ecr 2.0: A Modular Framework for Evolutionary Computation in R

Jakob Bossek
University of Münster
Leonardo-Campus 3
Münster, Germany 48149
bossek@uni-muenster.de

ABSTRACT

The novel R package `ecr` (version 2), short for Evolutionary Computation in R, provides a comprehensive collection of building blocks for constructing powerful evolutionary algorithms for single- and multi-objective continuous and combinatorial optimization problems. It allows to solve standard optimization tasks with few lines of code using a black-box approach. Moreover, rapid prototyping of non-standard ideas is possible via an explicit, white-box approach. This paper describes the design principles of the package and gives some introductory examples on how to use the package in practise.

CCS CONCEPTS

• **Software and its engineering** → **Frameworks**; • **Computing methodologies** → *Search methodologies*;

KEYWORDS

Software-Tools, Evolutionary Optimization

ACM Reference format:
Jakob Bossek. 2017. ecr 2.0: A Modular Framework for Evolutionary Computation in R. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 7 pages.
https://doi.org/10.1145/3067695.3082470

1 INTRODUCTION

Evolutionary Computation (EC) in all its facets, e. g., evolutionary/genetic algorithms or genetic programming, is a sophisticated field in both research and application. Since basically all EC algorithms stick to the basic evolutionary loop of parent selection, variation and survival selection, it is straightforward to design modular frameworks to facilitate the design and application of evolutionary algorithms. A plethora of such frameworks have been developed in the last years for all major programming languages. Prominent examples are - among many others - *EO* [14], *jMetal* [7] for Java and *DEAP* [5, 11] for the Python programming language. Another prominent example is the HeuristicLab optimization environment [26, 27]. Most EC frameworks follow the black-box framework model [20]: provide a large collection of common evolutionary compoments and aim to hide the majority of complex internal implementation details. Though undeniably being useful, the black-box approach is restricted due to its nature. Certain specific customizations may not be possible with the black-box. Thus the user needs to concern oneself with undocumented low level implementations which is both cumbersome and time-consuming. An exception is *DEAP*. The authors of the latter implemented a lightwight white-box framework, where every step of the evolutionary algorithm is explicit, transparent and both easy to read and understand.

The R programming language [19] itself contains some build-in methods to tackle continuous optimization problems. Besides, the R community has implemented a vast set of optimizers (the comprehensive CRAN task view on optimization and mathematical programming [25] is a good starting point to explore the field of solvers) including some R packages which (partly) focus on evolutionary algorithms. The `genalg` [29] offers an R based genetic algorithm for binary and real-valued representations, but it is very restricted. Optimization of real-valued and permutation-based search spaces only is possible with the `gaoptim` package by Tenorio [24]. The packages `rCMA` [15] and `cmaesr` [2] contain implementations of the popular Covariance-Matrix-Adaptation Evolutionary Strategy [12, 13]. `rgenoud` [29] integrates a quasi-Newton method into an evolutionary algorithm. Several implementations of the Differential Evolution algorithm introduced by Storn and Price [23] for continuous global optimization problems exist: `DEoptim` [18] and `DEoptimR` [4] provide pure R implementations whereas the implementation in `RcppDE` [9] is written in C++. The field of Genetic Programming is covered well with the `rgp` package [10]. The `GA` package [21] successfully attempts to supply a black-box framework for genetic and evolutionary algorithms. It provides a large collection of evolutionary operators and supports binary, real-valued and permutation representations. All aforementioned R frameworks are just capable of building EAs for single-objective optimization problems. A collection of building blocks to aid the implementation of multi-objective evolutionary algorithms is provided by the packages `mco` and `emoa` [16, 17]. However, the latter packages in turn are not capable of handling single-objective problems. Moreover, the development unfortunately has stagnated. Finally, all above mentioned packages implement the black-box model and are limited to standard representations, i. e., it is not possible to define custom genotypes. Recently, our R package `ecr` (version 1) [3], short for *Evolutionary Computation in R*, was released. This was our first attempt to implement a flexible general-purpose EC framework to tackle arbitrary single- or multi-objecitve continuous or combinatorial optimization functions in R and overcome all

GECCO '17 Companion, July 15-19, 2017, Berlin, Germany

ACM ISBN 978-1-4503-4939-0/17/07…$15.00
https://doi.org/10.1145/3067695.3082470

Table 1: Features of some EC frameworks in R.

