EvoLP.jl: A playground for Evolutionary Computation in Julia

Xavier F. C. Sánchez-Díaz[1,*], Ole Jakob Mengshoel[1]

[1]*Norwegian University of Science and Technology, Trondheim, Norway*

Abstract

Optimisation is highly relevant in many problems in artificial intelligence, machine learning, engineering and statistics. In these situations, optimisation by means of evolutionary computation becomes especially relevant as it makes few assumptions (such as differentiability) about the objective function. Problems such as these represent various research opportunities, both in the Norwegian and European contexts. In this work we present an open-source software framework, EvoLP.jl, as an effort to support the research in this niche. EvoLP.jl is a Julia package that implements reusable pieces of code for experimenting with single-objective evolutionary computation algorithms and its components. The framework is composed of blocks that span the separate phases of the evolutionary process: population initialisation, selection, crossover, and mutation. These blocks can be put together to create a modular solver, where each of the components can easily be swapped for testing. In addition, we provide some built-in algorithms and a few optional utilities for analysis (like benchmark test functions, result reporting and statistics logging). EvoLP.jl is an effort of the Norwegian Open Artificial Intelligence Lab and strives to comply with the guidelines of the Julia scientific community. It is well-tested, provides extensive documentation and is free—available for everyone to use under an open-source license. It is our intention that EvoLP.jl becomes a useful tool not only for research in evolutionary computation but also in the education and innovation scenarios.

Keywords

Evolutionary Algorithms, Genetic Algorithms, Particle Swarm Optimisation, Evolutionary Computation Software Tools

1. Introduction

The concept of Artificial Intelligence (AI) was originally used to refer to the science and engineering of intelligent agents [1]. In the last decade, however, the dominant research topic—Machine Learning (ML) and its applications—has overshadowed other areas of AI research. In Norway, for example, the trend has been similar. In the last three annual reports from the Norwegian Research Center for AI Innovation (NorwAI), several publications and projects in AI are described but none of them mention any optimisation techniques, let alone any of those belonging to the Evolutionary Computation (EC) family [2, 3, 4].

NAIS 2023: The 2023 symposium of the Norwegian AI Society, June 14-15, 2023, Bergen, Norway

*Corresponding author.

xavier.sanchezdz@ntnu.no (X. F. C. Sánchez-Díaz); ole.j.mengshoel@ntnu.no (O. J. Mengshoel)

https://saxarona.github.io/ (X. F. C. Sánchez-Díaz); https://www.ntnu.no/ansatte/ole.j.mengshoel (O. J. Mengshoel)

0000-0003-2271-439X (X. F. C. Sánchez-Díaz); 0000-0003-2666-5310 (O. J. Mengshoel)

A quick search in the project bank of The Research Council of Norway shows 375 projects when searching for "artificial intelligence", "AI" or their Norwegian translation, between 2018 and 2023 [5]. In contrast, performing a similar search for the same period using the keyword "evolutionary" (or its Norwegian translation) yields only 14 projects [6]. We believe there is potential in delving deeper in this line of research and its methods, as they can also be used for (and in conjunction with) AI algorithms [7].

As an effort to increase the interest in optimisation by means of EC in the Norwegian scientific community, we propose a software package in Julia that provides reusable computing patterns for experimenting and analysing several components for single-objective EC algorithms. The framework focuses on the idea of using small building blocks that can be put together, and encompass the different stages of an evolutionary process: initialisation of the population, parent selection, recombination, mutation and survival. In addition, it provides a few built-in algorithms—namely the 1+1 Evolutionary Algorithm (EA), a generational Genetic Algorithm (GA) and a Particle Swarm Optimisation (PSO) solver—and test functions to try them out. We aimed our attention to the components and their connections: how to put together the different tools available to *play around* (as in a playground) with a new solver.

Our framework, EvoLP.jl—which is a short form of ***Evo****lusjonær* ***Leke****plass*[1]—is free and open-source, and is also accessible via the General Julia Registry: it can be installed from the package manager in a couple of instructions. Fully guided tutorials and short examples are available in the documentation, which can be browsed through the Julia REPL as well, and are detected by most linting services in IDEs.

