

# ELEA – Build your own Evolutionary Algorithm in your Browser

Markus Wagner<sup>1</sup>, Erik Kohlros<sup>2</sup>, Gerome Quantmeyer<sup>2</sup>, Timo Kötzing<sup>2</sup>

<sup>1</sup> Monash University, Australia; <sup>2</sup> Hasso Plattner Institute/University of Potsdam, Germany

markus.wagner@monash.edu, erik.kohlros@gerome.quantmeyer@student.hpi.uni-potsdam.de, timo.koetzing@hpi.de

## ABSTRACT

We provide an open source framework to experiment with evolutionary algorithms which we call *Experimenting and Learning toolkit for Evolutionary Algorithms (ELEA)*. ELEA is browser-based and allows to assemble evolutionary algorithms using drag-and-drop, starting from a number of simple pre-designed examples, making the startup costs for employing the toolkit minimal. The designed examples can be executed and collected data can be displayed graphically. Further features include export of algorithm designs and experimental results as well as multi-threading.

With the very intuitive user interface and the short time to get initial experiments going, this tool is especially suitable for explorative analyses of algorithms as well as for the use in classrooms.

## KEYWORDS

Benchmarking; education; tool

### ACM Reference Format:

Markus Wagner<sup>1</sup>, Erik Kohlros<sup>2</sup>, Gerome Quantmeyer<sup>2</sup>, Timo Kötzing<sup>2</sup>. 2023. ELEA – Build your own Evolutionary Algorithm in your Browser. In *Conference 2023*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

The *Free Evolutionary Algorithm Kit (FrEAK)* is a “toolkit to design and analyse evolutionary algorithms, written in Java” [1], developed initially at the University of Dortmund. The last update to the repository was 10 years ago, and technology has evolved significantly since then. With this demonstration paper we introduce a web-based alternative, using the *Blockly* framework developed by Google [2, 5].

Our intention is not to transfer all and only the functionality of FrEAK to the web, but rather have a completely fresh start, offering functionalities directed at making this tool (1) a starting point for scientific investigations, giving quick insights with little coding overhead; as well as (2) a valuable helper for teaching evolutionary algorithms in the classroom, focusing students’ activities on understanding evolutionary algorithms rather than coding them. These two goals aim at making evolutionary algorithms more accessible to a wide range of people, lowering the costs required to get first results. We call our toolkit *Experimenting and Learning toolkit for Evolutionary Algorithms (ELEA)*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

Conference’23, .

© 2023 Association for Computing Machinery.  
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM... \$15.00  
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Specifically, ELEA offers the following features:

- Visual Programming System based on Blockly [2, 5].
- Several standard algorithms and test functions implemented.
- Export of Code to JavaScript for offline modification and use.
- Data visualisation.
- Data download as csv or for IOHAnalyzer [4].
- Multi-threading support.
- Open source, can be forked.

The visual programming system based on Blockly allows new users to build their own evolutionary algorithms with very little training (see Section 2 for a discussion on such programming systems). The pre-implemented algorithms and test functions enable first results with the toolkit within minutes. Also, for experienced programmers, using ELEA results in smaller overhead times, because only the EA needs to be designed, and data handling is almost completely taken care of by the system.

In order to allow for far-reaching analyses, the JavaScript sources are available for download, which can then be modified at will for analyses not covered by the functionalities of ELEA. As a core feature, ELEA can graphically display data at run time, making the data easily accessible.

We see two main use cases for ELEA, fundamental research and teaching EAs, which we discuss in turn.

**1. Use for fundamental research.** In the spirit of fast prototyping, ideas or hunches can be fact-checked quickly for small problem sizes using ELEA’s existing blocks, but also custom blocks can be designed quickly. Rigorous investigations with multiple repetitions are supported by multi-threading and data management. Furthermore, more complicated examples can be set up by downloading the code and implementing the necessary changes by editing the code directly. We believe that also the run time analysis community can profitably use ELEA.

**2. Use for teaching EAs.** Teachers and students can explore in-class the effects of algorithmic design decisions. The focus is on designing evolutionary algorithms rather than spending time with data management and program syntax. Colour-coded blocks assist students to associate colours and shapes [7] which in turn can help build understanding and retain knowledge. The browser-based environment avoids cross compatibility issues and gives a uniform standard for teachers and students for algorithm definition and benchmarking. At its best, ELEA’s building blocks invite students and teachers to “play” with algorithmic components, much like with wooden building blocks or Legos, to explore algorithmic spaces in playful and systematic ways.

