Self-Rewriting Differential Evolution with Operator Synthesis

Iztok Fister, Jr.
University of Maribor
Maribor, Slovenia
iztok.fister1@um.si

Aleksander Kalacun
University of Maribor
Maribor, Slovenia
aleksander.kalacun@student.um.si

Iztok Fister
University of Maribor
Maribor, Slovenia
iztok.fister@um.si

Abstract

Although nowadays there exist the so-called evolving algorithms, which are based on principles of biological evolution, such as reproduction, mutation, and selection, they still obey the von Neumann principle "Program = algorithm + data". Unfortunately, the algorithms are rigid in the sense that they remain unchanged in the computer's memory during execution, while their evolvability is mainly referred to the data applied. In this paper, we introduce a novel variant of the differential evolution evolving algorithm capable of performing code-level adaptation and breaking the wall of the algorithm's rigidity. In our proposed method, differential evolution mutation strategies are represented as short symbolic programs. Periodically, the system synthesizes new operators by mutating the top operators with operators like variable swaps, F-scaling, and additive difference terms, merges them into the operator library with deduplication and a size cap, and credits operators based on empirical success using a softmax function. After each phase, the proposed method self-rewrites the program by serializing the evolved operator library and best solution back into its own source code and re-executing, yielding heritable, cumulative changes to the program code itself. On canonical test benchmarks, the novel optimizer discovers effective, previously absent operators and improves across phases, illustrating a practical path toward structurally replicating and self-adapting.

CCS Concepts

• **Theory of computation → Design and analysis of algorithms**.

Keywords

differential evolution, code adaptation, self-rewriting algorithms

ACM Reference Format:
Iztok Fister, Jr., Aleksander Kalacun, and Iztok Fister. 2026. Self-Rewriting Differential Evolution with Operator Synthesis. In *Genetic and Evolutionary Computation Conference (GECCO Companion '26), July 13–17, 2026, San Jose, Costa Rica.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3795101.3805457

1 Introduction

Nowadays, the von Neumann principle of digital computer architecture is exposed as a bottleneck in creating the general Artificial Intelligence (AI), demanding that the algorithms need to be changed during execution, besides data [4]. Such self-improving algorithms can evolve through self-rewriting, i.e., the concept that enables algorithms to change during execution.

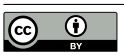
The concept of self-rewriting lies in a process of biological evolution of program code that operates through variation, inheritance, and selection. Living organisms replicate, and their inheritable traits change over time; those individuals that survive better determine which traits will be passed on to the next generation. Because humans are also the outcome of this evolutionary process, we display layered, cumulative improvements: our knowledge, technologies, and even societal structures develop and build upon each previous generation. In stark contrast, most computer programs or software are static artifacts; i.e., once compiled or interpreted, the *source code* that defines an algorithm typically does not change during or after its execution. Even adaptive or self-adaptive meta-heuristic optimizers modify internal state while the *implementation* that generates those updates remains fixed. This separation limits algorithms' capacity to accumulate, inherit, and refine the very mechanisms by which they search.

To the authors' knowledge, no prior studies directly address this question. The most closely related work includes paper [1], which explores adjacent aspects but does not examine the specific problem investigated here. In line with this, we present a simple but rigorous framework in which a stochastic population-based optimizer can:

- *modify its own source code* in response to the gained experience,
- *synthesize* new search operators during the run, retaining successful inventions across phases,
- *rewrite* and *restart* the new program code.

In a nutshell, we introduce *Self-Rewriting Differential Evolution* (SRDE), where DE mutation strategies are represented as short symbolic programs over population vectors and scalars. The mutation strategies compete under selection pressure, and the system periodically synthesizes novel strategies by editing the syntax of high-performing ones and merging them into the library. At phase boundaries, the updated operator set and the current best solution are serialized into marked blocks inside the program's *own source file*, and the process restarts the heritable, code-level change. The last step enables us to write self-rewriting code and is part of metaprogramming, included in some modern programming languages, such as Python.

The remainder of the paper is organized as follows: Section 2 familiarizes potential readers with the theoretical background of the study. In Section 3, the proposed method is discussed in detail. The experiments and the results are presented in Section 4, while the paper concludes with Section 5, which also outlines potential directions for future work.