2. Related Work

Research on EC can take on many different forms and paths, and the differences between solvers can become very specific. Nevertheless, many modular software tools have succeeded in creating abstractions that are general enough to apply to several of the algorithms they include. For Java, JCLEC [8] and its multi-objective variation JCLEC-MO [9] are two examples. More recently, JGEA [10] continued with this similar modular design. HeuristicLab [11] is another framework developed entirely in C# and employing a graphical user interface. DEAP [12], developed in Python 2.0, posed many interesting components that we wanted to include in our design. However, the framework has not been entirely rewritten to Python 3, so it suffers from many compatibility issues.

Python running times are not competitive against compiled languages, but several frameworks have found solutions around that fact. For example, EvoJax [13] uses hardware acceleration for neuroevolution, or LEAP [14] which uses Dask for building a computational graph that is easy to parallelise. Another example in this category is pygmo which is a Python wrapper for the pagmo C++ optimisation library [15].

Julia has been a rapidly growing programming language in the scientific community due to its performance and simple syntax, and offers similar software tools. The optimisation landscape in Julia consists of many different packages, for example SciML [16] which encompasses a

[1] Although EVOLP (Evolving Logic Programs) is an acronym already in use, package names in Julia are case-sensitive and must include `.jl` in the name, making EvoLP.jl unique during web search and indexing.

whole scientific environment for machine learning and optimisation, JuMP [17] for mathematical optimisation, and Metaheuristics [18] and BlackBoxOptim [19] for non-convex and non-differentiable problems. Specific solutions for EC also exist (like EBIC.jl [20], Cambrian [21] and Evolutionary [22]).

3. Design Principles

The decision of developing EvoLP.jl in Julia comes from different standpoints. First is the performance against Python (which is the go-to scripting language for AI). Although scientific libraries (like numpy or scipy in Python) have been compiled for faster performance, they require that the code is vectorisable. This is not an easy task for EC algorithms, where many stochastic decisions are taken in an iterative manner. Another consideration about the programming language was the paradigm they use. Although iterative in nature, each of the phases of the evolutionary process can be abstracted to a single function modifying an object at a time (as opposed to an object being modified by its own methods). The functional paradigm also fits better with the scientific notation used in this line of research. EvoLP.jl takes advantage of these features of the Julia programming language along with polymorphism by multiple dispatch. Unlike C++ or Java, Julia uses `structs` (also known as `types` in other functional programming languages) instead of objects. Therefore, all procedures and operations that manipulate such types need to be implemented as functions instead of being methods inside an object. In this way, abstraction is easier to achieve since a single function represents a single step in the evolutionary process.

The polymorphic nature of Julia allows functions to behave differently using multiple dispatch. This enables the programmer to create multiple 'versions' of a function, for example the `mutate` function, which will behave differently depending on the type of mutation in an algorithm and the arguments passed. By doing so, we encapsulate all the specifics of a given operator inside a single function. This is useful for maintenance as there are no complicated program flows in a big `mutate` function, but rather several small `mutate` functions. The other advantage is extensibility: to add a new *mutator* one needs only to add a new function.

The framework is built to capitalise on these features, and provides small, reusable pieces of code that we refer to as *blocks*. These blocks can be either one of two kinds: `type` or `function` blocks, and can be put together to generate evolutionary algorithms in a few lines of code, where swapping components is simple and easy. EvoLP.jl started as an extension to the GA and PSO implementations presented by Kochenderfer and Wheeler [23]. We later added important features that are common in an analysis work flow (e.g., result reporting or logging of statistics) and extended most of the functions to ensure reproducibility when dealing with stochastic components. In this way, experiments are easy to reproduce and share, which makes EvoLP.jl a good alternative when designing and analysing EC algorithms.

After creating a development cycle we were content with, the first version of EvoLP.jl was published and registered as an official Julia package. The code, along with documentation, examples and the development roadmap can be found in the GitHub repository.[2]

[2]See https://github.com/ntnu-ai-lab/EvoLP.jl/

4. Examples

This section includes complete, illustrative examples where blocks are coupled together in a single work flow for two well-known optimisation problems. A detailed description of the framework components is included in posterior sections.

