pop
9   fitness <- init$fit
10  for(it in 1:max.it){
11   tmp.pop <- selection(pop,pop.size,dimension,fitness,
12                        sel,t.size)
13   tmp.pop <- simple.crossover(lb, ub, pop.size, dimension,
14                               tmp.pop, pc)
15   #tmp.pop <- arith.crossover(lb, ub, pop.size, dimension,
16   #                           tmp.pop, pc)
17   #tmp.pop <- n.arith.crossover(lb, ub, pop.size, dimension,
18   #                            tmp.pop, pc, max.it,i)
19   #tmp.pop <- linear.crossover(lb, ub, pop.size, dimension,
20   #                           tmp.pop, pc, func)
21   #tmp.pop <- uniform.mutation(func,lb,ub,tmp.pop,pop.size,
22   #                           dimension, pm)
23   tmp.pop <- n.uniform.mutation(func,lb,ub,tmp.pop,pop.size,
24                                 dimension,pm,max.it,it)
25   tmp.fitness <- tmp.pop$fit
26   tmp.pop <- tmp.pop$pop
27   if (elitism == TRUE){
28     best.tmp <- min(tmp.fitness)
29     best.old <- min(fitness)
30     if (best.old < best.tmp){
31      idx <- which.min(fitness)
32      idx.worst <- which.max(tmp.fitness)
33      tmp.pop[idx.worst,] <- pop[idx,]
34      tmp.fitness[idx.worst] <- fitness[idx]
35     }
36   }
37   fitness <- tmp.fitness
38   pop <- tmp.pop
39  }
40  return(list(pop = pop, fit = fitness))
41 }
```

**Figure 15:** Main function

Regarding the elitism, having the final population, we have to identify whether the elitism is set or not. If so, we have to check in which population the best individual is. In case the best individual is in the new population, the current one replaces the old one. Otherwise, we swap the worst individual from the current population by the best one from the previous one; thus, we guarantee that the best individual will always be in the population.

## 5.6 Benchmarks

Those benchmarks previously presented in Section 2 are implemented in Fig. 16. It is important to remark that we did not use any loop, as suggested in Section 4. Even though some R packages implement these functions, the main idea is to program everything from scratch.

# 6 Experiments

All experiments have been conducted in a Windows 10, 64 bits, 16 GB of RAM, 500 GB of SSD, R version

```
Rosenbrock <- function(x){
 x1 <- x[1:length(x)-1]
 x2 <- x[2:length(x)]
 return(sum(100 * (x1 - x2^2)^2 + (x1-1)^2))
}

Griewank <- function(x){
 i <- sqrt(1:length(x))
 return(1/4000 * sum(x^2)-prod(cos(x)/i))
}

Ackley <- function(x){
 n <- length(x)
 return(-20 * exp(-0.2 * sqrt(sum(x^2/n)))
  - exp(1/n*sum(cos(2*pi*x))) + 20 + exp(1))
}

Schwefel <- function(x){
 return(sum(x^2-10*cos(2*pi*x)+10))
}

Alpine <- function(x){
 return(sum(abs(x*sin(x)+0.1*x)))
}
```

**Figure 16:** Benchmark functions

3.3.3 (Project, 2018), and RStudio 1.0.136 (RStudio Team, 2018). The GA configuration is: $p_c$ = 0.6, $p_m$ = 0.01, *populatio*$_s$*ize* = 50, and *dimension* = 30. In the case of tournament selection, we use four individuals in the competition. Further, we set a seed for random numbers; thus, the result of the experiment will be the same in all computers in terms of the quality of solutions. Furthermore, all experiments have been performed using 30 trials; then, we are able to present the best result, the mean, and the standard deviation.

Fig. 17 presents the main script used for testing the code. Firstly, we load all functions presented previously utilizing the instruction *source*(). The file 'crossover.R', for instance, contains all crossover operators. The other R files follow the same principle. Either, we automatized the test only for the benchmark functions, which are stored in a vector of lists. Changing the code for executing all operators or adding new operators will be a trivial task.

Tables 3 to 6 show the results for 1000 iterations with uniform mutation and roulette wheel, uniform mutation and tournament, non-uniform mutation and roulette wheel, and non-uniform mutation and tournament, respectively. As we can see, the best combinations for those parameters is the non-uniform mutation and tournament selection, possibly because of the fine-tuning capability of the non-uniform mutation. On the other hand, Schwefel function presented the best combination using uniform mutation and tournament. Other combinations using arithmetical crossover presented good results in Griewank, Ackley, and Alpine functions; however, they are not the best ones.

## 7 Conclusions

This tutorial presented how to implement Genetic Algorithms to solve unconstrained numerical optimization problems in R from scratch. As we could see, the code is concise and straightforward,

