Operon C++: an Efficient Genetic Programming Framework for Symbolic Regression

Bogdan Burlacu
bogdan.burlacu@fh-ooe.at
Josef Ressel Centre for Symbolic Regression
Heuristic and Evolutionary Algorithms Laboratory
Hagenberg, Austria

Gabriel Kronberger
gabriel.kronberger@fh-ooe.at
Josef Ressel Centre for Symbolic Regression
Heuristic and Evolutionary Algorithms Laboratory
Hagenberg, Austria

Michael Kommenda
michael.kommenda@fh-ooe.at
Josef Ressel Centre for Symbolic Regression
Heuristic and Evolutionary Algorithms Laboratory
Hagenberg, Austria

ABSTRACT

Genetic Programming (GP) is a dynamic field of research where empirical testing plays an important role in validating new ideas and algorithms. The ability to easily prototype new algorithms by reusing key components and quickly obtain results is therefore important for the researcher.

In this paper we introduce *Operon*, a C++ GP framework focused on performance, modularity and usability, featuring an efficient linear tree encoding and a scalable concurrency model where each logical thread is responsible for generating a new individual. We model the GP evolutionary process around the concept of an *offspring generator*, a streaming operator that defines how new individuals are obtained. The approach allows different algorithmic variants to be expressed using high-level constructs within the same generational basic loop. The evaluation routine supports both scalar and dual numbers, making it possible to calculate model derivatives via automatic differentiation. This functionality allows seamless integration with gradient-based local search approaches.

We discuss the design choices behind the proposed framework and compare against two other popular GP frameworks, DEAP and HeuristicLab. We empirically show that *Operon* is competitive in terms of solution quality, while being able to generate results significantly faster.

CCS CONCEPTS

• **Computing methodologies** → *Supervised learning by regression*; *Parallel algorithms*; **Genetic programming**; • **Software and its engineering** → Abstraction, modeling and modularity;

KEYWORDS

genetic programming, C++, symbolic regression

ACM Reference Format:
Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. 2020. Operon C++: an Efficient Genetic Programming Framework for Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2020 (GECCO '20 Companion).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377929.3398099

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398099

1 INTRODUCTION

On the surface, implementing a genetic programming system is deceptively simple: all one needs to do is define an appropriate primitive set, a couple of genetic operators, and a fitness function. However, in reality several implementation aspects are non-trivial:

Solution encoding In tree-based GP, implementations using a `Node` class with pointers/references to parent/child nodes exhibit poor cache locality and may cause cache misses. Furthermore, design choices such as including heap-allocated objects (for example, strings data members for a node's name or description) into the node structure will increase memory pressure and slow down the copy/clone operations frequently performed by a GP system.

Solution evaluation Real-time constraints on the bytecode (just-in-time) compiler in managed languages[1] such as C# Java or Python leave their respective runtimes fewer opportunities for optimized code generation. In particular, they are less likely to use SIMD[2] (Single Instruction Multiple Data) [8] instructions that can improve efficiency in computationally intense workloads such as the evaluation routine. For this reason, computational bottlenecks in managed code are usually addressed by delegating calculations to native libraries that are specifically designed and optimized for this purpose, as for example Python's NumPy [21] module which calls into heavily optimized C and Fortran numerical routines.

Parallelization Genetic programming is an embarassingly parallel[3] process where a variety of approaches ranging from manual thread management to thread pools to distributed computing may be used. We distinguish two levels of parallelism in computationally-intensive tasks: *task-level* and *data-level* parallelism. In order to achieve good scalability, both these aspects need to be included in the design of a GP system: logical tasks should be efficiently mapped onto threads while computations should exploit locality and vectorized execution to optimize CPU resource usage.

Numerical stability Special attention must be paid to the implementation of statistical methods used by the algorithm. Naive

[1] By "managed" we refer to a language that runs in a virtual environment, be it C#'s CLR, Java's JVM or Python's VM.
[2] SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) are examples of SIMD instruction sets.
[3] A problem is "embarassingly parallel" if it can easily be divided into components that can be executed concurrently, such as fitness evaluation in GP.

implementations of such methods can lead to numerical issues which can distort the results. Schubert and Gertz [26] find that even well-known, high-profile software tools employ unstable numerical methods.

Ideally, the implementer will be able to offload part of the functionality required to run a GP system to specialized support libraries such as Boost [25], Eigen [12] or NumPy [21].

In what follows, we detail our approach towards a scalable, high-performance GP system. We then empirically test its capabilities on a set of symbolic regression benchmark problems and compare it to two other frameworks: DEAP [9] and HeuristicLab [30]. We show that performance benefits up to an order of magnitude can be gained by careful consideration of implementation details without compromising modularity, extensibility or solution quality.

2 GENETIC PROGRAMMING FRAMEWORKS

The software ecosystem surrounding genetic programming already boasts a number of powerful frameworks such as ECJ (Java) [19, 28], HeuristicLab (C#) [30] or DEAP (Python) [9] that offer a fully-featured environment supporting multiple solution representations and algorithms. Many other alternatives are available for every mainstream programming language[4]. Due to space limits as well as time and resource constraints, we restrict our comparison to DEAP and HeuristicLab in this paper. We base our choice on programming experience with the framework's language, ease of use and the existence of examples and documentation.

2.1 HeuristicLab

HeuristicLab [30] is an optimization framework developed since 2002 by the Heuristic and Evolutionary Algorithms Laboratory (HEAL) at the University of Applied Sciences Upper Austria. It includes a wide selection of algorithms and problems[5]. Additionally, the framework offers a GUI (Windows only) that allows to run algorithms, perform experiments and analyze the results.

Programming with HeuristicLab Built upon a collection of core components organized in a plugin-based architecture, HeuristicLab maintains complete separation between the user interface and the business logic. This allowed us to use its C# API to configure experiments for the empirical section of this paper. We were also able to implement a new tree initialization algorithm using existing operators as inspiration. However, since the API is not well documented the programmer sometimes has to look at the source code to figure out some of the finer details. For this work we used the trunk version of HeuristicLab at svn revision r17523.