4.1. The Rosenbrock function

In this example, we solve a minimisation benchmark for continuous optimisation, the Rosenbrock function [24]. For this problem, the following components are used:

- **Generator**: Continuous normal, i.e., with a population of 2D vectors $\mathbf{x} \in X$ such that $X \sim \mathcal{N}(\mu, \sigma^2)$ with $\mu = \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$ and $\sigma^2 = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$
- **Selector** (S): Rank-based, generational.
- **Recombinator** (C): Interpolation crossover with $\lambda = 0.5$
- **Mutator** (M): Gaussian mutation with $\sigma = 0.5$
- **Objective function**: 2-dimensional version of the Rosenbrock built-in test function, i.e. $f(\mathbf{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2$ with default values of $a = 1$ and $b = 5$. The optimum is $f([a, a^2]) = 0$.
- **Logbook**: Calculating maximum, minimum, mean and median fitness values of the population at every iteration.
- **Algorithm**: Built-in generational GA, with a population size $|X| = 500$ and termination after 100 generations. Both crossover and mutation probabilities are 100%.
- **Result**: The return value of the algorithm, available for further inspection.

The code can be implemented in a single file:

```
using EvoLP, OrderedCollections, Statistics

X_size = 500
k_max = 100
X = normal_rand_vector_pop(X_size, [0, 0], [1 0; 0 1])
S = RankBasedSelectionGenerational()
C = InterpolationCrossover(0.5)
M = GaussianMutation(0.5)
statnames = ["mean_f", "max_f", "min_f", "median_f"]
fns = [mean, maximum, minimum, median]
log_dict = LittleDict(statnames, fns)
statsbook = Logbook(log_dict)

result = GA(statsbook, rosenbrock, X, k_max, S, C, M)
@show optimum(result)
@show optimizer(result)
@show f_calls(result)
@show statsbook.records[end]
```

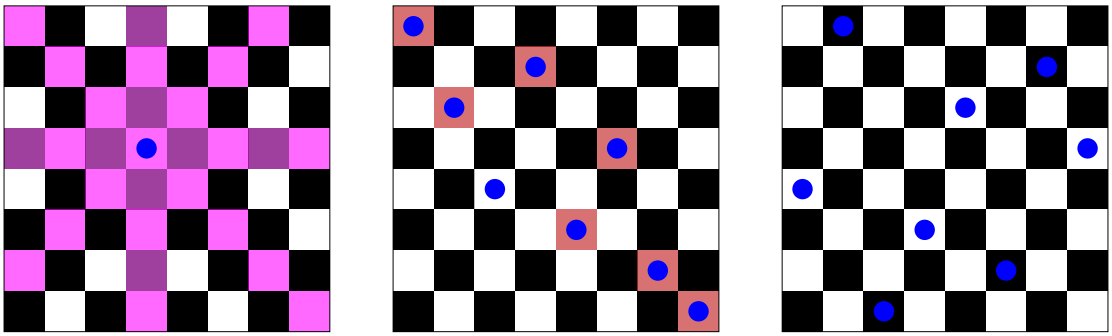
Here is one possible output of the example above:

```
optimum(result) = 0.0015029528354023858
optimizer(result) = [1.0367119356341026, 1.0803427525882299]
f_calls(result) = 50050
(mean_eval = 3.7839504926952294, max_f = 22.281919411164413, min_f =
↪  0.0015029528354023858, median_f = 2.429775485243721)
```

This full example is available as a guided tutorial in the documentation of EvoLP.jl.[3]

4.2. The 8-queen problem

This example deals with a classical combinatorial problem in AI where the goal is to place eight queens in a chess board such that no queen checks each other [1]. Figure 1 shows three configurations where the constraints and possible clashes are highlighted.



(a) Constraints of a queen (b) Conflicts between queens (c) A possible solution

Figure 1: The 8-queen problem. 1a shows the constraints (in pink) imposed by the placement of a single queen piece (in blue). 1b highlights the conflicts arising from a possible configuration of the board. 1c illustrates one possible solution with no conflicts.