GECCO Companion '26, San Jose, Costa Rica

ACM ISBN 979-8-4007-2488-6/2026/07
https://doi.org/10.1145/3795101.3805457

2 Theoretical Background

2.1 Differential Evolution

A Differential Evolution (DE) is a population-based Evolutionary Algorithm for continuous global optimization [5]. At its core, the DE algorithm is a simple yet very effective heuristic for solving various real-life problems. The idea of the DE algorithm is a mathematical model based on scaled vector differences. The DE algorithm evolves a population of vectors through consecutive generations g, where each vector represents a solution of the problem, either directly or indirectly. Normally, the DE population consists of Np real-coded vectors that can be defined mathematically as follows:

$$\mathbf{x}_i^{(g)} = (x_{i,1}^{(g)}, \ldots, x_{i,D}^{(g)}), \quad \text{for } i = 1, \ldots, Np, \tag{1}$$

where each element $x_{i,j}^{(g)}$ lies within the interval $[x_j^{(L)}, x_j^{(U)}]$ for $j = 1, \ldots, D$. Vectors $\mathbf{x}^{(L)}$ and $\mathbf{x}^{(U)}$ denote the lower and upper bounds of the particular problem variable $x_{i,j}^{(g)}$, while D refers to the problem dimension. In each generation, all the vectors in the population go through a set of evolutionary operators, i.e., mutation, crossover, and selection. A trial vector is produced after applying these operators, and then competes with the current vector (i.e., parent) for surviving into the next generation based on the fitness value. The mutation operator, executed for every vector in the population, is defined using the so-called DE mutation strategies that yield a mutant vector $\mathbf{u}_i^{(g)}$. The 'DE/rand/1/bin' mutation strategy means that the parent vectors $\mathbf{x}_{r_1}^{(g)}$, $\mathbf{x}_{r_2}^{(g)}$, and $\mathbf{x}_{r_3}^{(g)}$ are selected using the indices drawn randomly from uniform distribution in the interval $[1, Np]$, 1 vector difference is added to the vectors $\mathbf{x}_{r_1}$. while the number of parameters donated by the mutant vector follow a binomial distribution, in other words:

$$\mathbf{u}_i^{(g)} = \mathbf{x}_{r_1}^{(g)} + F \cdot (\mathbf{x}_{r_2}^{(g)} - \mathbf{x}_{r_3}^{(g)}), \tag{2}$$

where $i \neq r_1 \neq r_2 \neq r_3$. Let us mention that a variable $F \in [0.1, 1.0]$ is the scaling factor in the mentioned equation.

After generating the mutant vector $\mathbf{u}_i^{(g)}$, a crossover is performed according to the crossover rate Cr and the corresponding vector $\mathbf{x}_i^{(g)}$ from the population as follows:

$$v_{i,j}^{(g)} = \begin{cases} u_{i,j}^{(g)}, & \text{if } rand() \leq Cr \text{ or } j = j_{rand}, \\ x_{i,j}^{(g)}, & otherwise. \end{cases} \tag{3}$$

The crossover rate Cr, normally defined within the interval $[0, 1]$, controls the probability of parameters donated by the mutant vector $\mathbf{u}_i^{(g)}$. Additionally, the j_{rand} index ensures that at least one value from the mutant vector $\mathbf{u}_i^{(g)}$ is modified in the new trial vector $\mathbf{v}_i^{(g)}$.

The last operation is selection, in which the newly created trial vector $\mathbf{v}_i^{(g)}$ competes with its parent for survival into the next generation. In the case of minimization, the selection operator can be defined mathematically as:

$$\mathbf{x}_i^{(g+1)} = \begin{cases} \mathbf{t}_i^{(g)}, & \text{if } f(\mathbf{v}_i^{(g)}) \leq f(\mathbf{x}_i^{(g)}), \\ \mathbf{x}_i^{(g)}, & otherwise. \end{cases} \tag{4}$$

Let us mention that the selection operator in DE is typically noted as *one-to-one* selection.