	GA	mco	emoa	ecr	ecr2
black-box approach	✓	✓	✓	✓	✓
white-box approach	✗	✓	✓	✗	✓
custom genotypes	✗	✗	✗	✓	✓
custom operators	✓	✓	✓	✓	✓
parallelization	✓	✗	✗	✓	✓
single-objective	✓	✗	✗	✓	✓
multi-objective	✗	✓	✓	✓	✓

aforementioned limitations. The experience in using the black-box `ecr` to implement evolutionary algorithms as well as some major drawbacks due to bad design choices and associated performance issues, motivated a reimplementation as a white-box.

The main contribution of this paper is the introduction of the official successor `ecr` (version 2) denoted as `ecr2` in the following. It is designed as a toolbox for rapid prototyping of custom evolutionary algorithms for single- and multi-objective optimization problems via a flexible white-box approach. Table 1 gives an overview of important features for the major EC frameworks in R.

The remainder of the paper is organized as follows: Section2 sketches the core ideas of `ecr2`. A detailed introduction by code examples is given in Section 3. Next, in Section 4 the flexibility of parallel programming is illustrated. A brief performance study is performed to compare `ecr2` with its predecessor `ecr` and the `GA` package in Section 5. Finally, Section 6 concludes the work and gives an outlook on future development.

2 DESIGN PRINCIPLES AND CONVENTIONS

Despite all EAs following the common evolutionary loop, using them in practice and in particular developing/adapting custom EAs requires fine-grained customization. In the black-box framework model an increase in flexibility and an addition of customization capabilities is mostly correlated with code bloating and a general increase in code complexity. This makes the code more prone to bugs and requires lengthy documentation. However, even the most flexible black-box framework is limited: allowing for every possible extension is an impossible task for black-box framework developers [5]. Consequently, in the worst case even small adaptations to EAs or EA components may require the user to dive into the undocumentated internals of high-level components. We experienced the above drawbacks in the first version of `ecr`. Thus, the reimplementation is designed as a white-box framework. The core design principles are inspired by the sophisticated Python framework DEAP:

(1) Make everything explicit; do not hide the internals. Instead, let the user write the evolutionary loop by hand.
(2) Provide few simple building blocks / operators which are sufficient to implement the majority of EA variants.
(3) Stick to few reasonable conventions: a set of individuals is always a `list` and the fitness values are always stored in a $(m \times n)$-matrix where m is the number of objectives and n the number of elements in the set. Even in the single-objective case the fitness is stored in a $(1 \times n)$-matrix even though generally a simple numeric vector is more appropriate.
(4) Rapid realization of EA prototypes is more valuable than rapid execution. The latter can be alleviated by two means: 1) recoding critical components in C/C++ or FORTRAN[1] or 2) parallelization.

Following these design principles the core of `ecr` is limited to just three major building blocks: the *control object*, *operators* and *utility functions*. Basically, the control object is a wrapper for the fitness/objective function, some meta information on the fitness function, e. g., the number of objectives, and the set of operators which form the EA toolbox. These operators are registered to the control object via a single function which also allows to initialize the operators with parameters, e. g., the tournament size for a k-tournament selection operator. Once the control object has been set up, the evolutionary loop can basically be written with base R only accessing the tools registered to the control object. However, a collection of utility functions implement certain frequently used EA submodules and thus can be used to facilitate the implementation process. The best way to illustrate the interaction of the components is by example as done in the next section.

3 EXAMPLES

In this section we apply `ecr` in order to demonstrate its functionality by application-oriented examples[2]. We start with a simple Genetic Algorithm to solve the ONE-MAX problem to introduce the basic workflow. Next, a space-filling design is generated by means of an evolutionary algorithm. In this example custom representations and operators are introduced. The last example deals with multi-objective scheduling.