For this example, we use the following components:

- **Generator**: random permutation, i.e., with a population of 8D vectors $\mathbf{x} \in X$ such that $X \sim \mathcal{U}(\mathcal{S}_8)$ where $\mathcal{S}_8$ is the set of permutations of $[8] = \{1, 2, \ldots, 8\}$
- **Selector** (S): Random tournament selection of size $T_s = 5$.
- **Recombinator** (C): Order one crossover for permutation vectors.
- **Mutator** (M): Swap mutation.
- **Objective function**: $f(\mathbf{x}) = \sum_{i=1}^{i=8} \textsc{DiagConstraints}(q_i)$ where DiagConstraints is a custom procedure (coded separately) which counts the number of conflicts of each queen q_i.
- **Logbook**: Calculating maximum, minimum, mean and median fitness values of the population at every iteration.

[3] See https://ntnu-ai-lab.github.io/EvoLP.jl/stable/tuto/ga_rosenbrock.html

- **Algorithm**: Custom steady-state GA that generates two offspring per generation, with a population size $|X| = 100$, and termination after 500 generations. Crossover probability is 100% while mutation probability is 80%.
- **Result**: The return value of the algorithm, available for further inspection.

The code can be implemented in a single file, although it needs three parts: an objective function DiagConstraints, an algorithm, and a wrapper script. For simplicity, we show the last two components below:

```
function 8QSteadyGA(statsbook, f, X, k_max, S, C, M, mrate)
  n = length(X)
  # Generation loop
  for _ in 1:k_max
    fitnesses = f.(X)
    parents = select(S, fitnesses)  # this will return 2 parents
    parents = vcat(parents, select(S, fitnesses))  # add 2 more
    offspring = [cross(C, X[parents[1]], X[parents[2]])] # get first
    offspring = vcat(offspring, [cross(C, X[parents[3]],
    ↪  X[parents[4]])])
    X = vcat(X, offspring) # add to population
    # Mutation loop
    for i in eachindex(X)
      if rand() <= mrate
          X[i] = mutate(M, X[i])
      end
    end
    fitnesses = f.(X)
    compute!(logbook, fitnesses)

    # Find worst and remove, twice
    for _ in 1:2
      worst = argmax(fitnesses)
      deleteat!(X, worst)
      deleteat!(fitnesses, worst)
    end
  end
  # Result reporting
  best, best_i = findmin(f, X)
  n_evals = 2 * k_max * n + n
  result = Result(best, X[best_i], X, k_max, n_evals)
  return result
end
```

Then, the wrapper script which uses the `8QSteadyGA` algorithm to solve the problem using

the remaining blocks:

```
using EvoLP, OrderedCollections, Statistics

X_s = 100
T_s = 5
X = permutation_vector_pop(X_s, 8, 1:8)
S = TournamentSelectionSteady(T_s)
C = OrderOneCrossover()
M = SwapMutation()
statnames = ["mean_f", "max_f", "min_f", "median_f"]
fns = [mean, maximum, minimum, median]
log_dict = LittleDict(statnames, fns)
statsbook = Logbook(log_dict)
f = diag_constraints(x)

result  = mySteadyGA(statsbook, diag_constraints, X, 500, S, C, M,
↪  0.8)
@show optimum(result)
@show optimizer(result)
@show f_calls(result)
@show statsbook.records[end]
```

Here is one possible output of the example above:

```
optimum(result) = 0
optimizer(result) = Any[5, 1, 8, 6, 3, 7, 2, 4]
f_calls(result) = 100100
(mean_eval = 9.392156862745098, max_f = 20, min_f = 0, median_f = 8.0)
```

A visual representation of this specific solution is shown in Figure 1c. The full example (described at a greater detail) is available in the documentation of EvoLP.jl.[4]

5. The Framework

In this section, we detail the *blocks* of EvoLP.jl, some of which were showcased in the examples section.

5.1. Block Taxonomy in EvoLP.jl

Every basic block in EvoLP.jl is categorised into one of the following essential steps of EC:

- **Generators**. Function blocks for randomly initialising the population.

[4]See https://ntnu-ai-lab.github.io/EvoLP.jl/stable/tuto/8_queen.html

- **Selectors**. Type blocks for selecting parents for recombination via the `select` function.
- **Recombinators**. Type blocks for performing crossover via the `cross` function.
- **Mutators**. Type blocks for performing mutation on a single individual via the `mutate` function.