2.2 DEAP

DEAP[6] [9] (Distributed Evolutionary Algorithms in Python) is an evolutionary computation framework developed at the Computer Vision and Systems Laboratory (CVSL) at Université Laval, in Quebec city, Canada. Its main focus is to provide algorithms that work "out of the box" and are easy to extend.

DEAP supports all basic GP operations and implements a basic non-elitist generational GP as originally described by Koza [18]. Since DEAP's recombination operators are unaware of tree length and depth limits, these are statically enforced by replacing any offspring exceeding the limits with one of its parents.

Surrounded by Python's scientific ecosystem (Pandas, SciPy, NumPy, etc.), the framework makes it easy to organize experiments and run them locally or distributed over a network.

Programming with DEAP We found DEAP easy to extend with help from the documentation and the comments within its source code. We implemented an elitist generational GP and an additional tree initialization procedure with little effort, therefore we would recommend DEAP for its accessible API.

In terms of parallelization, DEAP supports multi-threaded evaluation via Python's multiprocessing module. However, in our experiments we observed better scalability at a coarser level, running multiple DEAP evolutionary processes in parallel. For this paper we worked with DEAP version 1.3.1.

3 THE OPERON FRAMEWORK

In this paper we introduce *Operon*, an efficient GP framework designed from the ground up to take advantage of today's parallel architectures, by introducing a new concurrent evolutionary model based on the concept of an *offspring generator*.

The main motivation behind this framework is on the one hand, to achieve an efficient implementation in terms of runtime and memory, and on the other hand to offer a completely out-of-the-box solution for optimization, where the user can just select a dataset and let the framework handle the rest. The implementation is structured into a core library and a command-line client exposing its functionality to the user.

At the same time, experienced practitioners are offered the possibility to configure almost every aspect of evolution via an intuitive programming interface which follows established conventions (e.g., Koza-style terminology and operators, easy way to define experiments and process results, reproducible runs). An overview of the framework's features is given in Table 1. The source code is available at https://github.com/foolnotion/operon.

3.1 Tree Encoding

The most important part of any GP system is the genotype *encoding* which must be expressive, efficient and easily manipulated by the corresponding genetic operators. We discuss what makes a good encoding from a performance standpoint and introduce a linear representation equivalent to a post-order traversal of the canonical GP tree. Then we describe tree initialization and evaluation procedures based on this encoding.

The main component of the representation is the `Node` structure, defined as a *PODType* (plain old data type), that is a scalar type with standard memory layout, which occupies a contiguous memory area and can be safely copied with `std::memcpy` or serialized to/from binary files. Memory to memory copy is the fastest way of copying objects and can be orders of magnitude faster than manually implemented deep copy/clone routines of non-trivial types. The `Node` structure incorporates only strictly necessary information using scalar data members as shown in Table 2.

In this encoding scheme a tree individual is defined as an array of `Nodes`. A subtree then corresponds to a subarray delimited by

[4] http://geneticprogramming.com/software/
[5] https://dev.heuristiclab.com/trac.fcgi/wiki/Features
[6] https://deap.readthedocs.io

Table 1: Implementation and design details of the *Operon* framework.

Implementation details

- Compact, efficient linear encoding for trees. Direct correspondence to GP tree concept via linear (postfix) indexing scheme.
- Trees represented as contiguous node arrays, with 40 bytes per tree node, promoting memory locality.
- Low memory footprint: 10k trees of length 50 (20k internally for parent and offspring populations) in under 20MiB of memory.
- Logical parallelism: *recombinants* (new offspring) are generated concurrently, the framework handles threads and scheduling.
- Low-overhead synchronization via atomic primitives.
- Ability to evolve very large populations (e.g., 1M individuals on standard hardware)
- Designed as a core library (*liboperon*) and a CLI client (*operon-gp*) easily integrated into e.g. Python or shell scripts

Genetic Programming design

- Multiple GP flavors sharing the same evolutionary main loop, differing in how they define an *offspring generator* concept.
- An offspring generator may fail (returns an *option type*) for configurable reasons such as offspring acceptance criteria.
- *Streaming* genetic operators designed to integrate efficiently with the concurrency model.
- Operators spend from the global evaluation budget, making it easy to implement fair comparisons between algorithmic flavors.
- Operators encapsulate termination criteria (e.g., budget based, selection pressure based, etc.).
- State-of-the-art, numerically-stable statistical methods (mean, variance, correlation).
- Seamless integration with automatic differentiation (same evaluation can work on both scalar and dual number types).
- Local search support (trust region-based fitting of tree numerical coefficients) (also counted out of the global evaluation budget)
- Detailed local search statistics (number of Jacobian evaluations, failed/successful gradient descent steps, convergence status).
- Ability to perform model evaluation in *single-* or *double precision* mode (e.g., using *float* (32-bit) or *double* (64-bit) data types).
- Novel tree initialization (balanced tree creator) able to produce arbitrary distributions of tree sizes and symbol frequencies.

Table 2: The Node data structure.

Node	
`Type`	The node's primitive type
`Arity`	The node primitive's arity
`Length`	The node's length
`Depth`	The node's depth
`Id`	Hash value mapping dataset variables
`Value`	Value for ephemeral constants or variable weights

a beginning and an end index. Operators such as initialization, crossover and mutation are designed to take advantage of this linear structure. We encode tree nodes in postfix order such that during evaluation, child nodes are evaluated before their parents.

Properties such as subtree length, depth and level are efficiently calculated in a single pass over all tree nodes exploiting the arithmetic properties of the linear post-order indexing scheme. They are defined as follows:

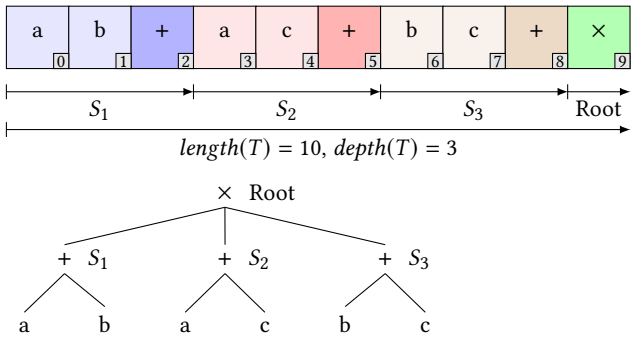
Length We define a `Node`'s length as the total number of nodes in the subtree rooted in that node.