In order to execute the code snippets below one needs to install `ecr` first. The current official release is available at the Comprehensive R Archive Network (CRAN). The installation is straight forward:

```
install.packages("ecr")
library(ecr)
```

The current development version is publicly available at the official GitHub repository https://github.com/jakobbossek/ecr2. Use the `devtools` package to install the developement version.

```
install.packages("devtools")
devtools::install_github("jakobbossek/ecr2")
library(ecr)
```

3.1 Example 1: Solving the ONE-MAX problem

This examples is inspired by the introductory example in [5]. The ONE-MAX problem is the problem of *counting bits* and can be formulated as follows: find the bitstring $\mathbf{x} \in \{0, 1\}^n$ that maximizes $f(\mathbf{x}) = \sum \mathbf{x}_i$. A straight forward representation of a bitstring in R is via a simple vector of zeros and ones. Hence, the objective function expects a single R vector and sums up its components.

```
fitness.fun = function(x) sum(x)
```

[1] It is possible in base R to interface C or Fortran code out of the box. Furthermore, the R package Rcpp [8] makes the connection to C++ easy.
[2] The example codes can be downloaded – beside many others – from the official GitHub repository https://github.com/jakobbossek/ecr2/tree/master/inst/examples.

Furthermore, we define some variables of the Genetic Algorithm. We want to use a $(\mu + \lambda)$-strategy with $\mu = 25$ and $\lambda = 30$.

```
MU = 25; LAMBDA = 30; N.BITS = 50; MAX.ITER = 100
```

Thereafter, we initialize the *control* object. This object is of utmost importance. It stores the evolutionary toolbox and information on the fitness function at hand.

```
ctrl = initECRControl(fitness.fun,
  n.objectives = 1L, minimize = FALSE)
```

Here we state the number of objectives via `n.objectives` and the direction of the optimization via `minimize` (the default is to minimize all objectives).

In the next step the toolbox is filled with evolutionary operators. We use the classical bitflip mutation with independent probability of mutation initialized as $p = 1/N.BITS = 0.02$ and standard crossover recombination. Moreover, the mating pool is filled via binary tournament selection and the survival strategy selects the fittest individuals.

```
ctrl = registerECROperator(ctrl,
  "mutate", mutBitflip, p = 1 / N.BITS)
ctrl = registerECROperator(ctrl,
  "recombine", recCrossover)
ctrl = registerECROperator(ctrl,
  "selectForMating", selTournament, k = 2L)
ctrl = registerECROperator(ctrl,
  "selectForSurvival", selGreedy)
```

We use the `registerECROperator` function to store each operator in the control object. The function expects the control object as the first argument. The second argument is the *slot* to store the operator in, e. g., the bitflip mutation operator can be accessed via `ctrl$mutate` or `ctrl[["mutate"]]`, respectively. Thhe third parameter is the operator itself. Additionally, parameters for the corresponding operator may be passed optionally as further arguments.

Next, an initial population of μ individuals is generated with the `genBin` generator (generates a list of binary strings) and each individual's fitness is computed by means of the `evaluateFitness` helper.

```
population = genBin(MU, N.BITS)
fitness = evaluateFitness(ctrl, population)
```

Finally, the evolutionary loop is implemented explicitly. It runs for `MAX.ITER` generations and finally returns the best individual and its fitness value.

```
for (i in seq_len(MAX.ITER)) {
  offspring = generateOffspring(ctrl,
    population, fitness,
    lambda = LAMBDA, p.recomb = 0.7, p.mut = 0.3)

  fitness.o = evaluateFitness(ctrl, offspring)

  sel = replaceMuPlusLambda(ctrl, population,
    offspring, fitness, fitness.o)
  population = sel$population
  fitness = sel$fitness
}

print(population[[which.max(fitness)]])
print(max(fitness))
```

Here, the call to `generateOffspring` in line 16 could be written more explicit via

```
offspring = recombinate(ctrl, population, fitness,
  lambda = LAMBDA, p.recomb = 0.7)
offspring = mutate(ctrl, offspring, p.mut = 0.3)
```

We can go even one level deeper and explode the `mutate` function in the third line of the previous listing: Sample λ random numbers from a $\mathcal{U}(0, 1)$ distribution and check for each whether it is lower than the mutation probability. This results in a logical vector of length λ. Next, the mutation operator is applied in a functional manner to all offspring individuals that should be mutated.

```
idx.mut = runif(LAMBDA) < 0.3
if (any(idx.mut) > 0)
  offspring[idx.mut] = lapply(offspring[idx.mut],
    ctrl$mutate)
```