Extra functionality is also provided in EvoLP.jl through a few optional blocks:

- **Result**. A type block for reporting the results of algorithms.
- **Logbook**. A type block for computing and storing statistics throughout a run.
- **Test functions**. A curated list of pseudoboolean and continuous benchmark functions from the literature [23, 25, 26] to test an algorithm.
- **Built-in algorithm**. A small selection of algorithms (1+1 EA [27], GA [28] and PSO [29]) that are ready to use, all in function block form.

5.2. Initialisation

To initialise the population in EvoLP.jl, we provide several population generators. These blocks generate random individuals and put them in a `vector` (known as the *population*). The population can be of different data types depending on the type of generator invoked:

- **Binary**. Generates a population of n random boolean vectors.
- **Continuous uniform**. Generates a population of n floating point vectors, from random values uniformly sampled between a lower and upper bound.
- **Continuous Gaussian**. Similar to the continuous uniform, this generator returns a population of floating point vectors but instead sampled from a normal distribution with known means and variances.
- **Discrete combination/permutation**. Returns a random sample from a discrete domain, set or container. Sampling can be performed with or without replacement.

Continuous generators (both uniform and normal variants) for `Particles` are provided as well, which generate a `Particle` population that can be used with the PSO built-in algorithm.

5.3. Parent Selection

In EvoLP.jl, the selection phase is performed in two steps:

1. Choose one of the available selection operators (or **selectors**) and instantiate it.
2. Then, use it to perform the selection. This is achieved using the `select` function.

The `select` function uses the chosen *selector* to obtain indices from a vector (which is usually the vector of fitness evaluations of all individuals in the population). EvoLP.jl implements these selectors as if solving a minimisation problem, so every selector is comparable to an $\arg\min$ function with some degree of stochasticity. The available selectors are:

- **Roulette wheel**. This selector uses a fitness proportionate probability of selection.

- **Rank-based**. The probability of selection considers the rank of the fitness instead of its values.
- **Random tournament**. A winner is obtained from a tournament among a random sample of size k. This process is repeated a second time to determine the second parent.
- **Truncation**. Two random indices are selected from the top k individuals.

The selection is commonly performed once (known in the literature as *steady-state*). However, sometimes it may be desirable to perform the selection n times (i.e., once per every individual in the population), especially when working with *generational* algorithms. EvoLP.jl provides both variants for each selector.

After the desired selector has been instantiated, the user can call the `select` function inside the main loop of an algorithm (as was showcased in the examples in Section 4). Here is an illustrative snippet featuring the rank based selector in its steady-state variant:

```
# Instantiate outside the algorithm
S = RankBasedSelectionSteady()
...
function exampleAlgorithm(S, ...)
  ...
  # Main loop
  for gen in 1:k_max
    ...  # Do something
    parents = select(S, fitnesses)
  end
  ...
end
```

The same `select` function can behave differently depending on the chosen selector (thanks to polymorphism). Continuing with the rank-based selector example, here is the implementation of its `select` function:

```
function select(::RankBasedSelectionSteady, y; rng=Random.GLOBAL_RNG)
  ranks = ordinalrank(y, rev = true)
  cat = Categorical(normalize(ranks, 1))
  return rand(rng, cat, 2)
end
```

Additional selectors can be added by creating a new selector type and implementing its corresponding `select` function (see Section 5.6).

5.4. Crossover

In a similar fashion to the selection process, the crossover step of an EA is performed in two parts in EvoLP.jl: first choose a crossover operator (or **recombinator**) and then invoke the

appropriate function to carry out the crossover. The function in this case is called `cross`, and generates one offspring only. The latest version of EvoLP.jl (v1.0 at the time of writing of this article) provides recombinators only for vector-based populations. The available recombinators are the following:

- **Single point crossover (1PX)**. Select a random index and combine the parents at that point. Works on numeric vectors (i.e. boolean, integer or continuous) and combination of discrete values.
- **Two-point crossover**. Similar to 1PX, but using two cutting points instead. Works on the same types of vectors as 1PX.
- **Uniform crossover**. For numeric vectors and combination of discrete values. Each value is randomly selected between the parents.
- **Interpolation crossover**. For continuous vectors. The result is a scaled addition of both parents.
- **Order One crossover**. For discrete permutation-based individuals. This operator ensures that values remain unique after the crossover.

After choosing one of the recombinators, the crossover takes place when the appropriate `cross` function is called inside the main loop of the algorithm.