3 Proposed method

3.1 Prerequisites

We extend the DE paradigm to accommodate the dynamic, evolving program premise. The DE is chosen for its robust framework, which allows us to modify discrete operations over generations. Given the basic DE mutation strategy presented in section 2.1, we initialize global operator library $\mathcal{L}^{(g)}$ in generation g, where every DE mutation strategy $o \in \mathcal{L}^{(g)}$ is a symbolic expression defining the **mutation step**. Formally, operators are linear combinations of terms over a search space S:

$$\mathcal{S} = \{\mathbf{x}_{r1}, \ldots \mathbf{x}_{rN}, \mathbf{x}^{\star}\}, \tag{5}$$

where $\mathbf{x}_{ri}$ for $i = 1, \ldots, N$ denotes N vectors drawn randomly for the library $\mathcal{L}^{(g)}$. To measure the performance of each operator $o \in \mathcal{L}^{(g)}$, we introduce bandit-style metrics: success number s_o (how many times the current best was outperformed) and total number of trials t_o. Combining the two, we define the empirical success rate to be more representative of the state, in other words:

$$r_o = \frac{s_o}{\max\limits_{i=1,\ldots,N} t_{o_i}}, \tag{6}$$

where M is the number of operators in the DE operator library. For instance, the operator $o \in \mathcal{L}^{(g)}$ for $N = 3$ conforming Eq. (2) is symbolically defined in the DE operator library as $\langle x_{r1}, x_{r2}, x_{r3}, F, r_o \rangle$, where elements x_{ri} for $i = 1, \ldots, N$ denotes symbolic definition of tree random selected vectors from the population, F the corresponding scale factor and s_o is the success number initially set to zero. The operator conforming the 'DE/rand/1/bin' mutation strategy in Eq. 2 serves as a starting point for synthesizing the new DE operators in the initial operator library $\mathcal{L}^{(0)}$.

As in standard DE, we keep track of the best operator in the population of operators, (i.e., $\mathbf{x}^{\star} \in \mathcal{P}^{(g)}$), while the library $\mathcal{L}^{(g)}$ update is performed every T generations.

3.2 The proposed algorithm in a nutshell

A diagram of a proposed algorithm, SRDE, is presented in Fig. 1, from which it can be seen that the algorithm consists of two modules: (1) control system, and (2) problem solving. The former is aimed at manipulating the program's code segment, while the latter is for normal DE problem-solving. Thus, the number of steps is denoted within the round brackets. In the remainder of the paper, the steps are described in more detail.

(1) Initialization and launch. The initialization of the SRDE captures saving the original DE program using the 'DE/rand/1/bin' mutation strategy, dividing it into more code blocks onto the disk, initialization of the DE operator library $\mathcal{L}^{(0)}$, and launching the DE program for optimization of the definite benchmark function.

(2) Success rate determination. The success number s_o and the total number of trials t_o, obtained after T generations using the specific DE mutation strategy, are passed from the program solving module to the control system. From this number, the success rate of the operator is calculated and updated in the DE operator library.

(3) Select symbolic operator. Symbolic operators for constructing the DE mutation strategy are selected from the DE operator library $\mathcal{L}^{(g)}$ based on the application of a softmax action selection

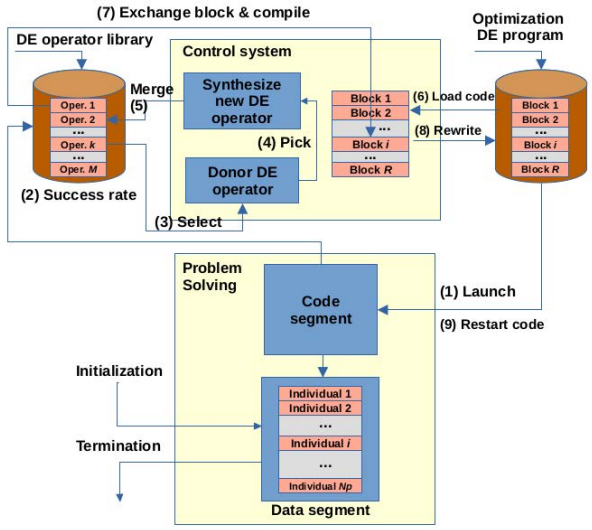


Figure 1: Mutation strategy construction in SRDE algorithm.

method. The method employs the notion of temperature, allowing us to tackle the exploration/exploitation problem efficiently. Probability of selecting a given operator from the library $o \in \mathcal{L}^{(g)}$ thus equals to:

$$p(o) = \frac{\exp\left(\frac{r_o - \max\limits_{i=1,\ldots,M} r_i}{\tau}\right)}{\sum_{k=1}^{N} \exp\left(\frac{r_k - \max r_{o'}}{\tau}\right)}, \tag{7}$$

where M denotes the maximum number of operators in library $\mathcal{L}^{(g)}$, and τ is a small predetermined constant temperature that skews the distribution by adding randomness in operator selection, forcing exploration of trailing new operators instead of exploiting existing ones.