Depth We define a `Node`'s depth as the longest path length between itself and any other descendant node[7]. This property is used by the genetic operators to ensure that the generated offspring do not exceed a static depth limit.

Figure 1 exemplifies the linear encoding for the formula $(a + b) \cdot (a + c) \cdot (b + c)$ which corresponds to the postfix representation `((a b +) (a c +) (b c +) ×)`. Although the formula can be represented in several equivalent ways within the same array length, the encoding is unambiguous due to each node's *arity* and *length* properties.

[7]We use *depth* instead of *height* because this is the accepted terminology in the field of GP cf. Koza and many others.

Figure 1: Linear tree encoding and indexing scheme.



3.2 Tree Initialization

The primitive set $P = F \cup T$ used by the algorithm, where F is the function set and T is the terminal set, is configured using a class that keeps track of allowed node types and their associated probabilities. This offers additional flexibility to the user allowing them to configure desired node frequencies in the initial population.

We introduce an original tree creation algorithm featuring the ability to generate tree lengths from a given probability distribution and fine-tune desired symbol frequencies in the population.

The tree creation procedure illustrated in Algorithm 1 starts from a root node randomly sampled from the primitive set and keeps track of a horizon of expansion points – node child slots yet unfilled by the algorithm. These slots are filled in breadth-first fashion while keeping track of the remaining number of nodes necessary to reach the target tree length. When a function node is added to the tree, it opens a number of new expansion points according to the function's arity. The arity of nodes sampled from the primitive set

is then limited according to the difference between the target length and the number of unfilled expansion points. When the difference becomes zero, the algorithm fills the remaining expansion points with leaf nodes.

Complexity For simplicity, we assume that sampling symbols from G happens in constant time. We assume a tree of size n is generated by the procedure. We can determine an upper bound for the procedure by making the simplifying assumption that every node is a function node. In this case, the complexity is given by $O(\overline{a} \cdot n) \approx O(n)$, where $\overline{a} \ll n$ is the average function arity.

Algorithm 1: Tree initialization

Data: Primitive set $P = F \cup T$.
Input: Target tree length L, tree depth limit D_{max}.
Output: A fully initialized GP tree

1 $a_{min}, a_{max} \leftarrow$ function arity limits as defined by F;
/* Adjust arity limits according to target length L (subtracting the root node) */
2 $a_{min} \leftarrow \min(a_{min}, L-1)$;
3 $a_{max} \leftarrow \min(a_{max}, L-1)$;
/* Sample random node from P within arity limits */
4 $root \leftarrow P(a_{min}, a_{max})$; // may be a leaf node if $L = 1$
5 $tuples \leftarrow [(root, 0)]$; // initialize a list of (node, depth) tuples
6 $S \leftarrow root$.Arity; // number of open expansion points
7 **for** $i = 0; i < L; i \leftarrow i + 1$ **do**
8 $(node, depth) \leftarrow tuples[i]$;
9 $d \leftarrow depth + 1$;
10 **for** $j = 0; j < node$.Arity$; j \leftarrow j + 1$ **do**
/* Calculate new arity limits */
11 $a'_{max} \leftarrow \min(a_{max}, L - S)$ **if** $d < (D_{max} - 1)$ **else** 0;
12 $a'_{min} \leftarrow \min(a_{min}, a'_{max})$;
13 $child \leftarrow P(a'_{min}, a'_{max})$; // Sample random child node from P
14 $tuples \leftarrow tuples + (child, d)$; // Append new child tuple
15 $S \leftarrow S + child$.Arity;
16 $nodes \leftarrow$ extract each node from $tuples$;
17 **return** *new* Tree*(nodes)*;

New node arity limits are computed at each iteration according to the depth limit and the difference $L-S$ between the target length and the running number of expansion points. Since $a'_{min}, a'_{max} > 0$ until limits are reached, trees generated by Algorithm 1 will generally be flat. That is, their depth and nested path length [13] will tend to be minimal (in accordance with expected node arity sampled from function set F). For this reason, we coin the name *Balanced Tree Creator* (BTC) for this algorithm.

To increase shape variability in trees produced by the *BTC* algorithm, we employ an additional tweak: the idea is to allow the possibility to sample a leaf node instead of a function node within the expansion loop, as long as this does not prevent the algorithm from reaching the target length. We realize this idea by changing line 12 of Algorithm 1 such that, when $S > 1$ (we have a sufficient number of available expansion points to go on):

$$a'_{min} = \begin{cases} 0 & \text{with probability } p \\ \min(a_{min}, a'_{max}) & \text{with probability } 1 - p \end{cases}$$

In this context, we call probability p the algorithm's *irregularity bias*, which can be specified by the user.

Speed We use *BTC* to generate 5000 trees of average length 50. On our test machine[8] the algorithm can create $\approx 1.6 \cdot 10^5$ trees/second in single-threaded execution and $\approx 4 \cdot 10^6$ multi-threaded.

3.3 Tree Evaluation

Solution evaluation usually dominates the runtime of an evolutionary algorithm. An *interpreter* design pattern [11] is typically employed to evaluate the tree for each fitness case in the dataset [18]. The interpreter implements logic for applying mathematical operations associated with each function node or loading values from the dataset for input leaf nodes. Interpreter performance is mainly influenced by two factors: *locality* and *data-level parallelism*.

Locality Locality is the tendency of a processor to access the same set of memory locations repetitively over a short period of time [29]. When a tree is interpreted over a large number of fitness cases, the processor's ability to perform *caching*, *prefetching* and *branch prediction* can have a significant impact on performance. Locality can manifest itself both *temporally* and *spatially*. Using a linear tree representation greatly improves spatial locality. Temporal locality can be promoted by grouping together operations of the same type.