You can find a running instance of ELEA as well as its source code at the following URLs:

<https://elea-toolkit.netlify.app/>  
<https://github.com/HPI-ELEA/elea>

Another code base that allows to benchmark evolutionary algorithms is the recently developed *Iterative Optimization Heuristics Profiler (IOHprofiler)* [4]. This modular software offers not just evolutionary algorithms, but also EDAs and other optimisation heuristics, as well as a modular approach to the profiling pipeline, including aspects such as automated algorithm configuration. In contrast to the IOHprofiler, ELEA is a lightweight tool based on an easy-to-access graphical user interface. This significantly lowers the initial cost to start using ELEA. In particular, IOHprofiler does not allow for defining algorithms, but a collection of pre-defined algorithms is provided. One of the core parts of IOHprofiler is the IOHalyzer, which offers a wide functionality for investigating the data produced by running algorithms on test functions. This nicely complements ELEA, which can produce compatible data.

There are also programming packages available to run evolutionary algorithms, for example LEAP [3], written in Python. LEAP “is a general purpose Evolutionary Computation package that combines readable and easy-to-use syntax for search and optimization algorithms with powerful distribution and visualization features.” While LEAP might be better suited for large scale benchmarking, we believe that the graphical display of the algorithm in ELEA helps understand the underlying principles better than a library call.

The remainder of this paper starts with background on visual programming (in Section 2), proceeds to give an example of using ELEA (in Section 3), before discussing some details of the system design (in Section 4), including multi-threading (see Section 5).

## 2 BACKGROUND ON VISUAL PROGRAMMING

A visual programming system (VPS) [8] allows users to create programs by manipulating program elements graphically rather than specifying them textually. In a VPS, a user creates a program by arranging “boxes and arrows”, where boxes represent entities and arrows represent relations.

A VPS can assist programmers to overcome three cognitive challenges [6]:

- Syntactic: arranging programming language components into well-formed programs.
- Semantic: assisting users with the comprehension of the meaning of programs.
- Pragmatic: bringing a program into a specific situation and understanding its behaviour.

Blockly [2, 5] is an example of such a VPS. It is an open-source, client-side library for the programming language JavaScript, providing an editor representing coding concepts as interlocking blocks. Blockly typically runs in a web browser, but it can also generate correct stand-alone code in JavaScript, Python, PHP, Lua, Dart, etc.

Figure 1 shows an example of Blockly running in a browser. The default graphical user interface of the Blockly editor consists of (1) a toolbox, which holds available blocks, and where a user can select blocks; and (2) a workspace, where a user can drag, drop and rearrange blocks. The workspace also includes, by default, zoom icons, and a trash can to delete blocks. Assembly of code consists in drag and drop of functional blocks, giving a final visual impression much like pseudo-code.

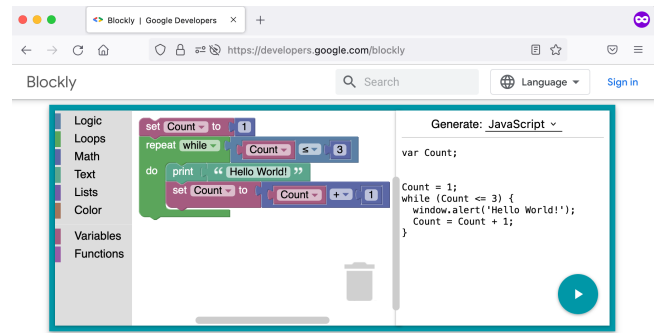


Figure 1: Blockly running in a browser (taken from [5]).

For ELEA, we leverage that Blockly is open-source and that custom blocks can be created. Each block consists of a definition, which defines the visual appearance, and a generator, which describes the block’s translation to executable code. Blocks can be written in JavaScript, but they can also be defined using blocks.

## 3 WHAT’S IN ELEA?

ELEA provides a number of new blocks pertaining to defining evolutionary algorithms, such as blocks for mutation, crossover, selection and so on. To demonstrate that it is easy (1) to define an algorithm using ELEA’s building blocks and (2) to run it and to plot the results, we show a complete example of a population-based evolutionary algorithm in Figure 2. Details of the particular scenario can be found in the figure’s caption.

In the following, we will focus our attention on the coloured building blocks that are used to construct the scenario. At present, ELEA provides 11 groups of building blocks, and we outline each group’s content next. Information on all blocks can be found in ELEA’s online documentation.