(4) Synthesize new DE operator. Each individual in SRDE is represented as a real-valued vector:

$$\mathbf{x}_i = (\underbrace{x_{i,r1}}_{base}, \underbrace{x_{i,r2}, x_{i,r3}}_{\text{term 1}}, \ldots, \underbrace{x_{i,r(L-Q-3)}, x_{i,r(L-Q-2)}}_{\text{term Q}}, \underbrace{x_{i,r(L-Q-1)}}_{F_{i,1}}, \ldots, \underbrace{x_{i,r(L-1)}}_{F_{i,Q}}, \underbrace{x_{i,rL}}_{Cr_i}) \tag{8}$$

where Q designates the number of additive terms higher than zero, $L = 3 \cdot Q + 2$ is the length of the vector. while $x_{i,rj} \in S$ for $j = 1, \ldots, L - Q - 2$ are operators $o \in \mathcal{L}^{(g)}$, parameters $F_{i,j}$ for $j = 1, \ldots, Q$ are DE scale factors, and Cr_i is the crossover rate.

After operator selection, new the so-called donor DE operator $\mathbf{x}_{donor}$ is selected randomly from the symbolic operator library $\mathcal{L}^{(g)}$.

Indeed, the new DE operator is synthesized from S among the top few symbolic operators from the library $\mathcal{L}_{top} \in \mathcal{L}$ according to their success rates r_o in the following three steps:

i) Operator swap: The function $swap(x_{donor,j}, x_{donor,k})$ exchange j-th and k-th element in donor vector, where $\langle j, k \rangle \in [1, 2Q{+}1]$ and $j \neq k$.
ii) Coefficient tweak around F: The scaling factor is modified as $F = k \cdot F$, where the variable is randomly drawn from a set of feasible values, i.e, $k \in \{0.3, 0.5, 0.8, 1.2, 1.5\}$.
iii) Additive difference term: It is constructed on the basis of decoded operators from the donor vector. In the case $N = 3$, mutation strategy is expressed: $'x_{donor,r1} + F\left(x_{donor,r2} - x_{donor,r3}\right)'$.

(5) Merge new DE operator. In this step, the new symbolic DE operator is merged into library $\mathcal{L}^{(g)}$ with exact-expression deduplication (i.e., if the same copy of operator from donor vector $o \in \mathbf{x}_{donor}$ already exists in $\mathcal{L}$, it is replaced) and a pool cap $|\mathcal{L}| \leq M$ (i.e., if the number of the operators is higher than M, the worst operator regarding the success rate s_o is deleted from the archive $\mathcal{L}^{(g)}$). The initial success rate is set to $r_o = 0.0$.

Self-rewrite procedure. The procedure captures the sequence of the following steps in the figure:

(6) loading the program divided into code blocks from disk,
(7) locating the proper code block, exchanging it with the symbolic DE operator definition,
(8) compiling the code specification and rewriting the compiled program to disk,
(9) restart the problem-solving code segment again.

4 Experiments

We compare the proposed SRDE against the original DE algorithm. Thus, the goal of the experiments is to see whether the SRDE incorporating the self-rewriting feature can produce the results that are comparable if not better to the results obtained with the baseline by optimizing five well-known benchmark functions: (f_1) Rastrigin, (f_2) Ackley, (f_3) Rosenbrock, (f_4) Schwefel, and (f_5) Griewank [2]. All benchmark functions were of dimension $D = 10$.