Data level parallelism Data level parallelism is achieved by using SIMD instructions to perform the same operation on multiple fitness cases simultaneously. Flow-control-heavy tasks such as tree interpretation may not easily benefit from SIMD and compilers may have difficulties producing vectorized instructions. In this case it is recommended to unroll the loop iterating over the fitness cases and employ vectorization within the loop body by batching together multiple data points. This has the benefit of reducing the overhead of control-flow instructions and facilitating vectorization at the cost of slightly increased memory usage during evaluation.

Suprisingly, only a few other GP implementations employ these techniques [6, 7] on the CPU.

Evaluation procedure
Our tree evaluation procedure makes use of all the techniques enumerated above. The linear data structure where all the Node elements are POD types promotes memory locality. We employ batching to minimize the impact of control-flow instructions and exploit data level parallelism.

The evaluation procedure uses the Eigen library [12] for vectorized arithmetic. During evaluation, a matrix with as many rows as the chosen batch size and as many columns as the number of tree nodes is dynamically allocated. A vector with as many elements as the number of fitness cases is additionally allocated for holding the evaluation results. When possible, allocated memory is aligned to a 32-byte boundary in order to facilitate vectorization.

[8] AMD Ryzen 3900X 12 core/24 threads processor.

Automatic differentiation
In addition to fast calculation of residuals, modern GP systems particularly in the field of System Identification often need to keep track of the first- and second-order derivatives of the model for local optimization based on gradient information [16]. We use the Ceres library[9] [2] for non-linear least squares fitting of model parameters. The library is used at Google to estimate the pose of Street View cars, aircrafts, and satellites; to build 3D models for PhotoTours; to estimate satellite image sensor characteristics, and more. It provides a dual number data type based on *Jets* [15] for computing exact function derivatives. The Jet type supports all operators in the GP primitive set. Thus, by employing C++ templates, we generalize our tree evaluation method to both real and dual-number domains. The user can specify whether the optimized numerical coefficients are written back into the tree individual following a Lamarckian learning model or used only for fitness evaluation following a Baldwinian learning model.

3.4 Offspring Generation

We introduce a streaming selection operator that responds to requests from other components (e.g., the *offspring generator* component), returning an index to the selected individual in the population array. We provide implementations for proportional, tournament and rank-based selection. *Operon* offers the possibility of configuring independent male and female selectors for the two parents involved in crossover, for more flexibility in configuring and testing GP algorithms composed of different combinations of operators.

We define an *offspring generator* as an object that abstracts the algorithm's offpring generation strategy. In this context, a *recombination event* is an event with two possible outcomes: *success* - when a suitable new offspring was able to be produced, and *failure* - when a new offspring could not be produced.

Figure 2 provides a high-level algorithmic outline of GP. The main evolutionary loop (Figure 2a) queries the *offspring generator* (Figure 2b) for new offspring until the new population is filled or termination is triggered. The offspring generator returns a *maybe type*[10] to signal possible *failure*, when a *recombinant* - the "raw" result of recombination - doesn't satisfy acceptance criteria. Typically, the inability to generate successful recombinants signals algorithm termination, as is the case for offspring selection [1].

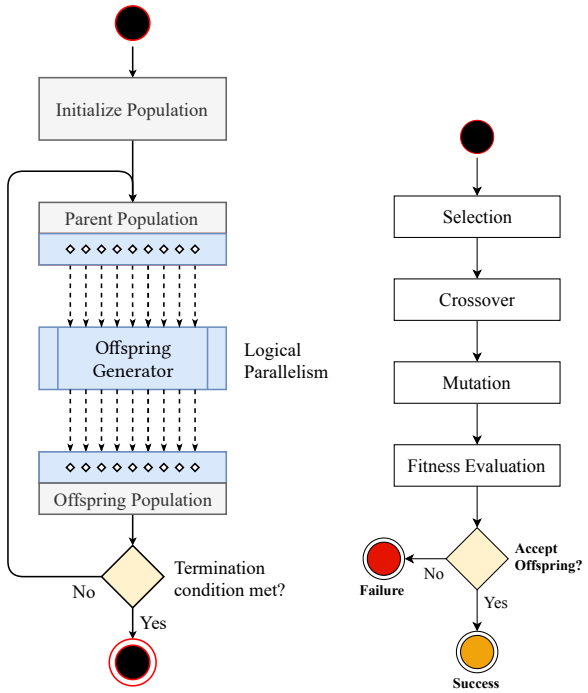
In the case of standard GP, offspring produced through the workflow described by Figure 2b are unconditionally accepted into the new population (*failure* never occurs). For this reason, we say that standard GP uses a *Basic Generator*. We define two additional types of offspring generator which implement different concepts from the literature.

Brood Generator The brood generator implements the concept of *soft brood selection* [3]: each set of parents generates a predetermined number of offspring called a *brood*. A tournament is then held between brood individuals in order to establish the winner.

Offspring Selection Generator New offspring are accepted into the new population if they fulfil some measure of success with regard to their parents [1, 3]. Typically, success is quantified as the child individual having better fitness than both its parents.

[9] http://ceres-solver.org/
[10] A *maybe type* is a polymorphic type that encapsulates an optional value.

Figure 2: *Operon*'s concurrency model employing logical parallelism in the offspring generation phase – Figure 2a. Each logical thread performs the work outlined in Figure 2b.



(a) Evolutionary workflow **(b) Generator workflow**

The design inherently promotes fairness in comparing different offspring generation strategies, since each fitness evaluation of a recombinant consumes from the algorithm's evaluation budget. Different algorithmic variants can be straightforwardly compared by fixing the budget. This rule also applies to hybridizations with local search when evaluating the model with a new set of numerical coefficients also consumes from the evaluation budget.

This affords the user great flexibility in creating different behaviors by employing different selection methods for each parent individual, generating any number of recombinants or combining any acceptance criteria, assigning fixed local evaluation budgets for recombinant trials, and so on.

3.5 Concurrency Model

Parallelism plays an important part in the proposed design, therefore the evolutionary main loop should progress with as little synchronisation effort as possible. For this reason, each recombination event is seen as an independent attempt to produce a new offspring individual, taking place in its own *logical* thread, not sharing any mutable state with other threads. Logical threads can be seen as jobs representing domain-specific work, typically organized into work queues that may be executed in interleaved fashion on a single physical thread depending on available machine resources.