`ecr` also offers a black-box interface for standard tasks, e. g., for a simple Genetic Algorithm like the one introduced in this Section. The black-box is a single R function whose function signature resembles the signature of the **`optim`** function in base R. This was done to provide a familiar entry point for R users who aim to solve some optimization problem with standard representations. However, this is just an additional feature which may be useful for some R users. The ONE-MAX function can be solved via the black-box with the following code.

```
res = ecr(fitness.fun = fitness.fun,
  n.objectives = 1L, minimize = FALSE,
  representation = "binary", n.bits = N.BITS,
  mu = MU, lambda = LAMBDA, survival.strategy = "plus",
  mutator = setup(mutBitflip, p = 1 / N.BITS),
  p.mut = 0.3, p.recomb = 0.7,
  terminators = list(stopOnIters(MAX.ITER)))

print(res$best.y)
print(res$best.x)
```

3.2 Example 2: Space-filling sampling plan

Space-filling designs play a crucial role, e. g., as initial designs in surrogate-assisted optimization or the wide field of statistics. Imagine the 2D continuous case: here we are interested in placing N points in the unit square. The minimal distance between each two points should be maximized (maximin criterion) and the points should be as uniformly distributed as possible. This is widely known as a Latin-Hypercube-Design (LHS) [22]. Here we use a $(50, 50)$-strategy with 10-elitism, i. e., the 10 fittest individuals of the i-th generation are guaranteed to survive, to find a maximin-LHS design. The $(N \times 2)$-matrix of point coordinates is a good representation in this case. Standard mutation is not suitable. Thus, we come up with a custom mutator: replace each point of a given design with probability 0.1 with a random point sampled uniformly at random in $[0, 1]^2$.

```
pointReplace = makeMutator(
  mutator = function(ind) {
    idx = which(runif(nrow(ind)) < 0.1)
    ind[idx,] = matrix(runif(2*length(idx)), ncol = 2)
    return(ind)
  },
  supported = "custom")
```

Each mutator expects an individual `ind` and an additional parameter list `par.list`, which may be used to override potential default parameter values.

Next, we define some variables and the fitness function, initialize the control object and generate an initial population of μ individuals by placing all N points uniformly at random in the unit square for each individual.

```
MU = 50; LAMBDA = 50; MAX.ITER=2000L
fitness.fun = function(x) min(dist(x))

ctrl = initECRControl(fitness.fun,
  n.objectives = 1L, minimize = FALSE)
ctrl = registerECROperator(ctrl, "mutate",pointReplace)
ctrl = registerECROperator(ctrl,
  "selectForSurvival", selGreedy)

population = gen(matrix(runif(N * 2), ncol = 2), MU)
fitness = evaluateFitness(ctrl, population)
```

To keep track of the optimization process we initialize a logger, which in the default settings logs the minimal, mean and maximal fitness values of each generation.

```
logger = initLogger(ctrl, init.size = MAX.ITER + 1)
updateLogger(logger, population, fitness,
  n.evals = LAMBDA)
```

The `init.size` argument defaults to 1000. We recommend to adjust this argument if the number of generations is known in advance and no other stopping condition is applied. If at some point in time the logger experiences an overflow, the size is doubled internally.

Finally the evolutionary loop is implemented. Here the logic simply copies the current population and mutates each individual with probability 0.8.

```
for (i in seq_len(MAX.ITER)) {
  offspring = mutate(ctrl, population, p.mut = 0.8)
  fitness.o = evaluateFitness(ctrl, offspring)

  sel = replaceMuCommaLambda(ctrl, population,
    offspring, fitness, fitness.o, n.elite = N.ELITE)
  population = sel$population
  fitness = sel$fitness
  updateLogger(logger, population, fitness, n.evals =
     MU)
}
```

Figure 1 illustrates a uniform sampling plan (right) and the maximin-LHS design (left) returned by the above EA. We observe, that the distances between pairs of points are indeed bigger. Figure 2 shows a line plot of the logged statistics as returned by `plotStatistics(logger)`. We observe a typical behaviour of EAs: a rapid increase in the best individuals fitness and a long period of just some minor improvements. Every `ecr` plot function makes use of the `ggplot2` R package [28] in order to produce sophisticated high quality graphics. An object of class `ggplot` is returned which may be decorated and modified with additional graphics layers, e. g., placing the legend above the plot in Figure 2 was realized by `plotStatistics(logger) + theme(legend.position = "top")`.