*Population (magenta red)* is the largest group of blocks. It provides blocks to initialise a population, such as randomised or explicitly specified. It also provides ways to “query” a population, i.e. to acquire information like the size of a population, to get information on the best individual in a population, or to select an individual from a population via methods like uniformly random selection or fitness-proportionate selection. This group also provides blocks to add individuals to a population, to merge or to sort them. *Individuals (green)* is the second largest group of blocks. They support the initialisation of individuals (randomised or explicitly specified), as well as the crossover (one-point, two-point, uniform) and mutation (mutating each bit with a given probability or mutation a given number of bits). *Fitness measure (cyan)* provides a number of blocks that calculate fitness functions like OneMax, LeadingOnes, Jump, and some diversity-based metrics. *Primitive datatypes (teal-blue)* allow us to create and set variables, to define and concatenate strings, and to create random numbers within a provided range. *Logic (teal-blue)* allows us to perform conditional executions (akin to an “if” in many programming languages) and to assemble Boolean expressions involving Boolean operators including equivalence checks. *Loops (teal-blue)* enable the simple definition of evolutionary loops, the (number-limited) repetition of selection (e.g. for selecting or creating solutions). It also includes a special block that aggregates

The screenshot displays the ELEA web interface for a 'simple\_plotting' example. On the left is a toolbox with categories like Population, Individuals, Fitness measure, Primitives, Logic, Loops, Functions, Logging, Multi-Threading, Time management, and experimental. The main workspace contains a Blockly-style code editor with the following logic:
 

- Initialize program** (circled 1): A 'do' block containing 'set genome\_length to 20', 'set fitness to 0', 'set lambda to 10', 'set population offspring\_population to create empty list', and 'run\_algorithm'.
- Run algorithm** (circled 2): A 'while' loop with condition 'no individual from parent\_population has fitness of at least genome\_length and rounds less than 300'. Inside, a 'for' loop '2 times, breed:' contains:
  - select individual based on fitnessproportionately from parent\_population (parent1, parent2)
  - crossover one-point parent1 with parent2 into offspring1 and offspring2
  - mutate individual offspring1 with a bit-wise probability of 0.05
  - mutate individual offspring2 with a bit-wise probability of 0.05
  - add offspring1 to offspring\_population
  - add offspring2 to offspring\_population
  - set population parent\_population to generate new population based on fitness from merge parent\_population with offspring\_population
- Plot** (circled 3): 'Plot fitness with: individual select best individual from population parent\_population' in 'Dataset: 1' in 'Plot: fitness as linegraph'.
- Print** (circled 4): 'print to output select best individual from population parent\_population' and 'print to output fitness with: individual select best individual from population parent\_population'.

 The right panel shows a 'Run' button, 'Abort', 'Show Code', 'Clear Output', and 'Show Tutorial'. Below the code is a 'fitness' plot (circled 5) with 'Hide' and 'Delete' buttons. The plot shows five data series (Dataset1 to Dataset5) representing 15 independent runs, with fitness increasing from 0 to approximately 20 over 60 generations. The plot is titled 'fitness' and has 'Hide' and 'Delete' buttons.

**Figure 2: ELEA example “Simple Plotting”.** We perform 15 independent repetitions ① of a (10 + 10)-EA that uses one-point crossover and uniform, bit-wise mutation. Individuals are bit strings of length 20. The solved problem is the OneMax problem (③: “sum of list individual”). The best individual at the end of each independent run is shown in the top right corner ④, and the quality of the best population member is plotted in the bottom right corner ⑤ (x-axis: generations, y-axis: quality). Note: while only a part of the legend is shown, all 15 runs are plotted.

data for the later export to IOHAnalyzer. *Functions* (lavender) comprises special, all-embracing blocks that are needed, for example, to run studies, like the multiple repetition of an algorithm. *Logging* (grey) provides blocks to print data to the output area of ELEA, to plot data in a scatter/line/bar graph, and to simply add a comment to an algorithm (akin to in-line code comments). *Multi-Threading* (blue cyan) enables the parallel execution of blocks. As we see this as an important feature, we provide in-depth details in Section 5. *Time management* (copper) contains blocks for pausing the execution for a number of seconds and for measuring wall-clock time. *Experimental* (black) covers blocks that do not fit elsewhere.