Optimal distribution of logical tasks to physical threads is left to the underlying scheduler. This ensures that the framework behaves consistently and predictably across different machines. We

employ the Thread Building Blocks[11] (TBB) library to specify logical parallelism instead of threads. Its runtime automatically maps concurrent tasks onto threads in a way that makes efficient use of processor resources according to a user-specified execution policy.

Determinism and reproducibility An important aspect to consider in a concurrent model is *deterministic execution*, which requires the absence of data races and commutativity of the critical sections[12]. Determinism allows metaheuristics such as GP to reproduce results by fixing the random number generator seed. Implementation-wise, this is achieved by providing a local preseeded instance of the random generator to each logical thread. Since the number of logical threads stays the same, reproducibility is not affected by the actual number of physical threads available for parallel execution.

In the case of user-defined operators depending on shared state, the library user is responsible for protecting the critical sections to avoid race conditions. More fine-grained algorithmic models could be derived from the default evolutionary model in order to improve efficiency (for example, if only parallel fitness evaluation is desired).

Since offspring generators consume from the global evaluation budget, a synchronization mechanism is necessary to track the number of fitness evaluations and other related progress indicators. This is achieved using atomic variables[13]. As a rule, each genetic operator within the proposed design maintains a minimal, immutable state, with few exceptions for the aforementioned atomic types. Since the main loop is aware of this information, the algorithm can be stopped even when the new offspring population is in the process of being filled if at any point termination is triggered via any of the atomic progress indicators.

3.6 Fitness and Statistical Measures

Implementing the "textbook equation" for common descriptive statistics such as variance, standard deviation, covariance and correlation can lead to numerical instability due to catastrophic cancellation [24]. In this regard, DEAP [9] is backed by the high-quality statistical methods provided by the Numpy [21] framework. HeuristicLab [30] uses a variant of Welford's algorithm [31] described by Knuth in [14]. The *Operon* framework uses numerically-stable, parallelizable, vectorizable variance, covariance and correlation algorithms by Schubert and Gertz [26], ported to C++ from the ELKI Data Mining Library (Java) [27].

4 PERFORMANCE ANALYSIS

We illustrate the performance of the proposed framework using profiling data and evaluation speed benchmarks on a test system with the following hardware:

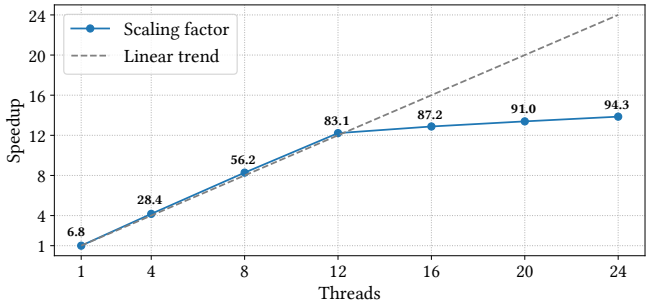
- AMD Ryzen™ 3900X, 12 core/24 thread, 3.8Ghz base frequency, 768Kb/6Mb/64Mb L1/L2/L3 cache.
- 32Gb DDR4-3600 CL16 memory

The code is compiled using the `gcc` compiler with optimizations enabled via compiler flags `-O3` and `-march=znver2`.

[11] https://software.intel.com/en-us/tbb
[12] A critical section or critical region is a section of the program which needs to be protected in ways that avoid the concurrent access.
[13] Atomic types encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads.

Fitness evaluation for the entire population can be executed in parallel by specifying the `std::par_unseq` (parallel unsequenced) execution policy. Functions with this policy are permitted to execute in unordered fashion in unspecified threads and unsequenced with other threads. This enables vectorization and work-stealing, leading to a more efficient usage of CPU resources. This functionality featured in the new `C++17` standard is backed by Intel TBB. Figure 3 shows the benefit of this approach as evaluation speed scales linearly (even slightly superlinearly) with the number of physical cores. Increasing the number of threads to 24 offers a further 13.5% speed increase (from 83.1 to 94.3 GPops/s). For this benchmark, the CPU frequency was locked to 3.8 Ghz to eliminate artifacts due to boosting.

Figure 3: Double precision speed-up vs number of threads for 1000 randomly-generated trees of average length 50 over 5000 data rows, using an arithmetic primitive set (+, −, ×, ÷). The labels show evaluation speed in billion GPops/second.



4.1 Code Profiling Analysis

We use code profiling tools to compare in detail the relative distribution of CPU time among the different GP components such as evaluation, selection, crossover and mutation for each tested GP framework. We also measure memory consumption in terms of *peak resident set size* (RSS), representing the total size in memory (including shared memory e.g. used by system libraries). Additionally for *Operon* we are also able to report *peak heap memory consumption*, representing the maximum amount of memory allocated by the program.

Tools We profile *Operon* using the `Valgrind` [20] instrumentation framework. We use the tools `callgrind` and `massif` to record function call history and memory usage, respectively. For DEAP, we use Python's `cProfile` module and convert the output to the same format using the `pyprof2calltree`[14] tool to get performance metrics and the `psutil`[15] library to retrieve the resident set size. For HeuristicLab, we use the Microsoft Visual Studio profiler and report peak memory usage using the `Process.PeakWorkingSet64` property.

Profiling setup We run a basic GP algorithm using 1000 individuals, 100 generations and maximum tree length 50, with a primitive set composed of the +, −, ×, ÷, exp, log, sin, cos functions. We vary the number of training samples to determine the relative impact of evaluation for different problem sizes.