```
source('GA.R')
source('init.population.R')
source('crossover.R')
source('mutation.R')
source('selection.R')
source('Benchmarks.R')

set.seed(123)
it <- 1000
pop.size <- 50
funcs <- c(Rosenbrock,Griewank,Ackley,Schwefel, Alpine)
bounds <- matrix(rep(NA,2*length(funcs)), nrow = 2)
res <- matrix(rep(NA,3*length(funcs)),nrow = 3)
bounds[1,] <- c(-5,-600,-32,-500,0)
bounds[2,] <- c(10,600,32,500,10)
dim <- 30
execs <- 30

result <-  vector("list",execs)
best <- rep(NA,execs)

cat("Running...\n")
for(f in 1:length(funcs)){
 lb <- rep(bounds[1,f],dim)
 ub <- rep(bounds[2,f],dim)
 for(i in 1:execs){
   result[[i]] <- GA(funcs[[f]], lb, ub, pop.size = pop.size,
                     dim = dim,
   max.it = it, pc = 0.6, pm = 0.01, sel = 1, elitism = TRUE)
   best[i] <- min(result[[i]]$fit)
 }
 res[1,f] <- min(best)
 res[2,f] <- mean(best)
 res[3,f] <- sd(best)
}
View(res)
```

**Figure 17:** Testing

allowing anyone to implement new GA variations or even hybrid algorithms. We explore the ability to perform vectorial operations and group functions in R. Either, we implemented the GA code to be easily extended to functions in which all genes have different domains in the same individual. Implementing the code using a single scalar to control the boundaries of each gene could simplify parts of the code by using logical indexing; however, extend it for multiples domains in the same chromosome would be much harder.

We expect that this tutorial is an incentive to those who want to explore the numerical optimization capability of GAs and/or to those who crave to enter in the R language world. In this context, we also incentive all interested people to implement different operators or apply the presented code to different kind of numerical problems, especially those involving the optimization of real-world problems.

## Acknowledgment

All codes are available on GitLab in the address https://gitlab.com/omar.carmona/real-coded-genetic-algorithm.

## References

Ackley, D. H. (1987). *A connectionist machine for genetic hillclimbing*, PhD thesis, Kluwer Academic Publishers, Boston – MA – USA.

**Table 3:** Results of the optimization of the five benchmarks functions using the following parameters: it = 1000, uniform mutation, roulette wheel

| | | Simple | Arithmetical | NU-Arithmetical | Linear |
|---|---|---|---|---|---|
| Rosenbrock | Best | 2451.071 | 28.8555 | 1799.662 | 1981.851 |
| | Mean | 24845.126 | 28.9317 | 62775.681 | 20396.216 |
| | Std.Dev | 24138.263 | 0.0277 | 108947.817 | 19196.078 |
| Griewank | Best | 17.7753 | 0.0000 | 1.3368 | 15.7905 |
| | Mean | 51.5129 | 0.0000 | 14.2834 | 57.9918 |
| | Std.Dev | 21.3925 | 0.0000 | 12.7659 | 36.948 |
| Ackley | Best | 11.3481 | 0.0000 | 6.1106 | 10.5472 |
| | Mean | 13.9118 | 0.0000 | 12.1561 | 13.2214 |
| | Std.Dev | 1.6119 | 0.0000 | 3.9350 | 1.3538 |
| Schwefel | Best | -10466.583 | -6595.6807 | -10990.016 | -10514.9026 |
| | Mean | -9300.963 | -5299.0428 | -6991.304 | -9332.6928 |
| | Std.Dev | 814.062 | 718.9849 | 1673.484 | 715.1635 |
| Alpine | Best | 6.0369 | 0.0000 | 14.629 | 5.9545 |
| | Mean | 12.4687 | 0.0000 | 21.7172 | 11.0664 |
| | Std.Dev | 4.1682 | 0.0000 | 4.3794 | 3.4686 |

**Table 4:** Results of the optimization of the five benchmarks functions using the following parameters: it = 1000, uniform mutation,tournament

| | | Simple | Arithmetical | NU-Arithmetical | Linear |
|---|---|---|---|---|---|
| Rosenbrock | Best | 29.9325 | 28.7685 | 28.8501 | 14.3187 |
| | Mean | 114.5754 | 28.8765 | 29.6603 | 71.867 |
| | Std.Dev | 53.6304 | 0.0369 | 2.9696 | 47.0512 |
| Griewank | Best | 0.0196 | 0.0000 | 0.0000 | 0.0008 |
| | Mean | 0.0613 | 0.0000 | 0.0004 | 0.0068 |
| | Std.Dev | 0.0247 | 0.0000 | 0.0023 | 0.0052 |
| Ackley | Best | 0.6155 | 0.0000 | 0.0000 | 0.0521 |