All ELEA blocks can be dragged and dropped into place, which lets you quickly assemble an algorithm or modify an existing one. Also, blocks can only be connected in particular ways, resulting in code that is always syntactically correct; for example, it is impossible to insert a logging command into a “parent spot” of the one-point crossover. Lastly, to further increase usability, blocks that are not connected to anything are greyed out, thus making it clear what is executed and what is not.

## 4 SYSTEM DESIGN

ELEA runs on Node.js and its user interface is built with Bootstrap. The whole tool is based around the Blockly framework, which

is developed by Google. Blockly creates an encapsulated module representing the workspace used in our tool and lets us easily build a website around it. In general, it is possible to customise our own blocks using either JavaScript or XML and provide these blocks to the Blockly module. There, we specify how our block should look, meaning for example what parameters can be put in or how the block connects to other blocks, and also provide a snippet of JavaScript code, which is executed in place of the actual block, when the user runs their code inside the tool. This code makes up the algorithm that can be downloaded from the tool. We can also configure how the toolbox looks by adding categories and arranging the blocks in a fitting way. Everything else, i.e. the moving and combining of blocks, is handled internally by Blockly. For more details see the Google Developers Documentation on Blockly [5].

Inside Blockly, the algorithm in block-form is represented using XML. This XML can be downloaded and later again uploaded into the workspace. Because every XML element used in this format references a snippet of code, our tool can easily transform this XML data to working JavaScript code.

Since we are using Node.js, you can download your algorithm as JavaScript including a suitable run time environment, and play around without the blocks locally.

## 5 MULTI-THREADING

To further improve the user experience, ELEA provides multi-threading blocks. Among other, these can be used to efficiently run multiple independent runs of an algorithm, which is a common task undertaken in research on randomised algorithms.

As an implementation detail, it is important to note that threads in JavaScript have their own scope, which means that they cannot directly use global variables. While this means that worker threads are “naturally” separated and thus run independently of each other, it also implies a need to implement (1) information import (e.g. to set a mutation rate outside of the actual algorithm, instead of hard-coding it inside), and (2) information sharing back to the main thread (e.g. for data logging purposes).

Figure 3 shows a complete example of multi-threading via our multi-threading blocks; in ELEA, you can find this as the example “Multi-Threading Performance Test”. While this example repeatedly calculates the 42nd Fibonacci number as a dummy task ( $1 \leq i \leq 30$  repetitions form one task), the online edition of ELEA contains an example using an evolutionary algorithm. In particular, this performance test compares three approaches: (1) the sequential execution of  $i$  repetitions using one thread, (2) the completely parallel execution (where  $i$  threads are started in parallel), and (3) the limited setup, where  $x$  parallel threads are used (here: the block “Hardware Concurrency” sets  $x$  to the number of CPU cores). The last approach can also be limited to, e.g. two worker threads (even when a machine has more cores) to allow the user to retain a responsive computer.

On our computer (a 2022 Macbook Air M2 with 8 CPU cores), the output in ELEA’s output window looks like this:

```
threads,num_iterations,num_threads,time
one thread,1,1,2136
all threads,1,1,2045.800000011921
limited threads,1,8,2140.5
-----
one thread,2,1,4122.900000035763
all threads,2,2,2139
limited threads,2,8,2089.7999999523163
-----
one thread,3,1,6250.899999976158
all threads,3,3,2173.899999976158
limited threads,3,8,2196.100000023842
[...]
```

Figure 4 shows the results, in particular how the total required time per task changes as the number of  $i$  repetitions increases. As expected, the wall-clock time for the sequential approach (using only one thread) increases approximately linearly. For the approach where  $i$  threads (i.e. up to 30 here) perform the calculations in parallel, run time also increases approximately linearly, although run times are about 70% faster for large  $i$ , resulting in a 3.4-fold speedup. For the limited approach, we can see “steps” in run time increase for the limited approach when multiples of eight are exceeded.

## ACKNOWLEDGMENTS

We thank Antony Kamp, Bjarne Sievers, and Oscar Manglaras for their contributions to early versions of ELEA. We thank Xiaoyue Li for her feedback on an earlier version.