[14] https://pypi.org/project/pyprof2calltree/
[15] https://pypi.org/project/psutil/

Table 3: Relative runtime impact and peak memory (in MiB)

Rows	Fitness	Crossover	Mutation	Selection	Heap	RSS
Operon						
50	86.80%	10.71%	0.61%	0.96%	3.7	10.3
100	92.50%	6.10%	0.35%	0.54%	3.7	10.3
500	97.52%	2.01%	0.11%	0.18%	3.7	10.4
1000	98.81%	0.97%	0.05%	0.09%	3.8	10.5
2000	99.43%	0.47%	0.03%	0.04%	3.9	10.7
5000	99.73%	0.22%	0.01%	0.02%	4.4	11.6
DEAP						
50	67.22% (47.38%)	1.78%	1.53%	2.26%	-	101.0
100	68.02% (47.36%)	1.70%	1.55%	2.18%	-	101.1
500	69.84% (42.52%)	1.61%	1.50%	2.07%	-	101.3
1000	71.47% (38.53%)	1.55%	1.37%	1.99%	-	101.3
2000	75.96% (30.75%)	1.30%	1.19%	1.66%	-	101.5
5000	80.90% (22.74%)	1.04%	0.95%	1.34%	-	102.2
HeuristicLab						
50	12.47%	18.62%	1.02%	10.69%	-	646.5
100	14.77%	18.26%	1.00%	10.32%	-	647.5
500	30.48%	13.55%	0.76%	7.60%	-	673.0
1000	44.29%	10.81%	0.57%	5.73%	-	685.3
2000	53.36%	7.28%	0.39%	3.93%	-	741.6
5000	68.76%	4.12%	0.22%	2.14%	-	761.6

In comparing runtime it is also important to point out that both DEAP and HeuristicLab implement efficient evaluation routines:

∘ HeuristicLab uses the same evaluation procedure as *Operon*, calling into native C++ code for tree evaluation. However, actual fitness (e.g., R^2 score) is computed on the managed side.

∘ DEAP compiles trees into functions using NumPy as a backend for evaluation. The compilation step has a relatively large overhead, decreasing efficiency for small problems.

Operon. The measurements displayed in Table 3 show that most of the algorithm's runtime is spent in the evaluation routine. The fact that evaluation still makes up for ≈ 87% of the runtime even with only 50 training samples demonstrates the efficiency of the linear encoding and recombination operators. Profiling shows that for large data (5000 rows), fitness calculation accounts for 99% of the algorithm's runtime.

DEAP. The DEAP runtime is also dominated by fitness evaluation. Since its evaluation routine compiles trees into functions, we also include compilation overhead relative to the duration of fitness evaluation. For small problems this overhead accounts for almost half of the evaluation time, decreasing with problem size, but remaining significant (22% for 5000 rows) for larger problems.

The first thing to notice about DEAP measurements is that the percentages do not sum up to 100%. This is because the remaining runtime is distributed among other operations that are hard to present in a structured way, such as deep-cloning of the population or static limit decorators of recombination operators, which compute the relevant limits (ie., depth and length) replacing offending individuals with a randomly selected parent.

HeuristicLab. HeuristicLab employs a meta-algorithm model where all the operators are applied according to the flow specified by an *operator graph* [30]. In this model each algorithmic component (e.g., selection, crossover, mutation, evaluation) is represented by an operator that is preceeded and succeeded by other operators. The operator graph is executed by an *execution engine* which ensures the correct order of execution.

This model offers additional flexibility as it allows users to create new algorithms or adapt existing ones by simply changing the operator graph accordingly, promoting component reusability and avoiding code duplication. However, this leads to a runtime and memory overhead visible in Table 3, as the relative impact of fitness calculation is the lowest and memory usage is increased.

4.2 Empirical analysis

We empirically analyze model quality and runtime performance of DEAP, HeuristicLab and *Operon* on a set of benchmark problems of different sizes (from 250 to 5000 training rows). References for each problem are included in Table 5. Since *Operon* supports both single- and double-precision model evaluation, we include both configurations in the experiment.

We test all three frameworks using their standard GP implementation (for DEAP, slightly extended to support elitism) and use the same configuration options described in Table 4. In an effort to provide the same testing conditions for all frameworks, we implemented the tree initialization method described in Algorithm 1 for DEAP and HeuristicLab. The test system is described in Section 4.

Each configuration was repeated 50 times in order to account for stochastic effects. For each framework we used the more advantageous parallelization model, which in the case of DEAP and HeuristicLab turned out to be coarse-grained parallelization (running multiple algorithmic instances in parallel).

Table 4: Algorithm parameterization

Function set	$+, -, \times, \div$, sin, cos, exp, log
Terminal set	constant, weight · variable
Tree limits	10 levels, 50 nodes
Tree initialization	Balanced tree creator - algorithm 1
Population size	1000 individuals
Generations	1000 generations
Parent selection	Tournament group size 5
Crossover probability	100%
Crossover operator	Subtree crossover
Mutation probability	25%
Mutation operator	Single-point mutation
Fitness function	R^2 correlation with the target

We report results in Table 5 as median values ± interquartile range (IQR). Model errors are expressed as normalized mean squared error NMSE = $\frac{\text{MSE}}{\text{Var}(y)}$, where $\text{Var}(y)$ is the variance of the regression target. The elapsed times of DEAP and HeuristicLab are adjusted by the observed parallel scaling factor for each problem.

The results show that in terms training quality, HeuristicLab and Operon perform comparatively the same, while DEAP lags behind in most problems. Although the differences are small, they are stastistically significant as indicated by a two-sided Mann-Whitney U test with $\alpha = 0.05$[16]. This might be explained by DEAP's handling of tree length and depth limits leading to different run dynamics.

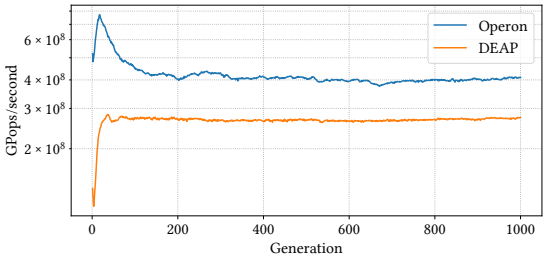
[16]Results and raw data can be downloaded from https://dev.heuristiclab.com/trac.fcgi/wiki/AdditionalMaterial#GECCO2020.

Table 5: Modeling results.