| | Mean | 1.2127 | 0.0000 | 0.0243 | 0.2132 |
| | Std.Dev | 0.3100 | 0.0000 | 0.1115 | 0.1280 |
| Schwefel | Best | **-12561.7277** | -11405.9518 | -12517.9301 | -12493.683 |
| | Mean | **-12549.7999** | -10630.1371 | -12375.2697 | -11776.073 |
| | Std.Dev | 7.0971 | 462.8211 | 127.6128 | 295.952 |
| Alpine | Best | 0.0652 | 0.0000 | 0.0000 | 0.0000 |
| | Mean | 0.1221 | 3.02924e-316 | 0.5505 | 0.0000 |
| | Std.Dev | 0.0354 | 0.0000 | 0.3037 | 0.0001 |

**Table 5:** Results of the optimization of the five benchmarks functions using the following parameters: it = 1000, non-uniform mutation, roulette wheel

| | | Simple | Arithmetical | NU-Arithmetical | Linear |
|---|---|---|---|---|---|
| Rosenbrock | Best | 7328.555 | 28.7887 | 189.7441 | 23308.46 |
| | Mean | 99065.147 | 28.8885 | 522987.7591 | 97250.44 |
| | Std.Dev | 80313.989 | 0.0409 | 647674.0747 | 64585.29 |
| Griewank | Best | 39.1963 | 0.0000 | 0.1928 | 42.0673 |
| | Mean | 115.8443 | 0.0000 | 15.9252 | 128.9577 |
| | Std.Dev | 54.6358 | 0.0000 | 35.3703 | 58.9478 |
| Ackley | Best | 10.8985 | 0.0000 | 1.9198 | 12.9310 |
| | Mean | 16.1601 | 0.0000 | 15.8141 | 16.3923 |
| | Std.Dev | 1.9112 | 0.0000 | 5.7599 | 1.6873 |
| Schwefel | Best | -8711.5960 | -4844.5211 | -9238.946 | -8424.8316 |
| | Mean | -6873.0373 | -3817.1849 | -5272.697 | -6758.1992 |
| | Std.Dev | 737.8165 | 579.3688 | 1393.787 | 951.6584 |
| Alpine | Best | 10.0562 | 0.0000 | 19.4486 | 11.5562 |
| | Mean | 20.6185 | 0.0000 | 30.9157 | 21.3603 |
| | Std.Dev | 6.4831 | 0.0002 | 8.6855 | 5.0837 |

Averick, B. M., Carter, R. G., Moré, J. J. and Xue, G. L. (1992). The minipack-2 test problem collection. Available at http://ftp.mcs.anl.gov/pub/tech_reports/reports/P153.pdf.

Borges, H. P., Cortes, O. A. C. and Vieira, D. (2018). An adaptive metaheuristic for unconstrained multimodal numerical optimization, *in* P. Korošec, N. Melab and E.-G. Talbi (eds), *Bioinspired Optimization Methods and Their Applications*, Springer International Publishing, Cham, pp. 26–37. https://doi.org/10.1007/978-3-319-91641-5_3.

**Table 6:** Results of the optimization of the five benchmarks functions using the following parameters: it = 1000, non-uniform mutation, tournament

| | | Simple | Arithmetical | NU-Arithmetical | Linear |
|---|---|---|---|---|---|
| Rosenbrock | Best | **17.2019** | **28.6645** | **27.7665** | **0.0559** |
| | Mean | **95.5899** | **28.6951** | **34.0667** | **39.5028** |
| | Std.Dev | 63.1179 | 0.0114 | 20.3872 | 45.8229 |
| Griewank | Best | **0.0000** | **0.0000** | **0.0000** | **0.0000** |
| | Mean | **0.0000** | **0.0000** | **0.0000** | **0.0000** |
| | Std.Dev | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| Ackley | Best | **0.0001** | **0.0000** | **0.0000** | **0.0000** |
| | Mean | **0.0002** | **0.0000** | **0.0000** | **0.0000** |
| | Std.Dev | 0.0001 | 0.0000 | 0.0001 | 0.0000 |
| Schwefel | Best | -12451.0483 | -10073.945 | -11621.6337 | -10551.4837 |
| | Mean | -11914.1278 | -9036.5918 | -10910.802 | -9576.9437 |
| | Std.Dev | 272.4585 | 669.6221 | 341.0773 | 644.8203 |
| Alpine | Best | 0.0011 | **0.0000** | 0.0000 | **0.0000** |
| | Mean | 0.0024 | **0.0000** | 0.4728 | **0.0000** |
| | Std.Dev | 0.0012 | **0.0000** | 0.4105 | **0.0000** |


Boyer, D. O., Martfnez, C. H. and Pedrajas, N. G. (2005). Cixl2: A crossover operator for evolutionary algorithms based on population features, *Journal of Artificial Intelligence Research* **24**: 1–48. https://doi.org/10.1613/jair.1660.

Cavalca, D. L. and Fernandes, R. A. S. (2018). Gradient-based mechanism for pso algorithm: A comparative study on numerical benchmarks, *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–7. https://doi.org/10.1109/CEC.2018.8477798.