Currently, the recombinators do not check against the type of the individuals they are working on. This is important to consider, since using a numeric-only recombinator on a permutation-based problem (where values of an individual need to be unique) could result in generating an unfeasible solution. In future versions of EvoLP.jl, we plan to introduce a type for solution encoding, to ensure that only compatible blocks can be connected.

It is also important to note that some recombinators (as well as some selectors and mutators) have parameters that modify their behaviour. When instantiating this kind of operators, the desired value is passed as an argument to the constructor method.

5.5. Mutation

As with selection and crossover, the mutation phase in an EA is performed in EvoLP.jl by using the same two-step approach: choosing a **mutator** first and then calling the `mutate` function in the algorithm. The available mutators for numeric vectors in EvoLP.jl are:

- **Bitwise mutator**. For boolean vectors only. It has a controlling parameter which modifies the probability λ of independently flipping each bit.
- **Gaussian mutator**. For continuous vectors only. Adds Gaussian noise with a standard deviation σ which is controlled via a parameter.

Mutators for permutation-based individuals are also provided:

- **Swap mutator**. Swaps the values of two randomly chosen positions in the chromosome.
- **Insert mutator**. Inserts a value in another position, shifting the rest of the chromosome.
- **Scramble mutator**. Randomly selects a sub-string and then shuffles it.

- **Inversion mutator**. Randomly selects a sub-string and reverses it.

When one of the mutation operators has been instantiated outside of the algorithm, the `mutate` function can be invoked in the main loop of a solver to perform the mutation. As with recombinators, some mutators work on numeric vectors only, and using them on permutation-based individuals could lead to generating an unfeasible solution.

5.6. Custom Operators

EvoLP.jl was designed to be extensible, with the idea that a user can add custom blocks if needed. This process is, again, a two-step task (in line with the operators described in previous sections). To create a new operator block, the user needs to provide:

1. A *subtype* for the desired block
2. An appropriate function to operate on such subtype

For example, let us consider that the user wants to implement a new mutation operator, `CrazyMutation`. All mutator blocks are subtypes of the abstract `MutationMethod` *supertype*. By creating a new type that is derived from `MutationMethod`, the user is creating a new type block (i.e. `CrazyMutation`) and essentially completing the first step. As with the other blocks, this new mutator would need a special case of the `mutate` function. This is the second step: create a new `mutate` function that can receive the `CrazyMutation` block as an argument and modifies the individual as desired. To implement new selectors or recombinators, a similar process is followed. The three abstract types that EvoLP.jl provides for extending the type blocks are the following:

- **SelectionMethod**. The base of all selectors. It is used along with `select`.
- **CrossoverMethod**. The base of all recombinators. It is used along with `cross`.
- **MutationMethod**. The base of all mutators. It is used along with `mutate`.

Since generators and algorithm blocks are function blocks, they do not have an abstract supertype (unlike selectors, recombinators and mutators). To add a new generator or algorithm, the user needs only to provide a new function.

Regardless of the kind of functionality a user wishes to add, it is recommended to consider reproducibility. In EvoLP.jl, all blocks that deal with stochastic components (i.e., generators, selectors, recombinators and mutators) posses a keyword argument that can receive a Random Number Generator (RNG) instance. Using a `StableRNG` object (provided by the StableRNGs.jl package [30]), we can ensure that a fixed seed will always return the same results. With this addition in mind, experiments become reproducible and can be shared in an executable form.

5.7. Extra Utilities

As mentioned in Section 5.1, EvoLP.jl includes as well additional blocks that provide useful functionality for designing, testing and analysing EAs. For a description of its capabilities and usage, we invite the reader to refer to the full documentation.[5]

[5]See https://ntnu-ai-lab.github.io/EvoLP.jl/stable/

6. Conclusion and Future Work

In this paper we presented EvoLP.jl, a framework focused on providing reusable computing patterns for whenever scientists design and analyse multiple components for single-objective EC solvers. By using the metaphor of building blocks, we developed an abstraction that is easy for the user to understand. In this way, both using the framework and extending it becomes a user-friendly experience. We described all basic blocks that the framework includes, covering all phases of an evolutionary process—initialisation, selection, crossover and mutation—as well as referred to further reading for its optional features. The quality of the framework is ensured by following the conventions and sticking to the requirements of the Julia scientific community: unit testing, version control, extensive documentation and availability of the source code for everyone to use and modify. We believe that this is one of the strengths of the software tool (in addition to the EC niche it tackles inside the Julia community), and plan to continue the development of the package further in hopes that it becomes a useful tool for research, education and innovation.