3.3 Example 3: Multi-objective scheduling

In the last example we tackle the bi-objective scheduling problem $1|d_i|\sum C_i, L_{\max}$, i. e., the task of finding a schedule/permutation

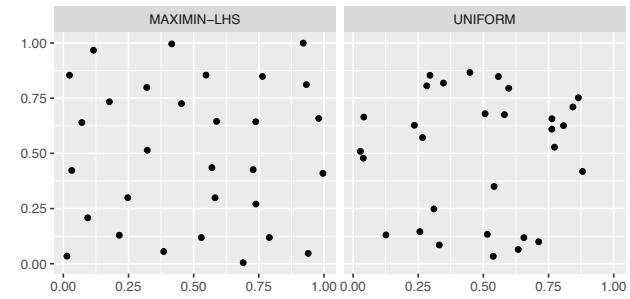

Figure 1: Maximin-LHS (left) and a uniform design generated by placing points uniformly at random in the unit square (right).

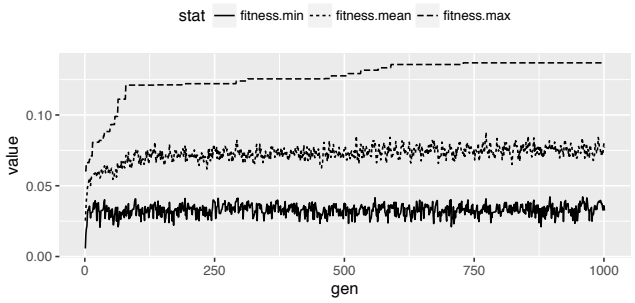

Figure 2: Line plot of minimal, mean and maximal fitness values for each generation.

of n jobs for a single machine minimizing the sum of completion times $C_i, i = 1, \ldots, n$ and simultaneously minimizing the maximum lateness $L_{\max} = \max_i(C_i - d_i)$, where d_i are the due dates of the jobs. The goal is to approximate the Pareto-optimal set/front of non-dominated solutions. For illustration we first simulate some jobs by a simple heuristic: each job has a processing p_i time sampled uniformly at random from a $\mathcal{U}(1, 20)$ distribution. The due dates are sampled in a subsequent step from a $\mathcal{N}(\mu = 10p_i, \sigma = 3p_i)$ distribution.

```
proc.times = runif(n.jobs, 1, 20)
due.dates = proc.times + sapply(proc.times,
  function(i) {
    rnorm(1L, mean = 10 * i, sd = 3 * i)
  })
jobs = data.frame(j = 1:n.jobs,
  p = proc.times, d = due.dates)
```

All jobs are stored in a `data.frame` with three columns: j for job number, p for processing time and d for the corresponding due date. Next we define the bi-objective fitness function. The jobs are ordered according to the passed permutation `job.order`. The return value is a simple vector with two components, namely $\sum C_i$ and $L_{\max}$.

```
fitness.fun = function(job.order) {
  sorted.jobs = jobs[job.order, ]
  c(sum(cumsum(sorted.jobs$p)),
    max(cumsum(sorted.jobs$p) - sorted.jobs$d))
```

```
}
```

The evolutionary setup is very similar to the preceeding examples and thus not discussed in detail. The only thing that needs attention here is that we register the NSGA-II [6] survival selection mechanism via `selNondom`.

```
MU = 20;LAMBDA = 10;MAX.ITER = 2000

ctrl = initECRctrl(fitness.fun,
  n.objectives = 2L, minimize = c(TRUE, TRUE))
ctrl = registerECROperator(ctrl,
  "mutate", mutScramble)
ctrl = registerECROperator(ctrl,
  "recombine", recOX)
ctrl = registerECROperator(ctrl,
  "selectForMating", selSimple)
ctrl = registerECROperator(ctrl,
  "selectForSurvival", selNondom)

population = genPerm(MU, n.jobs)
fitness = evaluateFitness(population, ctrl)
```