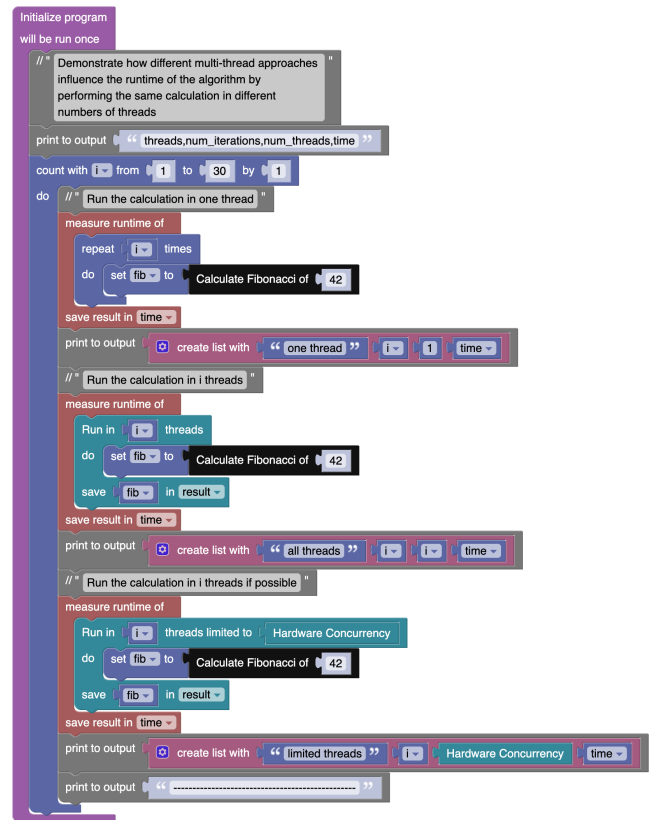


Figure 3: Multi-threading: complete example with a dummy task. The teal-coloured, C-shaped block “Run in  $i$  threads” are responsible for starting threads and for returning data back to the main thread.

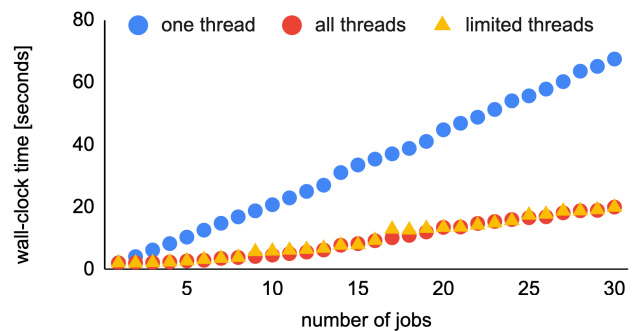


Figure 4: Multi-threading: performance results of a simple example. Each data point shows the total wall-clock time of an independent experiment involving the independent calculation of  $i \in \{1..30\}$  tasks.

## REFERENCES

- [1] Andrea, Dimo Brockhoff, Christian Gunia, Matthias Englert, Oliver Heering, Michael Leifhelm, Heiko Röglin, Patrick Briest, Dirk Sudholt, Stefan Tannenbaum, and Thomas Jansen. 2013. Free Evolutionary Algorithm Kit (FrEAK). Retrieved January 31, 2023 from <https://sourceforge.net/projects/freak427/>
- [2] Lucy Black. 2012. Google Blockly – A Graphical Language with a Difference. Retrieved January 31, 2023 from <https://www.i-programmer.info/news/98-languages/4357-google-blockly-a-graphical-language-with-a-difference.html>
- [3] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. 2020. Library for Evolutionary Algorithms in Python (LEAP). In *Proc. of GECCO'20*. ACM, 1571–1579. <https://doi.org/10.1145/3377929.3398147>
- [4] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. 2018. IOH-profiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. *CoRR* abs/1810.05281 (2018). <http://arxiv.org/abs/1810.05281>
- [5] Google. 2023. Blockly landing page. Retrieved January 31, 2023 from <https://developers.google.com/blockly/>
- [6] Alexander Repenning. 2017. Moving Beyond Syntax: Lessons from 20 Years of Blocks Programming in AgentSheets. *Journal of Visual Lang. and Computing* 3 (2017), 68–91. <http://ksiresearch.org/vlss/journal/VLSS2017/vlss-2017-repenning.pdf>
- [7] David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *Proc. of IDC'15*. ACM, New York, NY, USA, 199–208. <https://doi.org/10.1145/2771839.2771860>
- [8] Annie S Wu, Kenneth A De Jong, Donald S Burke, John J Grefenstette, and C Loggia Ramsey. 1999. Visual analysis of evolutionary algorithms. In *IEEE Congress on Evolutionary Computation*. IEEE, 1419–1425.