For future releases of EvoLP.jl we contemplate the addition of more test functions, mostly in the pseudoboolean domain. Support for key-word arguments in constructors is also planned, as well as introducing support for multi-objective problems. Types for solution encoding, survival and replacement (niching, crowding and other diversity-preserving mechanisms) are being considered as potential block additions although further experimentation is required. Finally, a predefined set of metrics for the logbook block is also in the works. Since EvoLP.jl is a free and open-source project, the code is available for anyone to try. We invite the reader to play around and contribute through any of the available channels in the repository.

Acknowledgments

We would like to acknowledge the financial support for EvoLP.jl, partly funded by Project no. 311284 of the Research Council of Norway, as well as access to computing resources from the Department of Computer Science of the Norwegian University of Science and Technology.

We would also like to thank the Norwegian Open Artificial Intelligence Lab for the promotion and hosting of the framework in its GitHub organisation.

References

[1] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Series in Artificial Intelligence, Pearson, 2020.

[2] J. A. Gulla, et al., NorwAI Annual Report 2020, Technical Report 01, Norwegian Research Center for AI Innovation, 2020. URL: https://www.ntnu.edu/documents/1294735055/1298789471/NorwAi+annual+report+2020_Final_06-04-21.pdf.

[3] J. A. Gulla, et al., NorwAI Annual Report 2021, Technical Report 03, Norwegian Research Center for AI Innovation, 2021. URL: https://www.ntnu.edu/documents/1294735055/1298789471/NorwAi+annual+report+2021_Final_31-03-22.pdf.

[4] J. A. Gulla, et al., NorwAI Annual Report 2022, Technical Report 02, Norwegian Research Center for AI Innovation, 2022. URL: https://www.ntnu.edu/documents/1294735055/1298789471/NorwAi+annual+report+2022_Final_Web.pdf.
[5] Forskningsrådet, Statistics about AI - Prosjektbanken, 2023. URL: https://prosjektbanken.forskningsradet.no/en/explore/projects?Kilde=FORISS&distribution=Ar&chart=bar&calcType=funding&Sprak=no&sortBy=score&sortOrder=desc&resultCount=30&offset=0&source=FORISS&Fritekst=artificial%20intelligence&Ar=2018&Ar=2019&Ar=2020&Ar=2021&Ar=2022&Ar=2023&LTP.1=LTP2%20IKT%20og%20digital%20transformasjon.
[6] Forskningsrådet, Statistics about EAs - Prosjektbanken, 2023. URL: https://prosjektbanken.forskningsradet.no/en/explore/projects?Kilde=FORISS&distribution=Ar&chart=bar&calcType=funding&Sprak=no&sortBy=score&sortOrder=desc&resultCount=30&offset=0&source=FORISS&Fritekst=evolutionary&Ar=2018&Ar=2019&Ar=2020&Ar=2021&Ar=2022&Ar=2023&LTP.1=LTP2%20IKT%20og%20digital%20transformasjon.
[7] A. Telikani, A. Tahmassebi, W. Banzhaf, A. H. Gandomi, Evolutionary Machine Learning: A Survey, ACM Computing Surveys 54 (2021) 161:1–161:35. doi:10.1145/3467477.
[8] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, C. Hervás, JCLEC: A Java framework for evolutionary computation, Soft Computing 12 (2008) 381–392. doi:10.1007/s00500-007-0172-0.
[9] A. Ramírez, J. R. Romero, C. García-Martínez, S. Ventura, JCLEC-MO: A Java suite for solving many-objective optimization engineering problems, Engineering Applications of Artificial Intelligence 81 (2019) 14–28. doi:10.1016/j.engappai.2019.02.003.
[10] E. Medvet, G. Nadizar, L. Manzoni, JGEA: A modular java framework for experimenting with evolutionary computation, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 2009–2018. doi:10.1145/3520304.3533960.
[11] Heurstic, E. A. L. (HEAL), Heuristiclab, 2023. URL: https://github.com/heal-research/HeuristicLab.
[12] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: A python framework for evolutionary algorithms, in: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 85–92. doi:10.1145/2330784.2330799.
[13] Y. Tang, Y. Tian, D. Ha, EvoJAX: Hardware-Accelerated Neuroevolution, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2022, pp. 308–311. doi:10.1145/3520304.3528770. arXiv:2202.05008.
[14] M. A. Coletti, E. O. Scott, J. K. Bassett, Library for evolutionary algorithms in Python (LEAP), in: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1571–1579. doi:10.1145/3377929.3398147.
[15] F. Biscani, D. Izzo, A parallel global multiobjective framework for optimization: Pagmo, Journal of Open Source Software 5 (2020) 2338. doi:10.21105/joss.02338.
[16] C. Rackauckas, Sciemon, J. Vaverka, B. S. Zhu, V. L., A. Strouwen, D. P. Sanders, G. Sterpu, J. Ling, P. E. Catach, P. Monticone, W. Dey, V. Churavy, A. Edelman, A. Haslam, A. Lenail, A. Kaushal, C. Laforte, C. Wang, F. Cucchietti, K. Bhogaonker, L. Milechin, F. C. White,