In order to make the comparison as fair as possible, the common parameters of the original DE and SRDE were set identically, as: the population size $Np = 50$, the total number of generations $max\_gen = 200$, and the crossover rate $Cr = 0.5$. Let us emphasize that the crossover rate was held fixed during the experimental work by SRDE. The other specific SRDE parameters were as follows: the number of generated random vectors $N = 5$, the number of additional terms $Q = 2$, and the maximum size of the symbolic operator library $M = 12$. Thus, the scaling factor F can be drawn from the set $k \in \{0.0, 0.3, 0.5, 0.8, 1.0, 1.2\}$, where the following relation needs to be satisfied: $\neg(F_1 = 0.0 \wedge F_2 = 0.0)$.

The code has been implemented in Python 3.12, and benchmarks were taken from the NiaPy library [6] using the other standard Python libraries (e.g., numpy). The benchmarks in the paper were run on an AMD Ryzen 5 5600XT running Fedora Linux. We also ran test on HPC using Singularity for further problem expansion.

In the preliminary study, two experiments were performed:

- evolving of DE mutation strategies throughout generations,
- results of SRDE by optimization benchmark functions,

The results of the mentioned experiments are discussed in the remainder of the paper.

4.1 Evolving of DE mutation strategies

The experiment was aimed at showing that the top three DE mutation strategies from the symbolic operator library $\mathcal{L}_{top} \subset \mathcal{L}$ really evolve during the evolutionary search of the SRDE, and thus the best fitness function values converge towards zero. In Table 1, a random operator evolution can be observed by optimizing the

Schwefel function benchmark function, where the best operator decreased from 2215.88 to 810.43 from gen 10 to 200 during one run.

Table 1: Evolving of DE mutation strategies on Schwefel function (10D, population 50)

gen	r_o	Mutation strategy	Best f
10	0.00	$x_{best} + 0.3F(x_{r2} - x_{r1}) + 0.5F(x_{r4} - x_{r3})$	2215.88
	0.00	$x_{best} + 1.0F(x_{r1} - x_{r2}) + 0.5F(x_{r4} - x_{r5})$	
	0.00	$x_{r2} + F(x_{r1} - x_{r3})$	
40	0.01	$x_{r1} + 1.5F(x_{r2} - x_{r5}) + 1.5F(x_{r3} - x_{r4})$	2061.08
	0.00	$x_{r4} + 1.0F(x_{r1} - x_{r2})$	
	0.00	$x_{r2} + 0.3F(x_{r1} - x_{r5}) + 1.0F(x_{r4} - x_{best})$	
70	0.01	$x_{r1} + 1.5F(x_{r2} - x_{r5}) + 0.5F(x_{r3} - x_{r4})$	1982.47
	0.01	$x_{r1} + 1.5F(x_{r2} - x_{r5}) + 1.5F(x_{r3} - x_{r4})$	
	0.00	$x_{r1} + 1.0F(x_{r2} - x_{r5}) + 1.5F(x_{r3} - x_{r4})$	
130	0.01	$x_{best} + 1.2F(x_{r1} - x_{r3}) + 0.3F(x_{r5} - x_{r2})$	1850.45
	0.01	$x_{best} + 0.3F(x_{r2} - x_{r5})$	
	0.01	$x_{best} + 0.8F(x_{r2} - x_{r4}) + 0.8F(x_{r3} - x_{r5})$	
170	0.01	$x_{best} + 0.3F(x_{r2} - x_{r5})$	1517.67
	0.01	$x_{r3} + 1.2F(x_{best} - x_{r5})$	
	0.01	$x_{best} + 1.2F(x_{r1} - x_{r3}) + 0.3F(x_{r5} - x_{r2})$	
200	0.02	$x_{best} + 0.3F(x_{r2} - x_{r5})$	**810.43**
	0.01	$x_{r5} + 0.8F(x_{r2} - x_{r3})$	
	0.01	$x_{r3} + 1.2F(x_{best} - x_{r5})$	

As is observed from the table, the success rates r_o increase with increasing the generations. Also, the formulation of the best DE mutation strategies became more complex by such evolving.

Interestingly, the code grew in size by 340-450 bytes during an independent run optimizing a typical benchmark function. This fact justifies the claim that the employed DE mutation strategies have evolved and become more sophisticated.

4.2 Results of SRDE by optimization benchmark functions

The purpose of the experiment was to illustrate that the results of the proposed SRDE are comparable, if not better than, the results of the original DE. In line with this, the results obtained by the proposed SRDE and original DE algorithms are estimated according to the five statistical metrics aggregated after 25 individual runs for each of the five observed benchmark functions in Table 2, where

Table 2: Results of experiments on the five benchmark functions achieved by the SRDE algorithm.