We are interested in the dominated hypervolume (HV) trajectory and thus provide a reference point and tell the logger to compute the HV for the fitness of each generation.

```
ref.point = c(15000, 500)

logger = initLogger(ctrl,
  log.stats = list(fitness = list("HV" = list(
    fun = computeHV,
    pars = list(ref.point = ref.point)))),
  init.size = MAX.ITER + 1L)
updateLogger(logger, population,
  fitness = fitness, n.evals = MU)
```

Finally, the evolution is launched for the specified number of generations. Figure 4 shows the plot generated by `plotFront` passing the final matrix of fitness values. The hypervolume developement is visualized in Figure 3.

```
for (i in seq_len(MAX.ITER)) {
  offspring = recombinate(ctrl,
    population, fitness = fitness,
    lambda = LAMBDA, p.recomb = 0.8)
  offspring = mutate(ctrl, offspring, p.mut = 0.3)

  fitness.o = evaluateFitness(offspring, ctrl)

  sel = replaceMuPlusLambda(ctrl, population,
    offspring, fitness, fitness.o)
  population = sel$population
  fitness = sel$fitness

  updateLogger(logger, population,
    fitness = fitness, n.evals = LAMBDA)
}

stats = getStatistics(logger)
pl.stats = plotStatistics(stats)
pl.front = plotFront(fitness,
  obj.names = c("SumCi", "Lmax"))
```

4 PARALLELIZATION

In a run of an EA the three main steps, i. e., fitness evaluation, parent/survival selection and application of evolutionary operators to generate offspring are iteratively performed until some termination condition is satisfied. Typically, in real-life applications, the fitness evaluation is the dominating part regarding the computational time needed. However, depending on the complexity of the underlying representation, evolutionary operators may be costly as well. Since EAs perform these steps over and over again it is straightforward to parallelize these tasks.

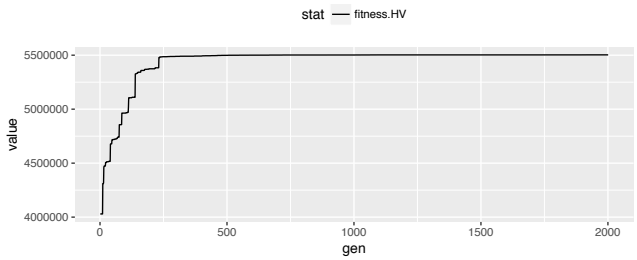

Figure 3: Line plot of the dominated hypervolume trajectory of example 3.

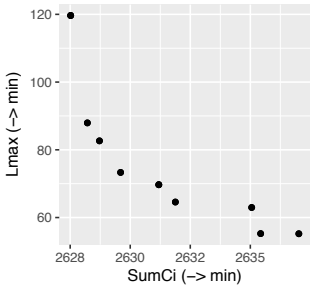

Figure 4: Approximation of the non-dominated front found by the EMOA in example 3.

In `ecr2` it is possible to parallelize different *levels* of computation to utilize parallel computational power, e. g., multiple CPU cores or a set of compute nodes of a HPC (High Performance Cluster). For maximum flexibility in `ecr2` parallelizable steps are realized with the R package `parallelMap` [1], which offers a unified interface to different parallelization back-ends in R and falls back to the non-parallel `lapply` function if no back-end is activated. The review of all back-ends and additional options of `parallelMap` is beyond the scope of this article. We thus refer the interested reader to the official tutorial at the public GitHub repository[3]. At the moment of writing, `ecr2` allows parallel fitness function evaluation (level `"ecr.evaluateFitness"`) and parallel offspring generation (level `"ecr.generateOffspring"`.

Setting up parallelization is realized by a simple function call from `parallelMap` prior to the code fragment to parallelize. E. g., given a multicore CPU with four cores `parallelStartMultiCore(cpus = 3,level = "ecr.evaluateFitness")` activates parallelized fitness function evaluation on three cores[4].

[3] https://github.com/berndbischl/parallelMap#parallelmap
[4] It is a good idea to keep one core for other computations in this case.

```
library(parallelMap)
library(ecr)

 ...
parallelStartMultiCore(cpus = 3,
  level = "ecr2.evaluateFitness")
...
fitness = evaluateFitness(control, population)
...
parallelStop()
```

The violin plots in Figure 5 visualize the distribution of the absolute running times in seconds for both the parallel and the sequential version of a (10 + 10)-EA executed 100 times on an artificially delayed fitness function.