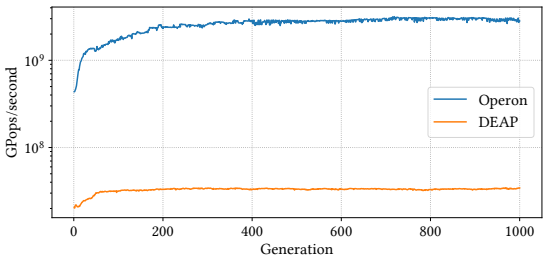
Framework	NMSE (train)	NMSE (test)	Elapsed (s)
Airfoil Self-Noise (1000 training rows) [5]			
Deap	0.294 ± 0.068	0.335 ± 0.107	38.6 ± 4.6
HeuristicLab	0.219 ± 0.035	0.256 ± 0.058	20.5 ± 3.0
Operon (double)	0.242 ± 0.061	0.270 ± 0.084	3.5 ± 3.1
Operon (float)	0.233 ± 0.047	0.262 ± 0.062	1.3 ± 0.5
Breiman-I (5000 training rows) [4]			
Deap	0.114 ± 0.015	0.122 ± 0.015	59.1 ± 15.9
HeuristicLab	0.115 ± 0.046	0.122 ± 0.042	52.1 ± 15.1
Operon (double)	0.106 ± 0.019	0.112 ± 0.020	12.8 ± 8.5
Operon (float)	0.108 ± 0.013	0.115 ± 0.011	5.4 ± 3.4
Chemical-I (711 training rows) [17]			
Deap	0.228 ± 0.024	0.368 ± 0.176	40.0 ± 2.9
HeuristicLab	0.204 ± 0.026	0.324 ± 0.171	18.7 ± 2.4
Operon (double)	0.194 ± 0.025	0.284 ± 0.140	4.2 ± 2.5
Operon (float)	0.194 ± 0.026	0.320 ± 0.189	1.5 ± 1.7
Concrete Compressive Strength (1000 training rows) [33]			
Deap	0.161 ± 0.014	0.576 ± 0.138	36.2 ± 1.6
HeuristicLab	0.151 ± 0.017	0.595 ± 0.191	17.2 ± 1.9
Operon (double)	0.152 ± 0.022	0.630 ± 0.215	2.7 ± 1.2
Operon (float)	0.148 ± 0.014	0.574 ± 0.162	1.0 ± 1.2
Friedman-I (5000 training rows) [10]			
Deap	0.165 ± 0.040	0.158 ± 0.031	55.2 ± 24.7
HeuristicLab	0.138 ± 0.006	0.139 ± 0.005	64.4 ± 13.9
Operon (double)	0.138 ± 0.003	0.139 ± 0.004	31.2 ± 13.7
Operon (float)	0.139 ± 0.006	0.139 ± 0.006	16.6 ± 10.9
Friedman-II (5000 training rows) [10]			
Deap	0.126 ± 0.058	0.129 ± 0.059	62.5 ± 15.5
HeuristicLab	0.041 ± 0.020	0.042 ± 0.022	67.7 ± 11.9
Operon (double)	0.041 ± 0.009	0.042 ± 0.010	29.4 ± 12.4
Operon (float)	0.043 ± 0.015	0.044 ± 0.016	7.7 ± 9.2
GP-Challenge (5000 training rows) [32]			
Deap	0.097 ± 0.013	0.098 ± 0.015	56.2 ± 25.0
HeuristicLab	0.079 ± 0.013	0.079 ± 0.016	60.4 ± 21.6
Operon (double)	0.074 ± 0.010	0.073 ± 0.011	34.5 ± 15.8
Operon (float)	0.076 ± 0.012	0.075 ± 0.012	8.7 ± 14.0
Poly-10 (250 training rows) [23]			
Deap	0.140 ± 0.124	0.182 ± 0.217	40.5 ± 1.8
HeuristicLab	0.170 ± 0.296	0.193 ± 0.403	14.1 ± 0.8
Operon (double)	0.076 ± 0.134	0.089 ± 0.177	0.9 ± 0.5
Operon (float)	0.078 ± 0.138	0.088 ± 0.172	0.5 ± 0.2
Spatial Coevolution (676 training rows) [22]			
Deap	0.003 ± 0.012	0.146 ± 0.258	34.0 ± 6.0
HeuristicLab	0.003 ± 0.005	0.024 ± 0.126	21.3 ± 1.6
Operon (double)	0.001 ± 0.002	0.008 ± 0.032	6.1 ± 1.2
Operon (float)	0.002 ± 0.001	0.005 ± 0.011	4.0 ± 1.4

In terms of runtime, *Operon* is a clear winner. We observe speedups ranging from 2x to 16x depending on problem size, in comparison to the faster of the two other frameworks on each problem. The speed improvements are a result of both faster evaluation and increased efficiency of the evolutionary process as a whole. The results further indicate that one can achieve double the execution speed in single-precision with no detrimental effects on quality. Interestingly, while DEAP is slower than HeuristicLab on the smaller problems, the situation changes for the larger ones where the impact of its tree compilation overhead is minimized, although the difference is less than 10% in DEAP's favor.

Figure 4: DEAP vs Operon evaluation speed



(a) Friedman-I problem (5000 rows)



(b) Poly-10 problem (250 rows)

We investigate the runtime differences between DEAP and *Operon* in more detail by comparing evaluation speed measured during the evolution on the two problems where the difference was the smallest and largest, respectively: Friedman-I and Poly-10. Figure 4 shows a discrepancy between *Operon*'s evaluation speed of $\approx 4.1 \cdot 10^8$ GPops/second on the Friedman-I problem and $2.8 \cdot 10^9$ on the Poly-10 problem. Profiling shows that evaluation speed is affected by different distributions of function primitives in the population. While on the Poly-10 problem the population is dominated by arithmetic symbols, on the Friedman-I problem trigonometric and exponential primitives are more frequent, in which case *Operon* is limited by the speed of standard library math functions. Incidentally, this also explains the spread of *Operon*'s elapsed times.

5 CONCLUSION

With our proposed C++ framework called *Operon*, we demonstrated useful design principles for maximizing memory efficiency and runtime performance, as illustrated empirically in comparison with two well-known frameworks: DEAP and HeuristicLab.