Cortes, O. A. C., Rau-Chaplin, A. and Lopes, R. F. (2012). A pso-based algorithm with local search for multimodal optimization without constraints, *XXXVIII Conferencia Latinoamericana En Informatica (CLEI)*, pp. 1–7.

Crawley, M. J. (ed.) (2012). *The R Book*, 2 edn, Wiley.

Eiben, A. E. and Smith, J. E. (eds) (2007). *Introduction to Evolutionary Computing*, 2nd printing edn, Springer, New York.

Friedman, J. H. (1994). An overview of predictive learning and function approximation, *in* V. Cherkassky, J. H. Friedman and H. Wechsler (eds), *From Statistics to Neural Networks*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–61. https://doi.org/10.1007/978-3-642-79119-2_1.

Griewank, A. O. (1981). Generalized descent for global optimization, *Journal of Optimization* **34**(1): 11–39. https://doi.org/10.1007/BF00933356.

Herrera, F., Lozano, M. and Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis, *Artificial Intelligence Review* **12**(4): 265–319. https://doi.org/10.1023/A:1006504901164.

Jamil, M. and Yang, X.-S. (2013). A literature survey of benchmark functions for global optimisation problems, *International Journal of Mathematical Modelling and Numerical Optimisation* **4**(2): 1–47. https://doi.org/10.1504/IJMMNO.2013.055204.

Karaboğa, D. and Kaya, E. (2018). Evaluation of performance of adaptive and hybrid abc (aabc) algorithm in solution of numerical optimization problems, *2018 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pp. 1–5. https://doi.org/10.1109/ASYU.2018.8554009.

Lander, J. P. (ed.) (2015). *R for Everyone: Advanced Analytics and Graphics*, Addison-Wesley Professional.

Maucec, M. S. and Brest, J. (2018). A review of the recent use of differential evolution for large-scale global optimization: An analysis of selected algorithms on the cec 2013 lsgo benchmark suite, *Swarm and Evolutionary Computation* . https://doi.org/10.1016/j.swevo.2018.08.005.

Michalewicz, Z. (ed.) (1999). *Genetic Algorithms + Data Structures = Evolution Programs*, 3 edn, Springer, New York.

Muc, A. and Sanetra, I. (2017). The effectiveness of optimization algorithms in shape and topology discrete optimisation of 2-d composite structures, *2017 7th International Conference on Modeling, Simulation, and Applied Optimization (ICMSAO)*, pp. 1–5. https://doi.org/10.1109/ICMSAO.2017.7934871.

Nocedal, J. and Wright, S. (eds) (2006). *Numerical Optimization*, Springer, New York - USA.

Project, R. (2018). The R project for statistical computing. Available at http://www.r-project.org/.

Rahnamyan, S., Tizhoosh, H. R. and Salama, N. M. M. (2007). A novel population initialization method for accelerating evolutionary algorithms, *Computers and Mathematics with Applications* **53**(10): 1605–1614. https://doi.org/10.1016/j.camwa.2006.07.013.

Rosenbrock, H. H. (1960). An automatic method for finding the greatest or least value of a function, *Computer Journal* **3**(3): 175–184. https://doi.org/10.1093/comjnl/3.3.175.

RStudio Team (2018). Rstudio: Integrated development environment for R. Available at http://www.rstudio.com/.

Schwefel, H.-P. (ed.) (1981). *Numerical optimization of computer models*, Wiley.

Scrucca, L. (2017). On some extensions to ga package: hybrid optimisation, parallelisation and islands evolution., *The R Journal* **1**(9): 187–206. Available at https://journal.r-project.org/archive/2017/RJ-2017-008.

Willighagen, E. and Ballings, M. (2015). *genalg: R Based Genetic Algorithm. R package version 0.2.0*. Available at https://CRAN.R-project.org/package=genalg.

Zang, W., Ren, L., Zhang, W. and Liu, X. (2018). A cloud model based dna genetic algorithm for numerical optimization problems, *Future Generation Computer Systems* **81**: 465–477. https://doi.org/10.1016/j.future.2017.07.036.