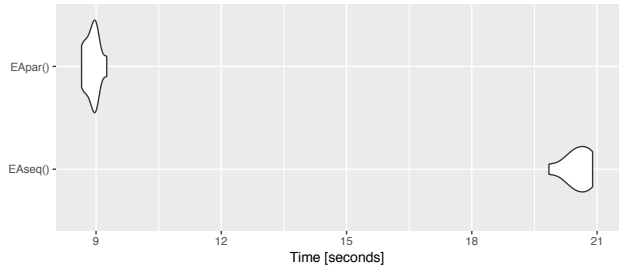

Figure 5: Violin plot showing the distribution of absolute running times of a sequential and parallel version of a (10 + 10)-EA to optimize a fitness function with delayed function evaluation. The parallelized version clearly outperforms the sequential version.

To get a list of all currently supported parallelization levels call the `parallelGetRegisteredLevels()` function.

```
parallelGetRegisteredLevels()
> ecr: ecr.evaluateFitness, ecr.generateOffspring
```

5 PERFORMANCE COMPARISSON

The predecessor `ecr` had some serious performance issues due to some unfortunate decisions during the design process. As a consequence one of the top points on the agenda for `ecr2` was to put those performance drawbacks behind. This goal was successfully accomplished in the course of reimplementation. By means of example Figure 6 shows violin plots of the running times of a (100 + 100)-EA runs each 100 times on a test function for each 1000 generations. The `ecr2` implementation was tested against the predecessor and the `GA` package [21]. We observe quite similar distributions of the `GA` and `ecr2` implementations with `ecr` being lagged far behind by a factor of about 10.

6 CONCLUSION

The majority of EC frameworks – and all EC frameworks for the R language in particular – implement general purpose evolutionary components in a black-box manner hiding complex internals. Though this black-box approach indeniably has its charm and benefits, it is nevertheless restricted in its customization options. Hence,

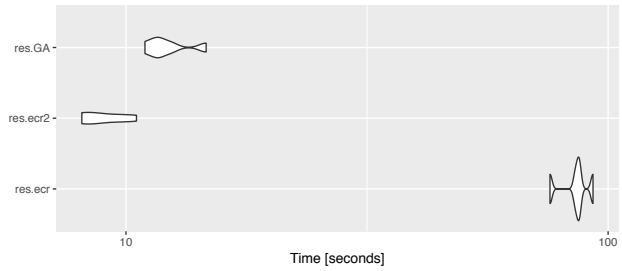

Figure 6: Violin plot showing the distribution of absolute running times of three implementations of a (100 + 100)-EA based on different frameworks in R.

the implementation of new ideas and specific features may leave the user overstrained. We introduced the reimplementation of the `ecr` package for evolutionary computation in R. It offers a manageable set of evolutionary components and helper functions for building evolutionary loops in an explicit fashion (white-box approach). Sticking to just a few conventions the package offers a flexible way for rapid prototyping and implementation of new EA ideas. We showed the application on a collection of optimization scenarios pointing out its usability in different contexts and the flexibility of the toolbox.

Future development is manifold: implementation of additional evolutionary operators, introduction of subpopulation models and migration schemata are just a few directions.