The fact that single-precision evaluation works well with GP falls in line with similar trends in deep learning. Further work will be needed to establish the general applicability of this technique.

The offspring generator concept offers flexibility in defining different GP evolutionary models, but lacks flexibility in defining new types of problems (e.g., classification) and primitive types. In this regard, both DEAP and HeuristicLab offer more comprehensive tools. Improvements in the architecture of the library should improve this aspect in future versions.

Further development work will focus on other problems and algorithms, serialization support and Python interoperability.

REFERENCES

[1] Michael Affenzeller and Stefan Wagner. 2005. Offspring Selection: A New Self-Adaptive Selection Scheme for Genetic Algorithms. In *Adaptive and Natural Computing Algorithms (Springer Computer Science)*, B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele (Eds.). Springer, 218–221.

[2] Sameer Agarwal, Keir Mierle, and Others. 2018. Ceres Solver. http://ceres-solver.org. (2018).

[3] Lee Altenberg. 1994. The Evolution of Evolvability in Genetic Programming. In *Advances in Genetic Programming*, Kenneth E. Kinnear, Jr. (Ed.). MIT Press, Chapter 3, 47–74.

[4] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. 1984. *Classification and Regression Trees*. Taylor & Francis.

[5] Thomas F. Brooks, D. Stuart Pope, and Michael A. Marcolini. 1989. *Airfoil self-noise and prediction*.

[6] Darren M. Chitty. 2012. Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing* 16, 10 (Oct. 2012), 1795–1814. https://doi.org/doi:10.1007/s00500-012-0862-0

[7] Vinícius Veloso de Melo, Álvaro Luiz Fazenda, Léo Françoso Dal Piccol Sotto, and Giovanni Iacca. 2020. A MIMD Interpreter for Genetic Programming. In *Applications of Evolutionary Computation*, Pedro A. Castillo, Juan Luis Jiménez Laredo, and Francisco Fernández de Vega (Eds.). Springer International Publishing, Cham, 645–658.

[8] R. Duncan. 1990. A survey of parallel computer architectures. *Computer* 23, 2 (Feb 1990), 5–16. https://doi.org/10.1109/2.44900

[9] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[10] Jerome H. Friedman. 1991. Multivariate Adaptive Regression Splines. *Ann. Statist.* 19, 1 (03 1991), 1–67. https://doi.org/10.1214/aos/1176347963

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[12] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

[13] Maarten Keijzer and James Foster. 2007. Crossover Bias in Genetic Programming. In *Proceedings of the 10th European Conference on Genetic Programming (Lecture Notes in Computer Science)*, Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar (Eds.), Vol. 4445. Springer, Valencia, Spain, 33–43. https://doi.org/10.1007/978-3-540-71605-1_4

[14] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[15] I. Kolar, P.W. Michor, and J. Slovak. 1993. *Natural Operations in Differential Geometry*. Springer. https://books.google.at/books?id=4m9x1uL71L4C

[16] Michael Kommenda, Bogdan Burlacu, Gabriel Kronberger, and Michael Affenzeller. 2019. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines* (2019), 1–31.

[17] Arthur Kordon. 2008. *Evolutionary Computation in the Chemical Industry*. Vol. 86. 245–262. https://doi.org/10.1007/978-3-540-75771-9_11

[18] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[19] Sean Luke. 1998. ECJ Evolutionary Computation Library. (1998). Available for free at http://cs.gmu.edu/~eclab/projects/ecj/.

[20] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*. San Diego, California, USA, 89–100.

[21] Travis Oliphant. 2006. NumPy: A guide to NumPy. USA: Trelgol Publishing. (2006). http://www.numpy.org/ [Online; accessed <today>].

[22] Ludo Pagie and Paulien Hogeweg. 1997. Evolutionary Consequences of Coevolving Targets. *Evolutionary Computation* 5, 4 (1997), 401–418. https://doi.org/10.1162/evco.1997.5.4.401 arXiv:https://doi.org/10.1162/evco.1997.5.4.401

[23] Riccardo Poli. 2003. A Simple but Theoretically-Motivated Method to Control Bloat in Genetic Programming. In *Genetic Programming*, Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 204–217.

[24] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. 1992. *Numerical Recipes in C*. Cambridge University Press, Cambridge.

[25] Boris Schäling. 2011. *The boost C++ libraries*. Boris Schäling.

[26] Erich Schubert and Michael Gertz. 2018. Numerically Stable Parallel Computation of (Co-)Variance. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management (SSDBM '18)*. ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/3221269.3223036

[27] Erich Schubert and Arthur Zimek. 2019. ELKI: A large open-source library for data analysis - ELKI Release 0.7.5 "Heidelberg". *CoRR* abs/1902.03616 (2019). arXiv:1902.03616 http://arxiv.org/abs/1902.03616

[28] Eric O. Scott and Sean Luke. 2019. ECJ at 20: toward a general metaheuristics toolkit. In *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, Prague, Czech Republic, 1391–1398. https://doi.org/10.1145/3319619.3326865

[29] W. Stallings. 2010. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall. https://books.google.com/books?id=-7nM1DkWb1YC

[30] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. 2012. Architecture and Design of the HeuristicLab Optimization Environment. In *First Australian Conference on the Applications of Systems Engineering, ACASE (Topics in Intelligent Engineering and Informatics)*, Robin Braun, Zenon Chaczko, and Franz Pichler (Eds.), Vol. 6. Springer International Publishing, Sydney, Australia, 197–261. https://doi.org/10.1007/978-3-319-01436-4_10 Selected and updated papers.

[31] B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* (1962), 419–420.

[32] Pawel Widera, Jonathan M. Garibaldi, and Natalio Krasnogor. 2010. GP challenge: evolving energy function for protein structure prediction. *Genetic Programming and Evolvable Machines* 11, 1 (March 2010), 61–88. https://doi.org/doi:10.1007/s10710-009-9087-0

[33] I.-C. Yeh. 1998. Modeling of strength of high-performance concrete using artificial neural networks. *Cement and Concrete Research* 28, 12 (1998), 1797 – 1808. https://doi.org/10.1016/S0008-8846(98)00165-3