M. Payne, S. Schaub, S. Fu, V. Meijer, W. Kirchgässner, A. Jain, Sciml/scimlbook: v1.1, 2022. URL: https://doi.org/10.5281/zenodo.7347643. doi:10.5281/zenodo.7347643.
[17] M. Lubin, O. Dowson, J. D. Garcia, J. Huchette, B. Legat, J. P. Vielma, JuMP 1.0: Recent improvements to a modeling language for mathematical optimization, 2023. doi:10.48550/arXiv.2206.03866. arXiv:arXiv:2206.03866.
[18] J.-A. Mejía-de-Dios, E. Mezura-Montes, Metaheuristics: A Julia Package for Single- and Multi-Objective Optimization, Journal of Open Source Software 7 (2022) 4723. doi:10.21105/joss.04723.
[19] R. Feldt, Blackboxoptim.jl, 2023. URL: https://github.com/robertfeldt/BlackBoxOptim.jl.
[20] P. Renc, P. Orzechowski, A. Byrski, J. Wăs, J. H. Moore, EBIC.JL: An efficient implementation of evolutionary biclustering algorithm in Julia, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1540–1548. doi:10.1145/3449726.3463197.
[21] D. G. Wilson, Cambrian.jl, 2023. URL: https://github.com/d9w/Cambrian.jl.
[22] Art, Evolutionary, 2023. URL: https://github.com/wildart/Evolutionary.jl.
[23] M. J. Kochenderfer, T. A. Wheeler, Algorithms for Optimization, Mit Press, 2019.
[24] H. H. Rosenbrock, An Automatic Method for Finding the Greatest or Least Value of a Function, The Computer Journal 3 (1960) 175–184. URL: https://doi.org/10.1093/comjnl/3.3.175. doi:10.1093/comjnl/3.3.175.
[25] S. Surjanovic, D. Bingham, Virtual library of simulation experiments: Test functions and datasets, Retrieved April 11, 2023, from http://www.sfu.ca/~ssurjano, 2013.
[26] H. Bambury, A. Bultel, B. Doerr, An Extended Jump Functions Benchmark for the Analysis of Randomized Search Heuristics (2022). URL: http://arxiv.org/abs/2105.03090. doi:10.1007/s00453-022-00977-1. arXiv:2105.03090.
[27] S. Droste, T. Jansen, I. Wegener, A rigorous complexity analysis of the (1+1) evolutionary algorithm for linear functions with Boolean inputs, in: 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360), 1998, pp. 499–504. doi:10.1109/ICEC.1998.700079.
[28] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st ed., Addison-Wesley Longman Publishing Co., Inc., USA, 1989.
[29] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95 - International Conference on Neural Networks, volume 4, 1995, pp. 1942–1948 vol.4. doi:10.1109/ICNN.1995.488968.
[30] JuliaRandom, StableRNGs, JuliaRandom, 2023. URL: https://github.com/JuliaRandom/StableRNGs.jl.