REFERENCES

[1] Bernd Bischl and Michel Lang. *parallelMap: Unified Interface to Parallelization Back-Ends.* https://github.com/berndbischl/parallelMap R package version 1.4.
[2] Jakob Bossek. 2016. *cmaesr: Covariance Matrix Adaptation Evolution Strategy.* https://github.com/jakobbossek/cmaesr R package version 1.0.1.
[3] Jakob Bossek. 2016. *ecr: Evolutionary Computing in R.* https://github.com/jakobbossek/ecr R package version 1.0.1.
[4] Eduardo L. T. Conceicao and Martin Maechler. 2016. *DEoptimR: Differential Evolution Optimization in Pure R.* http://CRAN.R-project.org/package=DEoptimR R package version 1.0-6.
[5] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python Framework for Evolutionary Algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12).* ACM, New York, NY, USA, 85–92. https://doi.org/10.1145/2330784.2330799
[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017
[7] Juan J. Durillo and Antonio J. Nebro. 2011. jMetal: A Java Framework for Multi-objective Optimization. *Adv. Eng. Softw.* 42, 10 (Oct. 2011), 760–771. https://doi.org/10.1016/j.advengsoft.2011.05.014
[8] Dirk Eddelbuettel and Romain Francois. 2011. Rcpp: Seamless R and C++ Integration. *Journal of Statistical Software* 40, 1 (2011), 1–18. https://doi.org/10.18637/jss.v040.i08
[9] Dirk Eddelbuettel extending DEoptim which itself is based on DE-Engine (by Rainer Storn). 2016. *RcppDE: Global Optimization by Differential Evolution in C++.* http://CRAN.R-project.org/package=RcppDE R package version 0.1.5.
[10] Oliver Flasch, Olaf Mersmann, Thomas Bartz-Beielstein, Joerg Stork, and Martin Zaefferer. 2014. *rgp: R genetic programming framework.* http://CRAN.R-project.org/package=rgp R package version 0.4-1.
[11] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *J. Mach. Learn. Res.* 13, 1 (July 2012), 2171–2175. http://dl.acm.org/citation.cfm?id=2503308.2503311
[12] N. Hansen. 2006. The CMA evolution strategy: a comparing review. In *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, J. A. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea (Eds.). Springer, Berlin, Heidelberg, 75–102.

[13] N. Hansen. 2009. *The CMA Evolution Strategy: A Tutorial.* http://www.lri.fr/~hansen/cmatutorial.pdf
[14] M. Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. 2002. *Evolving Objects: A General Purpose Evolutionary Computation Library.* Springer Berlin Heidelberg, Berlin, Heidelberg, 231–242. https://doi.org/10.1007/3-540-46033-0_19
[15] Wolfgang Konen and Nikolaus Hansen. 2015. *rCMA: R-to-Java Interface for 'CMA-ES'.* http://CRAN.R-project.org/package=rCMA R package version 1.1.
[16] Olaf Mersmann. 2012. *emoa: Evolutionary Multiobjective Optimization Algorithms.* http://CRAN.R-project.org/package=emoa R package version 0.5-0.
[17] Olaf Mersmann. 2014. *mco: Multiple Criteria Optimization Algorithms and Related Functions.* http://CRAN.R-project.org/package=mco R package version 1.0-15.1.
[18] Katharine Mullen, David Ardia, David Gil, Donald Windover, and James Cline. 2011. DEoptim: An R Package for Global Optimization by Differential Evolution. *Journal of Statistical Software* 40, 6 (2011), 1–26. http://www.jstatsoft.org/v40/i06/
[19] R Core Team. 2017. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/
[20] Don Roberts and Ralph Johnson. 1996. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Proceedings of the Third Conference on Pattern Languages and Programming.* Addison-Wesley.
[21] Luca Scrucca. 2013. GA: A Package for Genetic Algorithms in R. *Journal of Statistical Software* 53, 4 (2013), 1–37. http://www.jstatsoft.org/v53/i04/
[22] Michael Stein. 1987. Large Sample Properties of Simulations Using Latin Hypercube Sampling. *Technometrics* 29, 2 (1987), 143–151.
[23] Rainer Storn and Kenneth Price. 1995. Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces. (1995).
[24] Fernando Tenorio. 2013. *gaoptim: Genetic Algorithm optimization for real-based and permutation-based problems.* http://CRAN.R-project.org/package=gaoptim R package version 1.1.
[25] S Theussl. 2013. CRAN Task View: Optimization and Mathematical Programming. (2013). https://cran.r-project.org/web/views/Optimization.html Version: 2015-08-19.
[26] S. Wagner and M. Affenzeller. 2004. *The HeuristicLab optimization environment.* Technical Report.
[27] S. Wagner and M. Affenzeller. 2005. *HeuristicLab: A Generic and Extensible Optimization Environment.* Springer Vienna, Vienna, 538–541. https://doi.org/10.1007/3-211-27389-1_130
[28] Hadley Wickham. 2009. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York. http://ggplot2.org
[29] Egon Willighagen and Michel Ballings. 2015. *genalg: R Based Genetic Algorithm.* http://CRAN.R-project.org/package=genalg R package version 0.2.0.