Alg.	Value	f_1	f_2	f_3	f_4	f_5
DE	Best	1.53E+01	1.45E-03	5.23E+00	1.26E+03	1.51E-01
	Worst	3.57E+01	3.14E-03	6.76E+00	1.73E+03	5.69E-01
	Mean	2.84E+01	2.24E-03	5.95E+00	1.55E+03	4.77E-01
	Median	3.01E+01	2.22E-03	5.98E+00	1.55E+03	5.08E-01
	StDev	5.49E+00	4.65E-04	4.14E-01	1.54E+02	1.16E-01
SRDE	Best	**6.96E+00**	**1.47E-14**	**3.81E-08**	**3.57E+02**	**9.86E-03**
	Worst	4.63E+01	1.06E-02	5.75E+00	1.37E+03	5.22E-01
	Mean	**2.52E+01**	**1.09E-03**	**1.89E+00**	**9.19E+02**	**2.01E-01**
	Median	2.62E+01	1.60E-06	1.51E+00	1.01E+03	1.15E-01
	StDev	1.33E+01	3.18E-03	1.82E+00	3.38E+02	1.82E-01

the better results regarding the minimum and mean metrics are denoted in bold. As is evident, the SRDE outperformed the results achieved by the original DE according to the bold values.

To also evaluate the results in the table statistically, the Wilcoxon nonparametric test was performed with the significance level $\alpha = 0.5$. The p-value of 0.04236 shows that the results of the SRDE are also significantly better than those achieved by the original DE.

5 Discussion and conclusions

In this paper, we took a totally new look at evolving the evolutionary algorithms in the sense of the adaptation of the algorithm on a program code basis, where we proposed several unique contributions:

- **Code-level adaptation.** We formulated and implemented a self-rewriting mechanism that embeds the current best solution and the evolved operator library directly into the source code, conferring heritability of algorithmic changes across executions.
- **Operator synthesis.** We represent mutation rules as editable symbolic expressions and introduce simple, composable syntactic edits (variable swaps, F-scaling, additive difference terms) that generate genuinely *new* operators during the run.
- **Simplicity and cost.** The method is lightweight and is easy to reproduce since the learned operator set is part of the executable artifact.

The preliminary test on five benchmark functions revealed that the proposed concept of self-rewriting program code works for DE. This encourages us to continue our research in this domain, also in the future.

Obviously, we have many potential directions for the future: We encourage further analysis using other metaheuristic algorithms. The influence of the SRDE parameters on the behavior of the algorithm should be explored in more detail. During the experimental work, we found out that the concept of self-rewriting could be well-suited in conditions of open-ended evolution, where researchers search for conditions where evolving systems never settle into a single stable equilibrium [3]. However, this speculation needs a huge elaboration in the future.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used language tools such as Grammarly and Lumo (Proton AG) in order to improve the article's readability. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

[1] Iztok Fister Jr and Iztok Fister. 2023. Towards replicated algorithms. *arXiv preprint arXiv:2304.13524* (2023).

[2] Momin Jamil and Xin-She Yang. 2013. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation* 4, 2 (2013), 150–194.

[3] Norman Packard, Mark A. Bedau, Alastair Channon, Takashi Ikegami, Steen Rasmussen, Kenneth Stanley, and Tim Taylor. 2019. Open-ended evolution and open-endedness: Editorial introduction to the open-ended evolution I special issue. *Artif. Life* 25, 1 (April 2019), 1–3. doi:10.1162/artl_e_00282

[4] Herbert L. Roitblat. 2020. *Algorithms Are Not Enough: Creating General Artificial Intelligence.* The MIT Press. doi:10.7551/mitpress/11659.001.0001

[5] Rainer Storn and Kenneth Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *J. of Global Optimization* 11, 4 (dec 1997), 341–359. doi:10.1023/A:1008202821328

[6] Grega Vrbančič, Lucija Brezočnik, Uroš Mlakar, Dušan Fister, and Iztok Fister Jr. 2018. NiaPy: Python microframework for building nature-inspired algorithms. *Journal of Open Source Software* 3 (2018). Issue 23. doi:10.21105/joss.00613